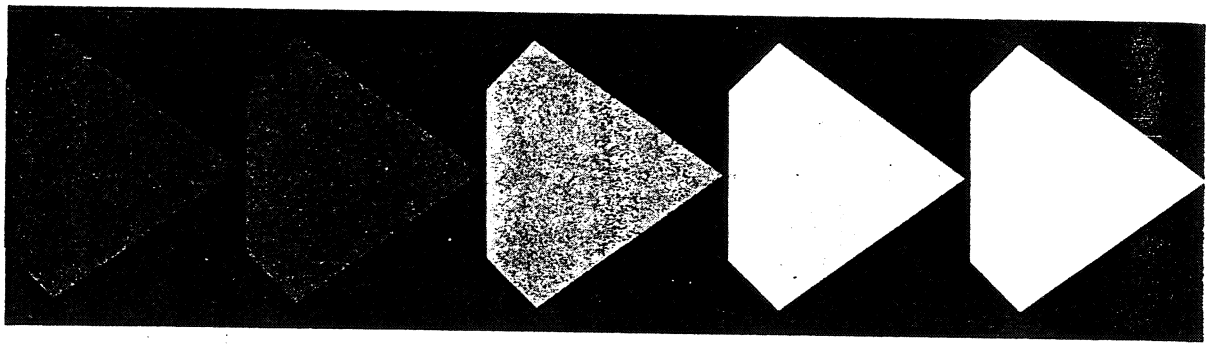
Data Flow

Programmability with increased performance? New strategies to attain this goal include two approaches to data flow architecture: data flow multiprocessors and the cell block architecture.

Data Flow Supercomputers



Jack B. Dennis
MIT Laboratory for Computer Science

The architects of supercomputers must meet three challenges if the next generation of machines is to find productive large-scale application to the important problems of computational physics. First, they must achieve high performance at acceptable cost. Instruction execution rates of a billion floating-point operations each second are in demand, whereas current architectures require intricate programming to attain a fraction of their potential, at best around one tenth of the goal. Brute force approaches to increase the speed of conventional architectures have reached their limit and fail to take advantage of the major recent advances in semiconductor device technology. Second, they must exploit the potential of LSI technology. Novel architectures are needed which use large numbers but only a few different types of parts, each with a high logic-to-pin ratio. In a supercomputer, most of these parts must be productive most of the time; hence the need to exploit concurrency of computation on a massive scale. Third, it must be possible to program supercomputers to exploit their performance potential. This has proven to be an enormous problem, even in the case of computations for which reasonably straightforward Fortran programs exist. Thus present supercomputer architectures have exacerbated rather than resolved the software crisis.

It appears that the objectives of improving programmability and increasing performance are in conflict, and new approaches are necessary. However, any major departure from conventional architectures based on sequential program execution requires that the whole process of program design, structure, and compilation be redone along new lines. One architecture under consideration is a multiprocessor machine made of hundreds of intercommunicating microcomputer processing elements. This architecture has attracted wide interest, but has many drawbacks; even if the processing elements had full floating-point capability and ran at a million instructions per second, at least one thousand would be required to attain a billion instructions per second performance. For such a number of processing elements there is no known way of permitting access to a shared memory without severe performance degradation. Similarly, no known way of arranging conventional microprocessors for synchronization or message passing allows efficient operation while exploiting fine grain parallelism in an application. And finally, there is no programming language or methodology that supports mapping application codes onto such a multiprocessor in a way that achieves high performance.

Language-based computer design can ensure the programmability of a radical architecture. In a language-based design the computer is a hardware interpreter for a specific base language, and programs to be run on the system must be expressed in this language.[1] Because future supercomputers must support massive concurrency to achieve a significant increase in performance, a base language for supercomputers must allow expression of concurrency of program execution on a large scale. Since conventional languages such as Fortran are based on a global state model of computer operation, these languages are unsuitable for the next generation of supercomputers and will eventually be abandoned for large-scale scientific computation. At present, functional or applicative programming languages and data flow models of computation are the only known foundation appropriate for a supercomputer base language. Two programming languages have been designed recently in response to the need for an applicative programming language suitable for scientific numerical computation: ID, developed at Irvine,[2] and Val, designed at MIT.[3,4]

Data flow architectures offer a possible solution to the problem of efficiently exploiting concurrency of computation on a large scale, and they are compatible with

0018-9162/80/1100-0048$00.75

modern concepts of program structure. Therefore, they should not suffer so much from the difficulties of programming that have hampered other approaches to highly parallel computation.

The data flow concept is a fundamentally different way of looking at instruction execution in machine-level programs—an alternative to sequential instruction execution. In a data flow computer, an instruction is ready for execution when its operands have arrived. There is no concept of control flow, and data flow computers do not have program location counters. A consequence of data-activated instruction execution is that many instructions of a data flow program may be available for execution at once. Thus, highly concurrent computation is a natural consequence of the data flow concept.

The idea of data-driven computation is old,[5,6] but only in recent years have architectural schemes with attractive anticipated performance and the capability of supporting a general level of user language been developed. Work on data-driven concepts of program structure and on the design of practical data-driven computers is now in progress in at least a dozen laboratories in the US and Europe. Several processors with data-driven instruction execution have been built, and more hardware projects are being planned. Most of this work on architectural concepts for data flow computation is based on a program representation known as data flow program graphs[7] which evolved from work of Rodriguez,[8] Adams,[9] and Karp and Miller.[10] In fact, data flow computers are a form of language-based architecture in which program graphs are the base language. As shown in Figure 1, data flow program graphs serve as a formally specified interface between system architecture on one hand and user programming language on the other. The architect's task is to define and realize a computer system that faithfully implements the formal behavior of program graphs; the language implementer's task is to translate source language programs into their equivalent as program graphs.

The techniques used to translate source language programs into data flow graphs[11] are similar to the methods used in conventional optimizing compilers to analyze the paths of data dependency in source programs. High-level programming languages for data flow computation should be designed so it is easy for the translator to identify data dependence and generate program graphs that expose parallelism. The primary sources of difficulty are unrestricted transfer of control and the "side effects" resulting from assignment to a global variable or input arguments of a procedure. Removal of these sources of difficulty not only makes concurrency easy to identify, it also improves program structure. Programs are more modular, and are easier to understand and verify. The implications of data flow for language designers are discussed by Ackerman.[12]

This article presents two architectures from the variety of schemes devised to support computations expressed as data flow graphs. First we explain data flow graphs by examples, and show how they are represented as collections of activity templates. Next we describe the basic instruction-handling mechanism used in most current projects to build prototype data flow systems. Then we develop the two contrasting architectures and discuss the reasons for their differences—in particular the different approaches to communicating information between parts of a data flow machine.

Data flow programs

A data flow program graph is made up of actors connected by arcs. One kind of actor is the operator shown in Figure 2, drawn as a circle with a function symbol written inside—in this case +, indicating addition. An operator also has input arcs and output arcs which carry tokens bearing values. The arcs define paths over which values from one actor are conveyed by tokens to other actors. Tokens are placed on and removed from the arcs of a program graph according to firing rules, which are illustrated for an operator in Figure 3. To be enabled, tokens must be present on each input arc, and there must be no token on any output arc of the actor. Any enabled actor may be fired. In the case of an operator, this means removing one token from each input arc, applying the specified function to the values carried by those tokens, and placing tokens labeled with the result value on the output arcs.

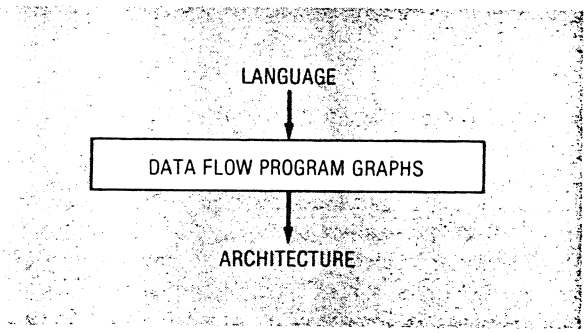


Figure 1. Program graphs as a base language.

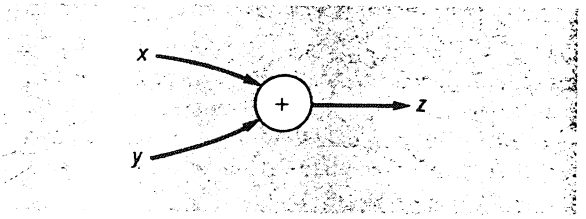


Figure 2. Data flow actor.

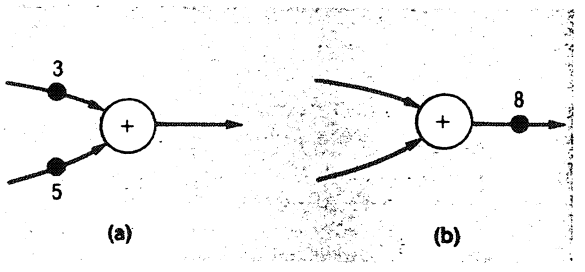


Figure 3. Firing rule: (a) before; (b) after.

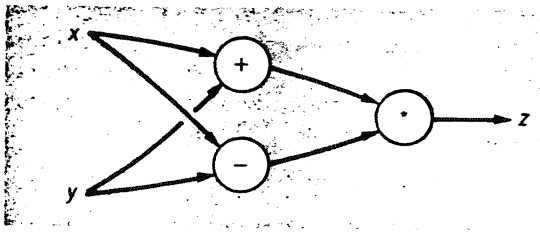


Figure 4. Interconnection of operators.

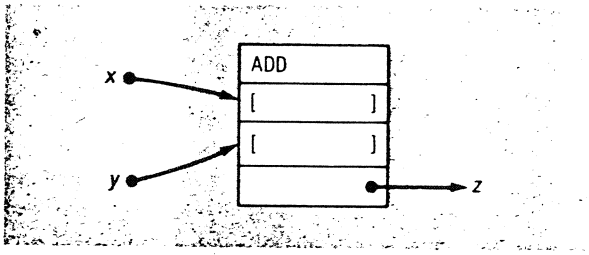


Figure 5. An activity template.

Operators may be connected as shown in Figure 4 to form program graphs. Here, presenting tokens bearing values for x and y at the two inputs will enable computation of the value

$$z = (x+y)*(x-y)$$

by the program graph, placing a token carrying the result value on output arc z.

Another representation for data flow programs—one much closer to the machine language used in prototype data flow computers—is useful in understanding the working of these machines. In this scheme, a data flow program is a collection of activity templates, each corresponding to one or more actors of a data flow program graph. An activity template corresponding to the plus operator (Figure 2) is shown in Figure 5. There are four fields: an operation code specifying the operation to be performed; two receivers, which are places waiting to be filled in with operand values; and destination fields (in this case one), which specify what is to be done with the result of the operation on the operands.

An instruction of a data flow program is the fixed portion of an activity template. It consists of the operation code and the destinations; that is,

instruction:

<opcode, destinations>

Figure 6 shows how activity templates are joined to represent a program graph, specifically the composition of operators in Figure 4. Each destination field specifies a target receiver by giving the address of some activity template and an input integer specifying which receiver of the template is the target; that is,

destination:

<address, input>

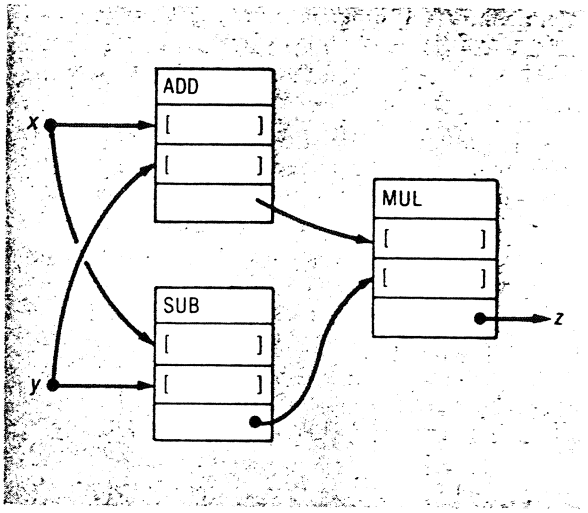


Figure 6. Configuration of activity templates for the program graph of Figure 4.

Program structures for conditionals and iteration are illustrated in Figures 7 and 8. These use two new data flow actors, switch and merge, which control the routing of data values. The switch actor sends a data input to its T or F output to match a true or false boolean control input. The merge actor forwards a data value from its T or F input according to its boolean input value. The conditional program graph and implementation in Figure 7 represent computation of

$$y := (\text{IF } x>3 \text{ THEN } x+2 \text{ ELSE } x-1)*4$$

and the program graph and implementation in Figure 8 represent the iterative computation

$$\text{WHILE } x>0 \text{ DO } = x-3$$

Execution of a machine program consisting of activity templates is viewed as follows. The contents of a template activated by the presence of an operand value in each receiver take the form

operation packet:

<opcode, operands, destinations>

Such a packet specifies one result packet of the form

result packet:

<value, destination>

for each destination field of the template. Generation of a result packet, in turn, causes the value to be placed in the receiver designated by its destination field.

Note that this view of data flow computation does not explicitly honor the rule of program graphs that tokens must be absent from the output arcs of an actor for it to fire. Yet there are situations where it is attractive to use a program graph in pipelined fashion, as illustrated in Figure 9a. Here, one computation by the graph has produced the value 6 on arc z while a new computation represented

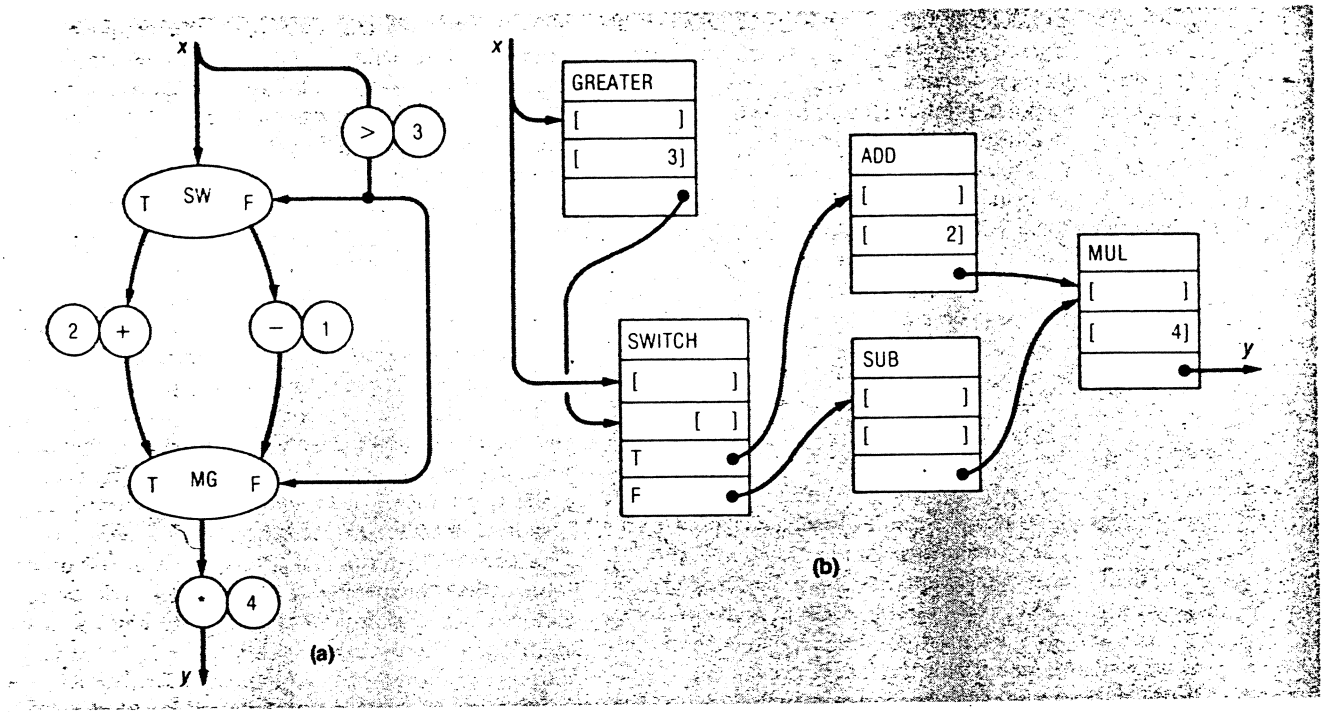


Figure 7. A conditional schema (a) and its implementation (b).

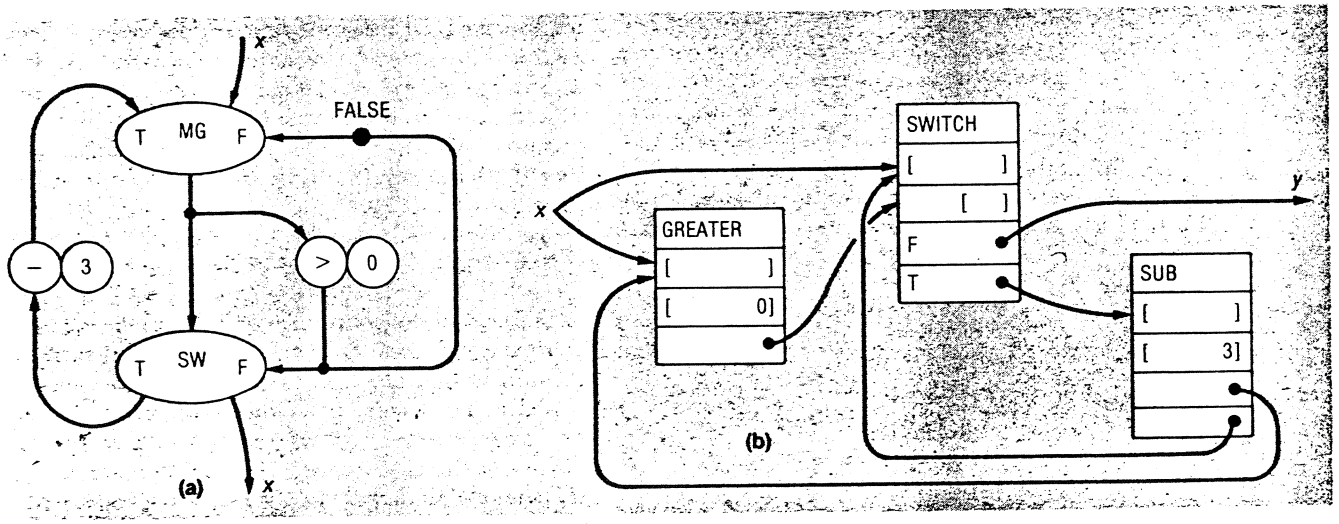


Figure 8. An iterative schema (a) and its implementation (b).

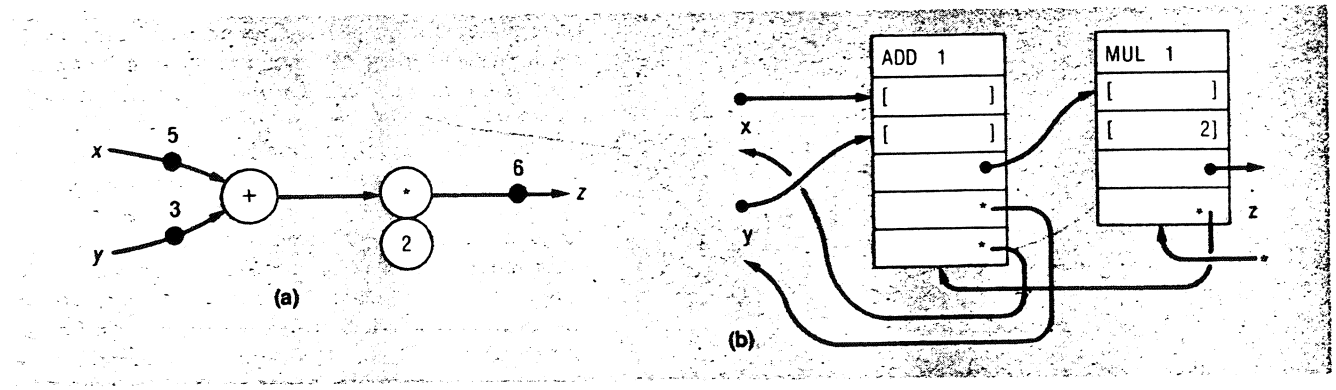


Figure 9. Pipelining in a data flow program (a) and its implementation (b).

by input values 5 and 3 on arcs x and y is ready to begin. To faithfully implement this computation, the add instruction must not be reactivated until its previous result has been used by the multiply instruction. This constraint is enforced through use of acknowledge signals generated by specially marked designations (*) in an activity template. Acknowledge signals, in general, are sent to the templates that supply operand values to the activity template in question (Figure 9b). The enabling rule now requires that all receivers contain values, and the required number of acknowledge signals have been received. This number (if nonzero) is written adjacent to the opcode of an activity template.

The basic mechanism

The basic instruction execution mechanism used in several current data flow projects is illustrated in Figure 10. The data flow program describing the computation to be performed is held as a collection of activity templates in the activity store. Each activity template has a unique address which is entered in the instruction queue unit (a FIFO buffer store) when the instruction is ready for execution. The fetch unit takes an instruction address from the instruction queue and reads the activity template from the activity store, forms it into an operation packet, and passes it on to the operation unit. The operation unit performs the operation specified by the operation code on the operand values, generating one result packet for each destination field of the operation packet. The update unit receives result packets and enters the values they carry into operand fields of activity templates as specified by their destination fields. The update unit also tests whether all operand and acknowledge packets required to activate the destination instruction have been received and, if so, enters the instruction address in the instruction queue. During program execution, the number of entries in the instruction queue measures the degree of concurrency present in the program. The basic mechanism of Figure 10 can exploit this potential to a limited but significant degree: once the fetch unit has sent an operation packet off to the operation unit, it may immediately read another entry from the instruction queue without waiting for the instruction previously fetched to be completely processed. Thus a continuous stream of operation packets may flow from the fetch unit to the operation unit so long as the instruction queue is not empty.

This mechanism is aptly called a circular pipeline—activity controlled by the flow of information packets traverses the ring of units leftwise. A number of packets may be flowing simultaneously in different parts of the ring on behalf of different instructions in concurrent execution. Thus the ring operates as a pipeline system with all of its units actively processing packets at once. The degree of concurrency possible is limited by the number of units on the ring and the degree of pipelining within each unit. Additional concurrency may be exploited by splitting any unit in the ring into several units which can be allocated to concurrent activities. Ultimately, the level of concurrency is limited by the capacity of the data paths connecting the units of the ring. This basic mechanism is essentially that implemented in a prototype data flow processing element built by a group at the Texas Instruments Company.[13] The same mechanism, elaborated to handle data flow procedures, was described earlier by Rumbaugh,[14] and a new project at Manchester University uses another variation of the same scheme.[15]

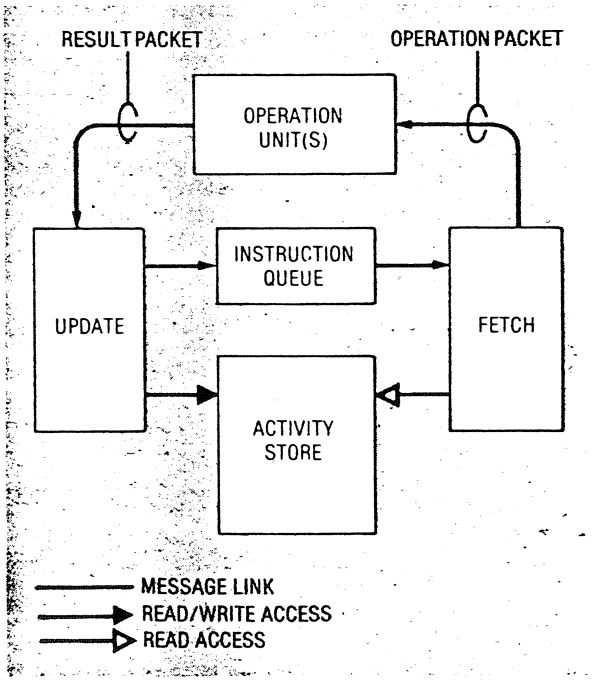


Figure 10. Basic instruction execution mechanism.

The data flow multiprocessor

The level of concurrency exploited may be increased enormously by connecting many processing elements of the form we have described to form a data flow multiprocessor system. Figure 11a shows many processing elements connected through a communication system, and Figure 11b shows how each processing element relates to the communication system. The data flow program is divided into parts which are distributed over the processing elements. The activity stores of the processing elements collectively realize a single large address space, so the address field of a destination may select uniquely any activity template in the system. Each processing element sends a result packet through the communication network if its destination address specifies a nonlocal activity template, and to its own update unit otherwise.

The communication network is responsible for delivering each result packet received to the processing element that holds the target activity template. This network, called a routing network, transmits each packet arriving at an input port to the output specified by information contained in the packet. The requirements of a routing network for a data flow multiprocessor differ in two important ways from those of a processor/memory switch for a conventional multiprocessor system. First, information flow in a routing network is in one direction—an im-

mediate reply from the target unit to the originating unit is not required. Second, since each processing element holds many enabled instructions ready for processing, some delay can be tolerated in transmission of result packets without slowing down the overall rate of computation.

The crossbar switch in conventional multiprocessor systems meets requirements for immediate response and small delay by providing for signal paths from any input to any output. These paths are established on request and maintained until a reply completes a processor/memory transaction. This arrangement is needlessly expensive for a data flow multiprocessor, and a number of alternative network structures have been proposed. The ring form of communication network is used in many computer networks, and has been used by Texas Instruments to couple four processing elements in their prototype data flow computer. The drawback of the ring is that delay grows linearly with size, and there is a fixed bound on capacity.

Several groups have proposed tree-structured networks for communicating among processing elements.[16,17,18] Here, the drawback is that traffic density at the root node may be unacceptably high. Advantages of the tree are that the worst case distance between leaves grows only as $\log_2 N$ (for a binary tree), and many pairs of nodes are connected by short paths.

The packet routing network shown in Figure 12 is a structure currently attracting much attention. A routing network with N input and N output ports may be assembled from $(N/2)\log_2(N)$ units, each of which is a 2×2 router. A 2×2 router receives packets at two input ports and transmits each received packet at one of its output ports according to an address bit contained in the packet. Packets are handled first come, first served, and both output ports may be active concurrently. Delay through an $N \times N$ network increases as $\log_2 N$, and capacity rises nearly linearly with N. This form of routing network is described in Leung[19] and Tripathi and Lipovski.[20] Several related structures have been analyzed for capacity and delay.[21]

The cell block architecture

In a data flow multiprocessor (Figure 11), we noted the problem of partitioning the instructions of a program among the processing elements to concentrate communication among instructions held in the same processing element. This is advantageous because the time to transport a result packet to a nonlocal processor through the routing network will be longer (perhaps much longer) than the time to forward a result locally.

At MIT, an architecture has been proposed in response to an opposing view: each instruction is equally accessible to result packets generated by any other instruction, regardless of where they reside in the machine.[22,23] The structure of this machine is shown in Figure 13. The heart of this architecture is a large set of instruction cells, each of which holds one activity template of a data flow program. Result packets arrive at instruction cells from the distribution network. Each instruction cell sends an operation packet to the arbitration network when all operands and signals have been received. The function of the

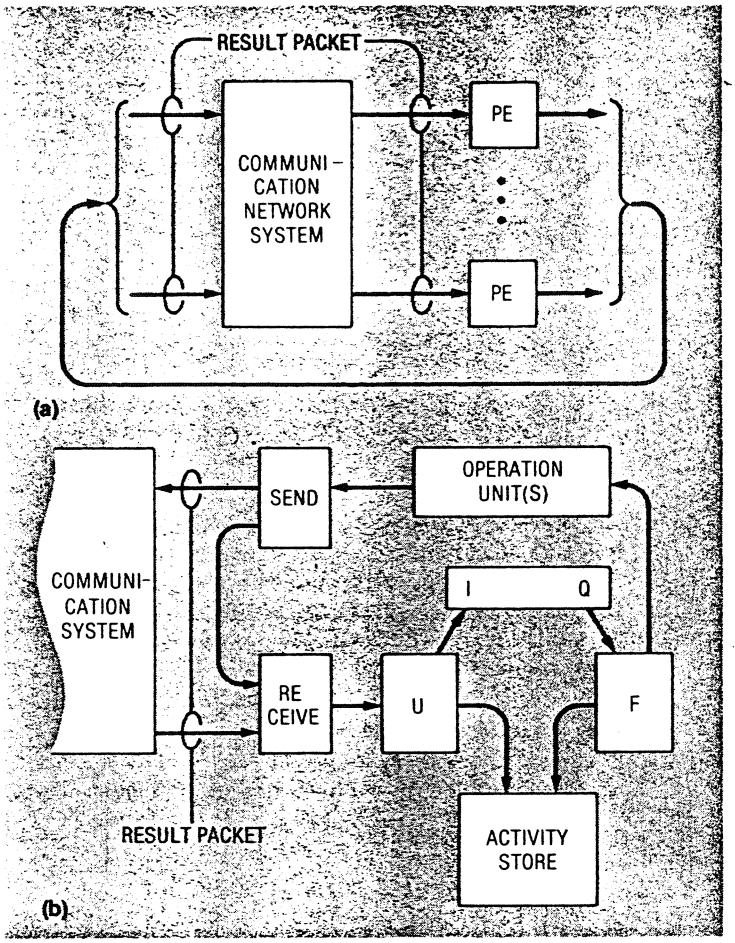


Figure 11. Data flow multiprocessor: (a) connection of many processing elements through a communication system; (b) relationship of each PE to the communication system.

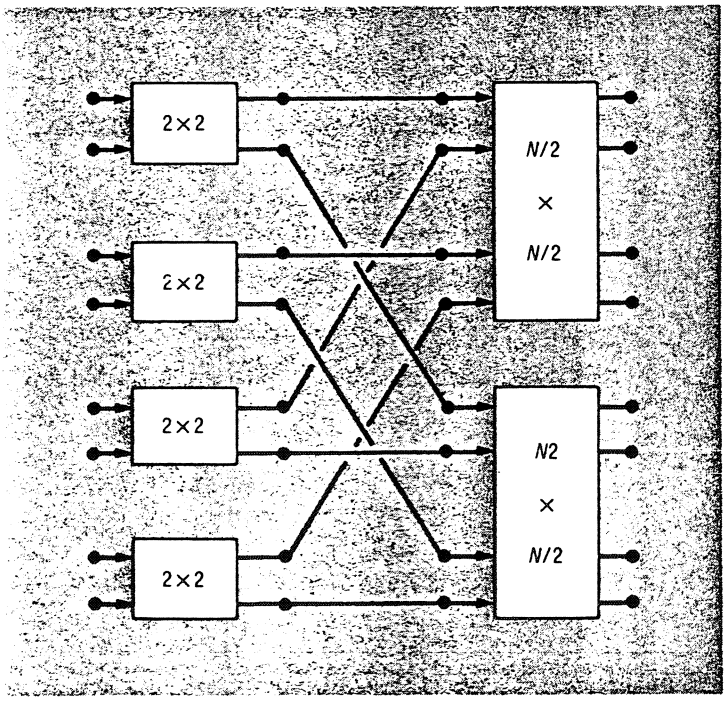


Figure 12. Routing network structure.

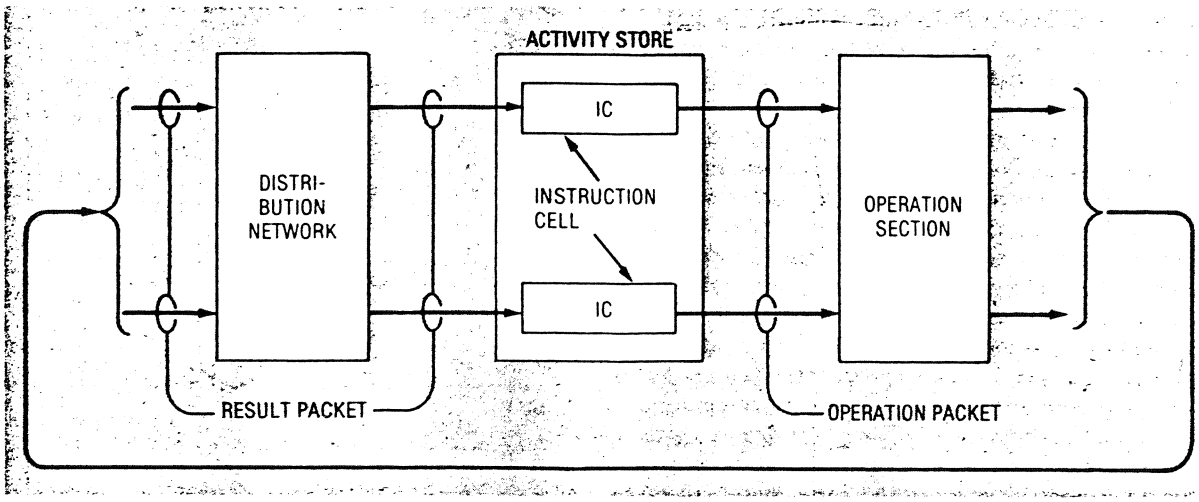


Figure 13. Genesis of the cell block architecture.

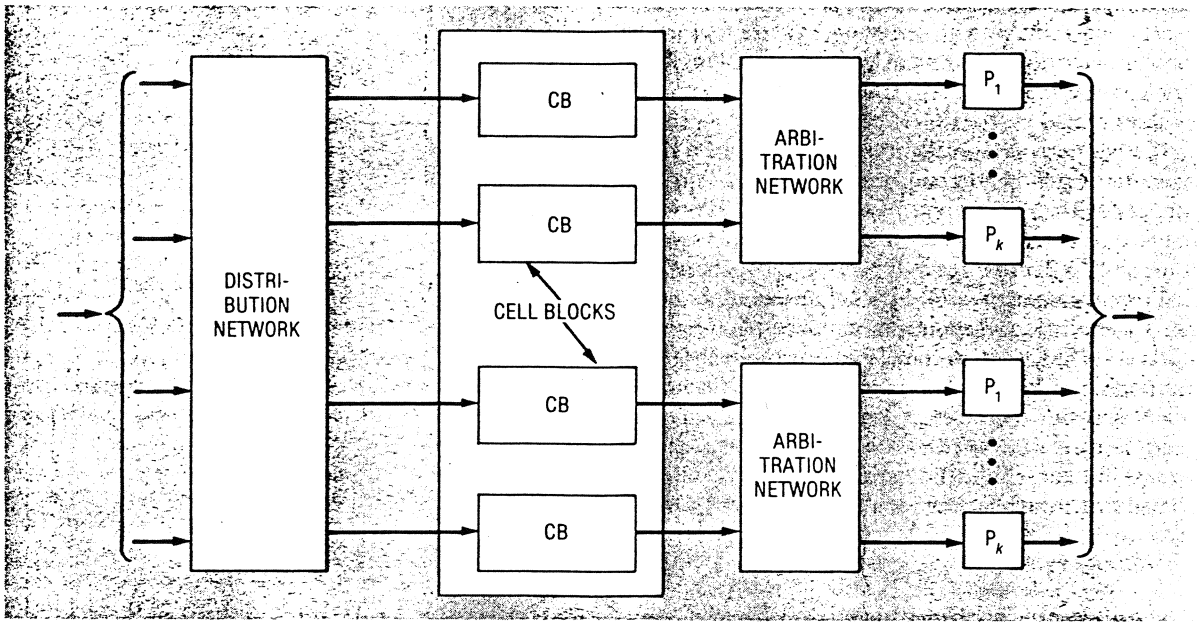


Figure 14. Practical form of the cell block architecture.

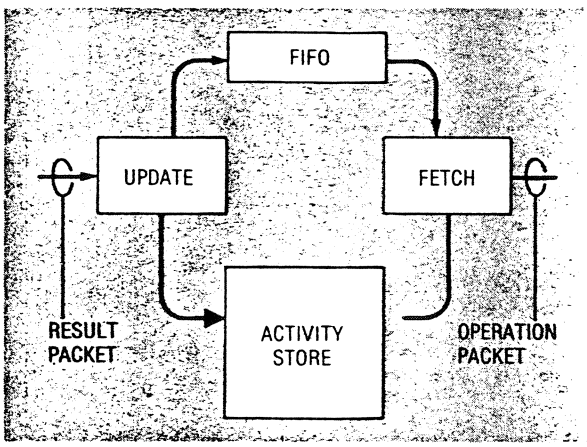


Figure 15. Cell block implementation.

operation section is to execute instructions and to forward result packets to target instructions by way of the distribution network.

The design in Figure 13 is impractical if the instruction cells are fabricated as individual physical units, since the number of devices and interconnections would be enormous. A more attractive structure is obtained if the instruction cells are grouped into blocks and each block realized as a single device. Such an instruction cell block has a single input port for result packets and a single output port for operation packets. Thus one cell block unit replaces many instruction cells and the associated portion of the distribution network. Moreover, a byte-serial format for result and operation packets further reduces the number of interconnections between cell blocks and other units.

The resulting structure is shown in Figure 14. Here, several cell blocks are served by a shared group of functional units $P_i, \ldots, P_k$. The arbitration network in each section of the machine passes each operation packet to the appropriate functional unit according to its opcode. The number of functional unit types in such a machine is likely to be small (four, for example), or just one universal functional unit type might be provided, in which case the arbitration network becomes trivial.

The relationship between the cell block architecture and the basic mechanism described earlier becomes clear when one considers how a cell block unit would be constructed. As shown in Figure 15, a cell block would include storage for activity templates, a buffer store for addresses of enabled instructions, and control units to receive result packets and transmit operation packets. These control units are functionally equivalent to the fetch and update units of the basic mechanism. The cell block differs from the basic data flow processing element in that the cell block contains no functional units, and there is no shortcut for result packets destined for successor instructions held in the same cell block.

Discussion and conclusions

In the cell block architecture, communication of a result packet from one instruction to its successor is equally easy (or equally difficult, depending on your point of view) regardless of how the two instructions are placed within the entire activity store of the machine. Thus the programmer need not be concerned that his program might run slowly due to an unfortunate distribution of instructions in the activity store address space. In fact, a random allocation of instructions may prove to be adequate.

In the data flow multiprocessor, communication between two instructions is much quicker if these instructions are allocated to the same processing element. Thus a program may run much faster if its instructions are clustered to minimize communication traffic between clusters and each cluster is allocated to one processing element. Since it will be handling significantly less packet traffic, the communication network of the data flow multiprocessor will be simpler and less expensive than the distribution network in the cell block architecture. Whether the cost reduction justifies the additional programming effort is a matter of debate, contingent on the area of application, the technology of fabrication, and the time frame under consideration.

Although the routing networks in the two forms of data flow processor have a much more favorable growth of logic complexity ($N \log N$) with increasing size than the switching networks of conventional multiprocessor systems, their growth is still more than linear. Moreover, in all suggested physical structures for $N \times N$ routing networks, the complexity as measured by total wire length grows as $O(N^2)$. This fact shows that interconnection complexity still places limits on the size of practical multiunit systems which support universal intercommunication. If we need still larger systems, it appears we must settle for arrangements of units that only support communication with immediate neighbors.

The advantage data flow architectures have over other approaches to high-performance computation is that the scheduling and synchronization of concurrent activities are built in at the hardware level, enabling each instruction execution to be treated as an independent concurrent action. This allows efficient fine grain parallelism, which is precluded when the synchronization and scheduling functions are realized in software or microcode. Furthermore, there are well-defined rules for translating high-level programs into data flow machine code.

What are the prospects for data flow supercomputers? Machines based on either of the two architectures presented in this paper could be built today. A machine having up to 512 processing elements or cell blocks seems feasible. For example, a 4×4 router for packets, each sent as a series of 8-bit bytes, could be fabricated as a 100-pin LSI device, and fewer than one thousand of these devices could interconnect 512 processing elements or cell blocks. If each processing unit could operate at two million instructions per second, the goal of a billion instructions per second would be achieved.

Yet there are problems to be solved and issues to be addressed. It is difficult to see how data flow computers could support programs written in Fortran without restrictions on and careful tailoring of the code. Study is just beginning on applicative languages like Val and ID.[24,25] These promise solutions to the problems of map-

ping high-level programs into machine-level programs that effectively utilize machine resources, but much remains to be done. Creative research is needed to handle data structures in a manner consistent with principles of data flow computation. These are among the problems under study in our data flow project at MIT. ■

Acknowledgment

This paper is based on research supported by the Lawrence Livermore National Laboratory of the University of California under contract 8545403.

References

1. J.B. Dennis, "On the Design and Specification of a Common Base Language," *Proc. Symp. Computers and Automata,* Polytechnic Press, Polytechnic Institute of Brooklyn, Apr. 1971, pp. 47-74.
2. Arvind, K.P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine,* Dept. of Information and Computer Science, University of California, Irvine, Technical Report 114a, Dec. 1978, 97 pp.
3. W.B. Ackerman and J.B. Dennis, *VAL: A Value Oriented Algorithmic Language, Preliminary Reference Manual,* Laboratory for Computer Science, MIT, Technical Report TR-218, June 1979, 80 pp.
4. J.R. McGraw, *Data Flow Computing: The VAL Language,* submitted for publication.
5. R.R. Seeber and A.B. Lindquist, "Associative Logic for Highly Parallel Systems," *AFIPS Conf. Proc,* 1963, pp. 489-493.
6. R.M. Shapiro, H. Saint, and D.L. Presberg, *Representation of Algorithms as Cyclic Partial Orderings,* Applied Data Research, Wakefield, Mass., Report CA-7112-2711, Dec. 1971.
7. J.B. Dennis, "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Sci.,* Vol. 19, Springer-Verlag, 1974, pp. 362-376.
8. J.E. Rodriguez, *A Graph Model for Parallel Computation,* Laboratory for Computer Science, MIT, Technical Report TR-64, Sept. 1969, 120 pp.
9. D.A. Adams, *A Computation Model With Data Flow Sequencing,* Computer Science Dept., School of Humanities and Sciences, Stanford University, Technical Report CS 117, Dec. 1968, 130 pp.
10. R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Applied Math.,* Vol. 14, Nov. 1966, pp. 1390-1411.
11. J.D. Brock and L.B. Montz, "Translation and Optimization of Data Flow Programs," *Proc. 1979 Int'l Conf. on Parallel Processing,* Bellaire, Mich., Aug. 1979, pp. 46-54.
12. W.B. Ackerman, "Data Flow Languages," *AFIPS Conf. Proc.,* Vol. 48, 1979 NCC, New York, June 1979, pp. 1087-1095.
13. M. Cornish, private communication, Texas Instruments Corp., Austin, Tex.
14. J.E. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Trans. Computers,* Vol. C-26, No. 2, Feb. 1977, pp. 138-146.
15. I. Watson and J. Gurd, "A Prototype Data Flow Computer With Token Labelling," *AFIPS Conf. Proc.,* 1979 NCC, New York, June 1979, pp. 623-628.
16. A. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," *AFIPS Conf. Proc.,* Vol. 48, 1979 NCC, New York, June 1979, pp. 1079-1086.
17. A. Despain and D. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," *Proc. Fifth Annual Symp. Computer Architecture,* Apr. 1978, pp. 144-150.
18. R.M. Keller, G. Lindstrom, and S.S. Patil, "A Loosely-Coupled Applicative Multi-processing System," *AFIPS Conf. Proc.,* 1979 NCC, New York, June 1979, pp. 613-622.
19. C. Leung, *On a Design Methodology for Packet Communication Architectures Based on a Hardware Design Language,* submitted for publication.
20. A.R. Tripathi and G.J. Lopovski, "Packet Switching in Banyan Networks," *Proc. Sixth Annual Symp. Computer Architecture,* Apr. 1979, pp. 160-167.
21. G.A. Boughton, *Routing Networks in Packet Communication Architectures,* MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, June 1978, 93 pp.
22. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Proc. Second Annual Symp. Computer Architecture,* Houston, Tex., Jan. 1975, pp. 126-132.
23. J.B. Dennis, C.K.C. Leung, and D.P. Misunas, *A Highly Parallel Processor Using a Data Flow Machine Language,* Laboratory for Computer Science, MIT, CSG Memo 134-1, June 1979, 33 pp.
24. Arvind and R.E. Bryant, "Design Considerations for a Partial Differential Equation Machine," *Proc. Computer Information Exchange Meeting,* Livermore, Calif., Sept. 1979, pp. 94-102.
25. L. Montz, *Safety and Optimization Transformation for Data Flow Programs,* MS Thesis, MIT, Dept. of Electrical Engineering and Computer Science, Feb. 1980, 77 pp.

Jack B. Dennis, professor of electrical engineering and computer science at MIT, leads the Computation Structures Group of MIT's Laboratory for Computer Science, which is developing language-based computer system architectures that exploit high levels of concurrency through use of data flow principles. Associated with the laboratory since its inception in 1963 as Project MAC, Dennis assisted in the specification of advanced computer hardware for timesharing and was responsible for the development of one of the earliest timeshared computer installations.

Dennis received his DSc degree in electrical engineering from MIT in 1958. He is a member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi, and is a fellow of the IEEE.

DEC halts cash for Dataflow research

By Alex Garrett

Digital Equipment's (DEC) failure to renew support for Manchester University's fifth generation Dataflow project has caused disappointment and speculation about future funding.

The one-year collaboration expired in July this year and Ian Watson, one of the project's co-founders, said he was disappointed because 'there were people within DEC who seemed to be taking it very seriously'.

John Gurd, who runs the University's Dataflow Research Group speculated that the company may be cutting back its funding for academic support but this was denied by a company spokeswoman.

She said there was 'no specific reason' for ending the collaboration, which DEC supported with a significant grant to buy equipment.

It is expected that DEC will make a number of announcements before Christmas on new joint research ventures with universities.

The project at Manchester has received a lot of attention because of the importance of dataflow architecture for fifth generation development projects.

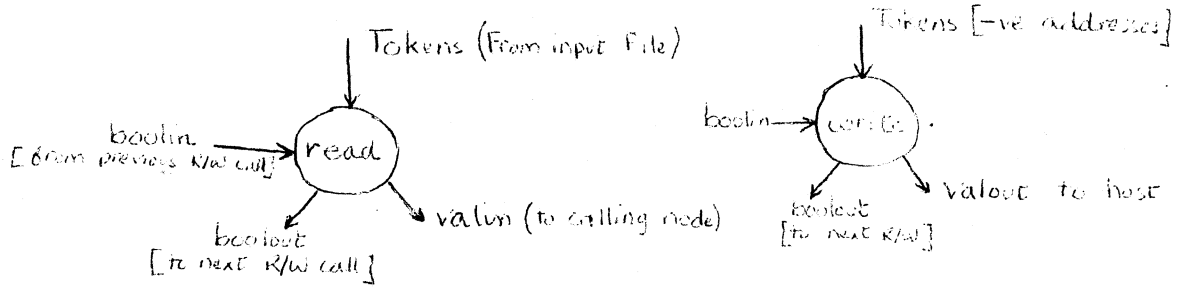
Professor Arvind, who leads a dataflow research unit at Massachusetts Institute of Technology estimated that 70% of such projects are based on dataflow, but said that no usable machine is likely to be available for at least two years.

Sequencing of input/output in dataflow

KBI: 11/2/'81

Consider a graph representing a simple program that reads some data A, B, C, computes functions F1 and F2 and outputs the results of the computations. If the functions F1 and F2 are independent, their evaluations in a dataflow machine may proceed in parallel. As a result, the order in which the global inputs A, B, C will be required and the global outputs produced (which depends on the relative computing times of F1 and F2) is undefined. Therefore the programmer must be allowed to specify an ordering of input and output. This means that the read and write routine activations are not independent (unravelled), as they must be performed sequentially.

The mechanism adopted by John Glauert is to allow the programmer to enforce a sequence by special boolean control parameters as shown below:



Although this method lends itself to both dataflow principles (R/W 'nodes' 'fire' only when inputs available) and to single assignment ~~values~~ rules (order of R/W statements is immaterial and cannot affect the results) there are some problems associated with it:

- The programmer has to declare boolean parameters which are not essential to the execution of the program. If the programmer wishes to alter the sequence, he may have to change many program lines, possibly the declaration part.
- All the tokens of the input file must be in the processing ring before the program can start.
- A most serious restriction is that the read/write statements are permitted only at the main program level - (when inputting text, for instance, a 'read' within a loop is extremely useful, if not necessary!)

The provision of streams and stream operators in LAPSE may reduce the overhead when dealing with the communication and transfer of data between the processing ring and the outside world. With this addition in the language,

the programmer has the facility to declare and manipulate unbound streams of tokens. Structured types such as records, arrays and sets can be viewed as special streams of information. The elements of a stream, inside the ring, will be subscripted using their indices.

It is thought that if the LAPSE commands READ and WRITE adequately operate on streams only, the notion of streams may be used to advantage to solve the problems of input/output.

Let's assume that the following operations of READ and WRITE are provided by the machine:

* READ(X,infile) — X must be a stream. Its elements $<x_k, 1<k<m>$ are assumed to be initially in an input file (infile). The end of the stream is signalled by an End of Stream Token.
The elements $<x_k, 1<k<m>$ are read and their indices set according to their order in the file. Many tokens can therefore be read simultaneously and each is then directed to the subgraph where it is required. Othere activations of the program graph that do not require any $<x_k, 1<k<m>$ can proceed at the same time.

* WRITE(outfile,X) — The writing of stream X cannot be initiated until all the elements $<x_k, 1<k<n>$ are produced (and End of Stream Token EST is defined). The rate at which the different elements are produced is immaterial: the 'WRITE' routine takes care of the sequencing, and the stream elements are output in their indices' order.

Then the following program segment, which reads characters from the host, one at a time, and then outputs the whole text as an ordered stream, illustrates how the sequencing problem is partially hidden from the programmer :

```
PROGRAM  IOTEXT (infile : input, outfile : output);
      Iteration charstream (text : stream of char);
           decl   A,  newA, new text : stream of char;
           Begin
               repeat
                   read(<head(A)>,infile);
                   newA := TAIL(A);
                   A := newA;
                   new text := consl[text, head(A)];
                   text := new text;
               until  TAIL(A) =  < >
           end;
Begin
    WRITE (outfile, charstream(<>))
end.
```

```
  < >          denotes an empty stream,
<simple token>     creates a 1-element stream,
 Head (X)      gives the first element of stream X (with index = 0)
 Tail (X)      gives a stream constituted by all the elements of X, except the first.
 consl [X, simple token]  creates a new stream by adding the token, as a first
                          element, to stream X.
```

The example also shows that streams can be generated from iterations : the R/W stream operators are therefore logical candidates to be incorporated in loops.

Dave Bowen has already discussed possible ways of implementing streams and useful stream operators in dataflow. The implementation of Read/Write operators, with the semantics described above, would in essence use a similar scheme as John Glauert's, but here the equivalent to the boolean parameters would be naturally provided by the indices of the stream tokens to achieve the necessary synchronisation.

A High-Level Primitive for Dataflow Systems Programming (1)

AJC4 2.5.79

1. Introduction

These notes describe the current state of an attempt to provide the dataflow machine with a suitable language for systems programming.

Until now a major concern has been compromising abstraction and feasibility. We do not desire such a general mechanism practical implementation would be infeasible, nor such a simple construct that the system programmer would have to be aware of unnecessary details.

This problem has been considered in at least two important recent papers: Hoare's Communicating Sequential Processes in CACM 21,8(Aug.78) and Brinch Hansen's Distributed Processes in CACM 21,11(Nov.78). Both deal with the parallel composition of sequential processes and both make use of Dijkstra's guarded command concept.

An implementation description of any of these proposals should give some insight into non-deterministic programming, some indication of its suitability for a dataflow environment, and something to compare with the alternative approaches already described elsewhere (such as Arvind and Gostelow's Dataflow Managers).

Brinch Hansen's language was chosen to start with and an implementation is described which preserves every characteristic of the original design. However, due to the sequential nature of the minor components of the language, some unnatural (from a dataflow viewpoint) features such as a program counter and a conventional store had somehow to be provided.

The discussion of the problems encountered during this process, and of the solutions adopted in each case, constitutes the main objective of these notes. Section 2 describes the implemented language, Section 3 the implementation model, while Section 4 gives some conclusions and an indication of further work to be carried out.

2. Distributed Processes

The language described in this section has been proposed by Brinch Hansen for real-time applications controlled by microcomputer networks with distributed storage. A complete description of the language and many example programs can be found in the above-mentioned paper.

A concurrent program defines only a fixed number of sequential processes that are to be started and executed simultaneously, and which exist forever. No other declarations are allowed within it.

A sequential process can declare own variables, common procedures and an initial statement. The initial statement may contain references and assignments to the process's own variables and calls to any of the program's common procedures. When a process is started, control is transferred to its initial statement.

```
process name
own variables
common procedures
initial statement
```

Variables can be declared of integer, boolean or character type, in scalar form or structured as a finite set, sequence or array of fixed dimension.

A common procedure can define input and output parameters, local variables and a statement. Input parameters are implicitly passed by value and output parameters by reference. The statement may contain references and assignments to the procedure parameters and local variables and to the enclosing process own variables. It may also contain calls to common procedures defined in the same or in another process. Recursion is not considered.

```
proc name (input param # output param)
local variables
statement
```

A statement is a sequence of zero or more of assignment, procedure call, guarded command, guarded region, enumeration or empty.

An assignment has the conventional syntax and semantics:

```
variable := expression.
```

A procedure call within a process P is of the form

```
call Q.R (expressions, variables)
```

where R is the called procedure name and Q is the name of the process where it is defined. Before the statement defined in R is executed the expression values of the call are assigned to the input parameters. When the statement defined in R is finished the output parameter values are assigned to the variables of the call. Also, R is considered an indivisible operation within process P, which will not execute any other statement until R is finished.

A guarded command arbitrarily selects for execution one among several guarded statements whose guards happen to succeed. One of the two available forms is

if B1:S1 | B2:S2 | ... end

which, if some of the guards B1, B2, ... succeed, selects and executes one of their corresponding statements or, if all guards fail, forces a program exception. The other form is

do B1:S1 | B2:S2 | ... end

which, while some of the guards B1, B2, ... succeed, selects and executes one of their corresponding statements. Control is given up as soon as none of the guards succeeds.

A guarded region waits until the success of some guards enables an arbitrary choice among several guarded statements to take place. One of the two available forms is

when B1:S1 | B2:S2 | ... end

which awaits until some of the guards B1, B2, ... succeed and then selects and executes one of their corresponding statements. The other form is

cycle B1:S1 | B2:S2 | ... end

which corresponds to an endless repetition of a when form.

An enumeration produces all the elements in a set or array and has the form

for x in y: S end

which executes the statement S for each element x in the data structure y. Array elements and scalar variables can be referred and assigned to within statement S, but set elements can only be referred to.

An empty statement corresponds to no operation and has the form

skip.

The use of the semicolon as a separator is optional everywhere.

At most one statement within a process can be in execution at any one time. When this statement terminates or is delayed within a guarded region the process becomes idle. A process is not regarded as idle while awaiting the completion of an external common procedure call it has issued. A process remains idle until at least one of its statements becomes enabled for execution. Statements that can be enabled for execution are fresh external calls of any of

the process' common procedures and statements which have been delayed previously within guarded regions but which have now found at least one succeeding guard. If more than one statement become enabled for execution an arbitrary choice among those will indicate the one to be executed.

3. Implementation Description

3.1 General Remarks

An implementation is proposed for each of the elements of the language. Although an effort has been made to reduce the unnatural (from a data-flow viewpoint) features of some constructs, it is not always possible to design an elegant solution which still meets every requirement. The constructs introduced thus demonstrate bad data-flow practice, but this has to be accepted as the price to be paid for the exercise. Even less attention has been paid to performance, as it seems unreasonable to optimize provisional code.
The main departures from normal data-flow lines are the introduction of a "store" and the sequencing of statement execution, as discussed below.

During the process, seven symmetrical matching store functions were assumed available, six of which deal with two-input nodes. Three sub-functions are concerned with the attitude of an incoming token which does not find its partner in the matching store: await (A - implying the storage of the token until the partner arrives), circulate (C - implying a loop round the ring and a later attempt) and empty (E - implying an immediate match with an empty token and normal continuation). Two sub-functions deal with the attitude of a token towards its partner: duplicate (D - implying that a copy of the partner is to be kept in the matching store after the match) and fetch (F - implying the removal of the partner from the matching store after the match). The combination of these sub-functions originated the six mentioned functions: AD, AF, CD, CF, ED, EF. For consistency reasons, only one of the tokens directed to a two-input operator can bear any of these functions, the other has always to bear AF. In the figures throughout these notes any matching store function other than AF or bypass has been explicitly indicated in parenthesis along the corresponding arc.

Store (Fig.1) exists globally (at process level) and locally (at procedure level).

Global store holds tokens bearing null identifiers in special 'set label and destination' (SLD) nodes which allow access from several distinct contexts to be carried out without the troubles outlined in AJC3. However, that kind of node exceeds the generality required by this particular problem as the iteration level field is never used.

Local store holds tokens in simple 'set destination' nodes, as the context is always preserved in these cases.

Reading (Fig.2) and writing (Fig.3) operations on both global and local store appear in the graphs as macro nodes. The mode of access when reading is normally explicitly annotated in the boxes. As an option the write macro node issues a completion signal when the written token has reached the target storage node.

3.2 Programs

A program is implemented (Fig.4) as a number of independent graphs, each of them corresponding to a process. The only action explicitly performed by a program is the duplication and delivery of its external activating trigger to each of its processes.

3.3 Processes

A process is implemented (Fig.5) as a number of storage nodes and independent graphs. There will be one storage node for the process's own statement trigger and one for each of its own variables. The independent graphs correspond to the common procedures and the initial statement defined by the process. The trigger delivered to the process by the program is regarded as its own statement trigger. This is always held by the process statement currently being executed, being kept in its corresponding storage node during any temporary idleness of the process. On process activation it is immediately transferred to the process initial statement as this is then implicitly enabled for execution.

3.4 Procedures

A procedure graph (Fig.6) has to be entered twice in the manner already described in AJC1: On Providing Common Graphs. The preliminary procedure subgraph is supposed to generate a unique identifier, but the problems associated with the exhaustion of these values have not been considered here. The top part of a procedure main graph is responsible for initializing the input parameter storage nodes with the values provided by the expressions of the call and for triggering the procedure statement. To accomplish the latter it either joins the other activities enabled for execution in their fight for the process own statement trigger, if it is an external call, or generates a trigger straight away, otherwise. When this trigger is eventually produced it is passed on to activate the procedure statement. The last part of a procedure graph is entered when the statement termination signal is received and its function is to distribute the computed results to the variables of the call, to clear up the storage nodes that have been used by the local variables and input parameters, as well as to

signal the idleness of the process if it was an external call.

3.5 Statements

A statement (Fig.7) is activated by means of a trigger which is owned by the corresponding process and passed to the enabled statement arbitrarily selected for execution. The statement graph retains that trigger until it either terminates or is delayed within a guarded region. In the first case the trigger is passed on as a termination signal and used to activate the next statement, if any. In the second case the trigger is sent back to its global storage node to signal the temporary idleness of the process. The action taken when the termination signal is output by the last of a sequence of statements varies from one case to another and is indicated in the relevant entries and figures.

3.5.1 Assignments

When an assignment (Fig.8) is activated the right-hand side expression is evaluated and, as this is the only statement in execution within the process at this time, there is no danger of any of the involved variable values being altered during this computation. When a result is eventually produced, it is written to the left-hand side variable storage node and the token arrival is ascertained before the termination signal is passed on.

3.5.2 Procedure Calls

A procedure call (Fig.9) initiates by requesting a unique identifier from the called procedure and by evaluating its input expressions. Properly labelled input expression results, internal/external call indicator and result delivery addresses are then supplied to the called procedure. Any further action is postponed until the results are eventually received. A procedure call issues no "delayed" signal and will only yield its termination signal after the updating of the output variables is completed.

3.5.3 Guarded Commands and Regions

Guarded commands and regions are all implemented as different ways of handling a guarded statement set. A guarded statement set, whose internal operation is described in detail below, is awoken by a boolean trigger and eventually responds with a boolean status signal. The boolean trigger indicates whether a statement is actually supposed to be executed ('control on') or whether only an inspection of the guards is required ('control off'). The boolean status signal has to be interpreted in accordance with the value of the trigger. If the trigger was 'on', the boolean status signal

will indicate either that some guards succeeded and one of their corresponding statements was executed ('success') or that all guards failed and no statement was executed ('failure'). If the trigger was 'off', the boolean status signal will indicate either that some guards succeeded ('success') or that all guards failed ('failure').

Both guarded commands trigger their guarded statement sets in a 'control on' condition; their differences residing in the way they treat the boolean status signal returned. An if statement (Fig.10) will yield its termination signal on receiving a success status or stop the program otherwise. A do statement (Fig.11) will yield its termination signal on receiving a failure status or re-trigger the command otherwise.

The operation of a guarded region (Fig.12) is more subtle, since, if it is unable to perform any action at the time it is triggered, it relinquishes control (actually by sending the trigger back to its global storage node). When control is given up, the region becomes passive in the sense that it will not start executing any statement. However it continues to evaluate its guards until at least one of them eventually succeeds as a consequence of some other action now being performed by the process. This kind of busy waiting certainly implies a waste of processing power and could be, if not completely avoided, at least largely alleviated. However, at this stage, efficiency was not a major concern, nor was Brinch Hansen's proposal being taken as a sufficiently definitive option to deserve the optimization treatment.

As soon as a guarded region detects any succeeding guards it starts competing for control once again. After this is accomplished the guards are re-evaluated to make sure that at least one is still succeeding and only after that is the corresponding selected statement executed.

To implement this, both guarded region forms initially trigger their guarded statement sets in a 'control on' condition. If a success status is returned the statement either terminates (when statement) or is re-triggered (cycle statement). If a failure status is returned, the region is delayed by returning the process own trigger to its storage node and the guarded statement set is reactivated but in 'control off' mode. This re-triggering in 'control off' mode, which prevents the execution of any statement and preserves the passive condition of the region, is repeated until a success status is output by the guarded command set. This indicates that at least one guard has succeeded and, in this case, the region starts competing to acquire control once again. Upon the receipt of the process own statement trigger the whole operation is repeated.

A guarded statement set (Fig.13) implements the nondeterministic and (very likely) non-reproducible selection of one among the succeeding guarded statements through a local trigger storage node,

which is initialized when the set is activated. Succeeding guarded statements compete for this trigger and, eventually, all return boolean status signals reflecting their achievments. These have a meaning analogous to that of the boolean status signal for the set, which is actually built by combining the individual ones.

A guarded statement (Fig.14) firstly evaluates its guard. The result is immediately returned as the status signal if in 'control off' mode or if the guard fails. If in 'control on' mode and the guard succeeds, the guarded statement requests the local trigger. If an "empty" token is received, indicating that the trigger has already been acquired by another statement with a succeeding guard, the statement fails. Otherwise the statement is executed and a success signal output afterwards.

3.5.5 Enumerations

An enumeration repeatedly executes a statement for each of the elements of a data structure. This is implemented using the activation trigger to read the first element of the data structure into a local storage node. If this has not an 'end of stream' value, the statement to be repeated is activated. The termination signal output by the statement is used to initiate the next cycle and to cause the value of an accessed array element to be updated.

An enumeration terminates when an end-of-stream token is read, in which case the termination signal is output.

4. Conclusions

This exercise has served to point out some of the problems one should expect to encounter when modelling a conventional machine on a data-flow architecture. Both the storage nodes and the sequencing of statement execution clearly contrast with what a dataflow design should look like. More definite conclusions could certainly be drawn if the model were actually implemented so that some quantitative measurement of the overhead introduced by the unnatural features of the implementation could be obtained. However, this seems unlikely even when a new emulator is available.

It also seems worth discussing the use of the 'match with an empty' facility to implement non-determinism in the machine, the provision for automatic token circulation when a match fails, and other alternatives for implementing a conventional store.

In a further set of notes, an attempt shall be made towards making this notation more data-flow like, without losing the level of abstraction exhibitted by the current constructs. A parallel

with Arvind and Gostelow's proposal is also expected to be established, but any other suggestions will be welcome.

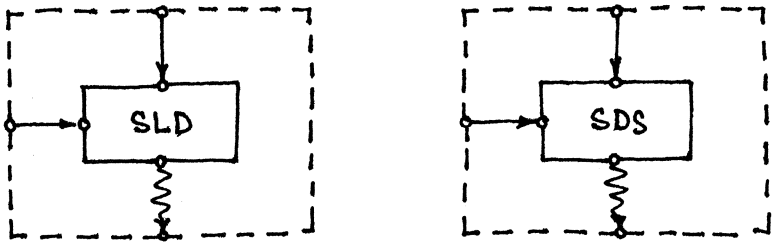


Fig. 1 Global and local storage nodes.

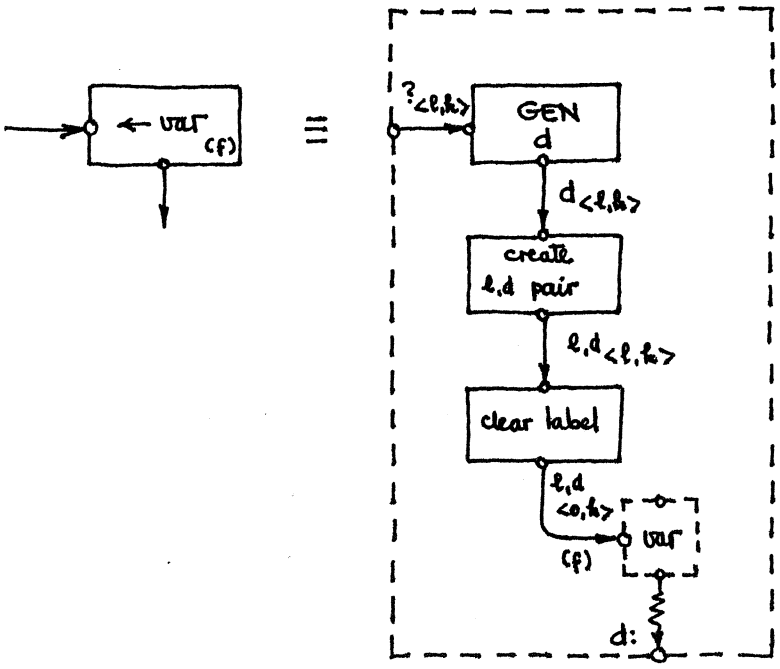


Fig. 2 Global read macronode.

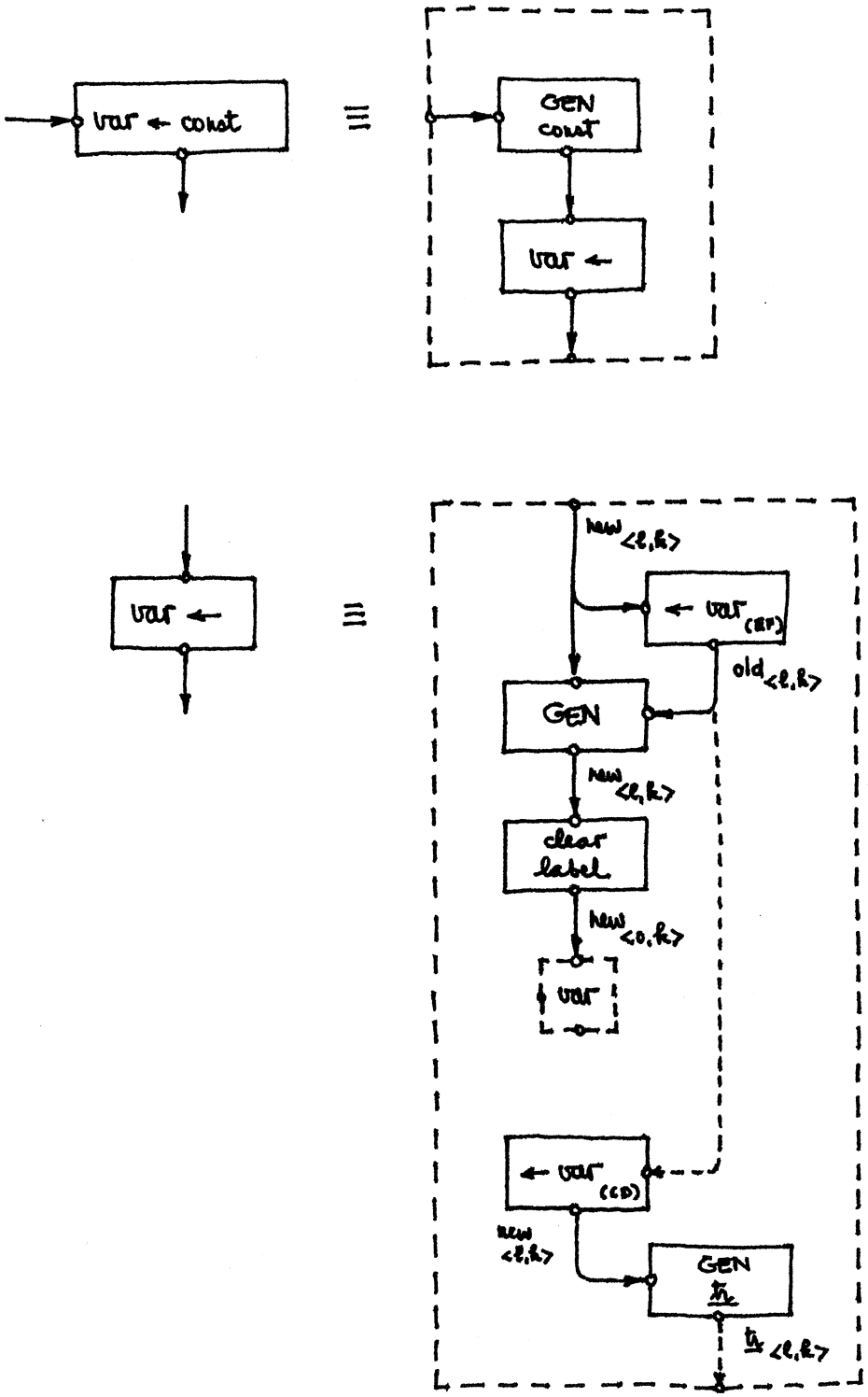


Fig. 3 Global write macro nodes.

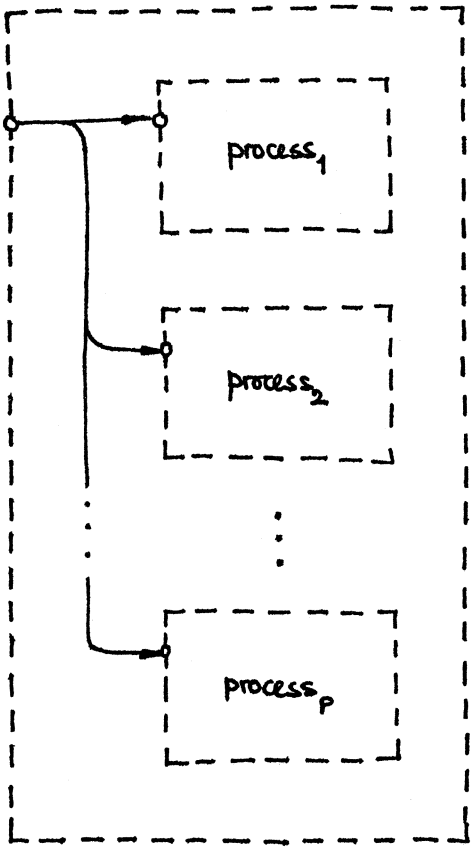


Fig. 4 A program.

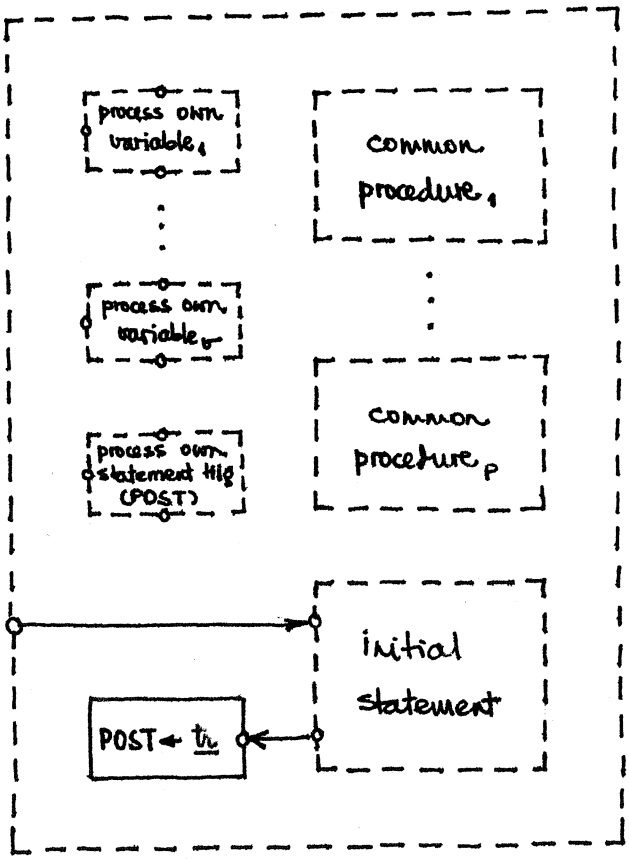


Fig. 5 A process

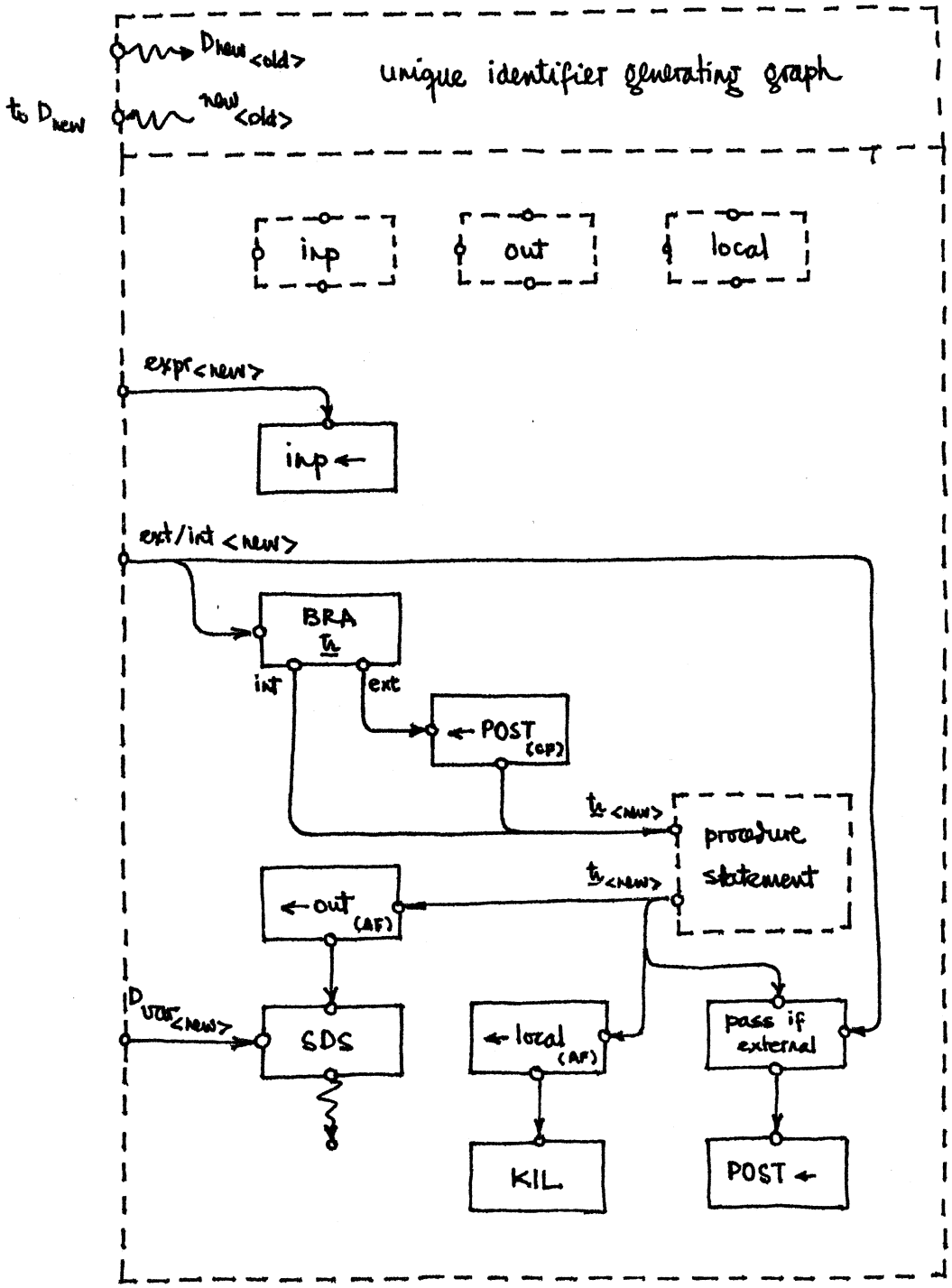


Fig.6 A procedure graph

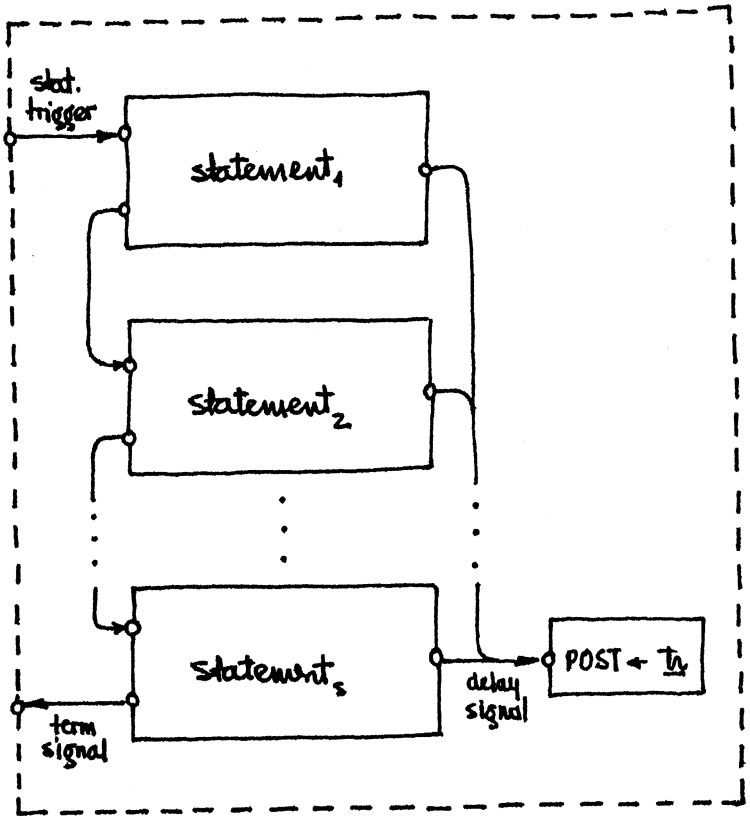


Fig.7 A general statement.

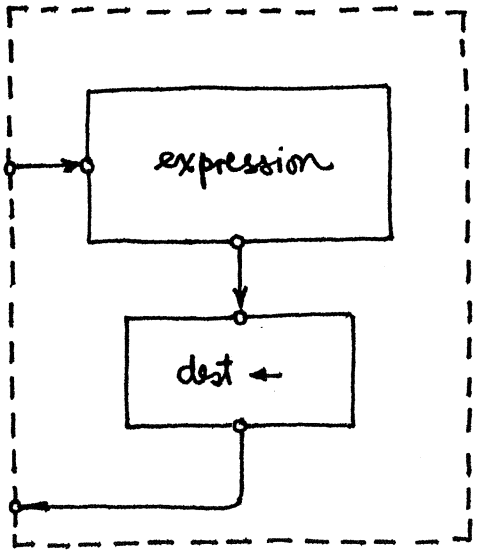


Fig.8 A dest ← expression assignment.

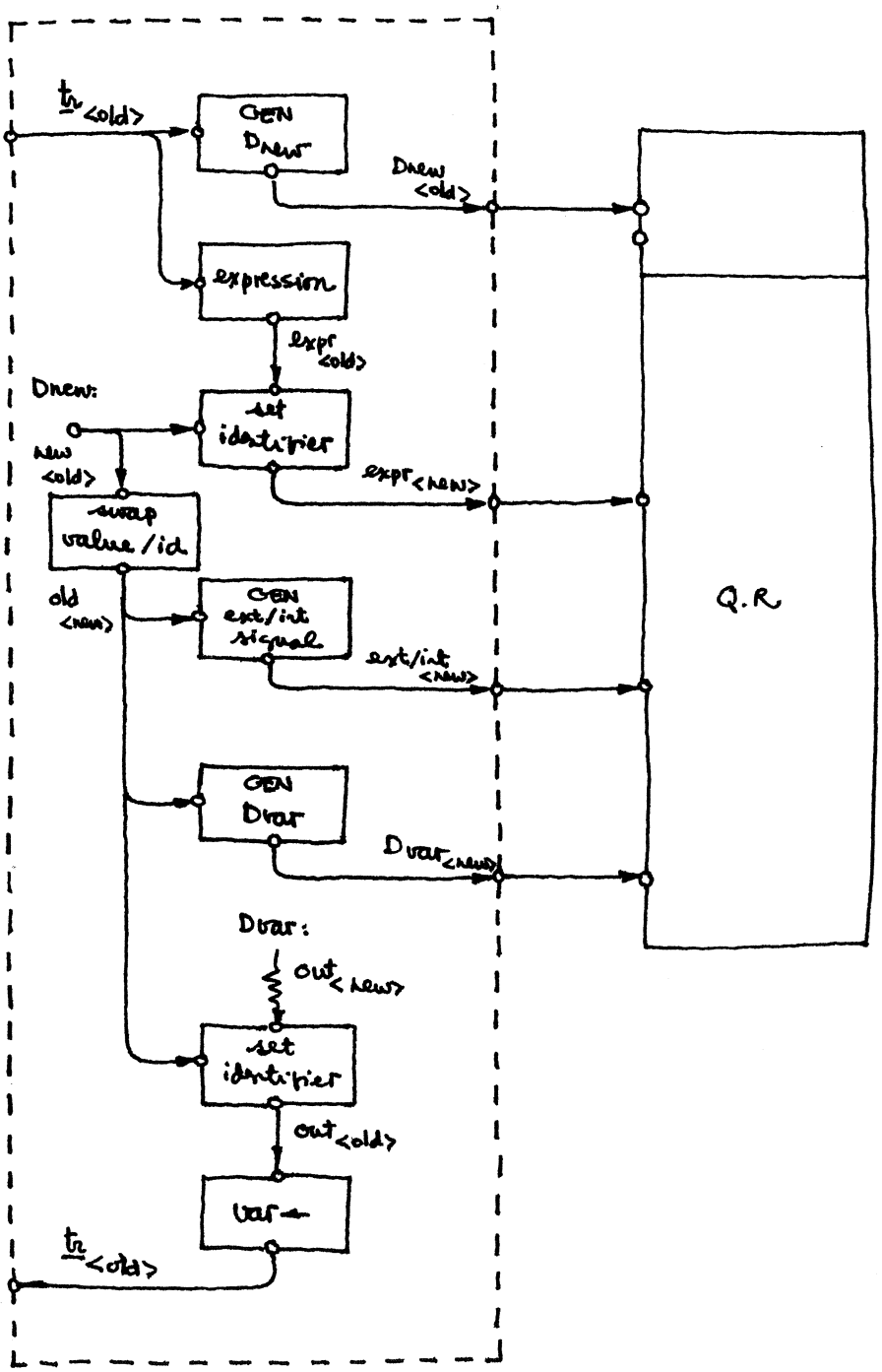


Fig. 9 A Q.R (expr, var) procedure call.

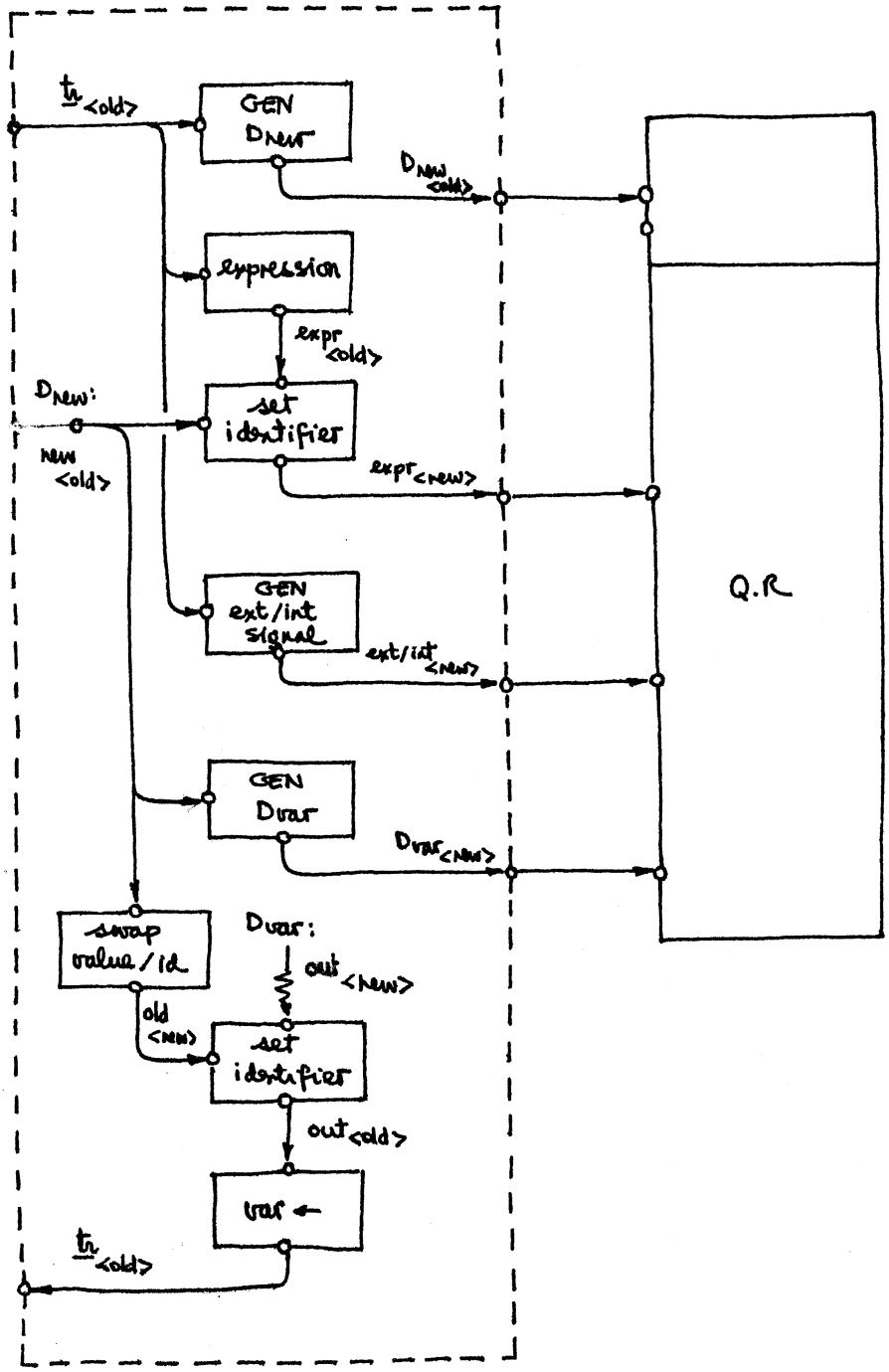


Fig. 9 A Q.R (expr, var) procedure call

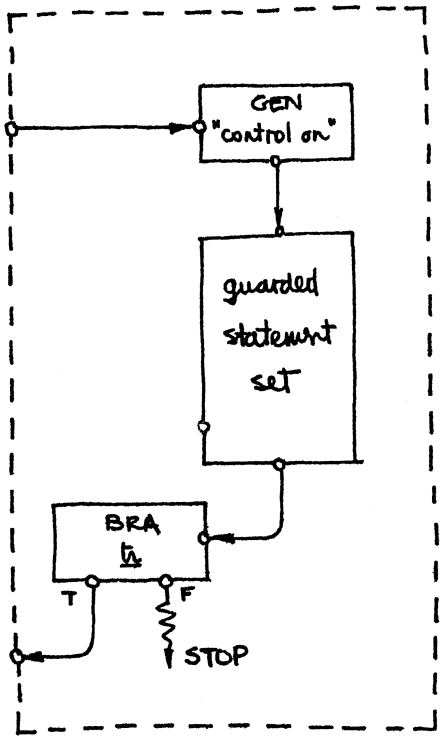


FIG.10 The IF statement

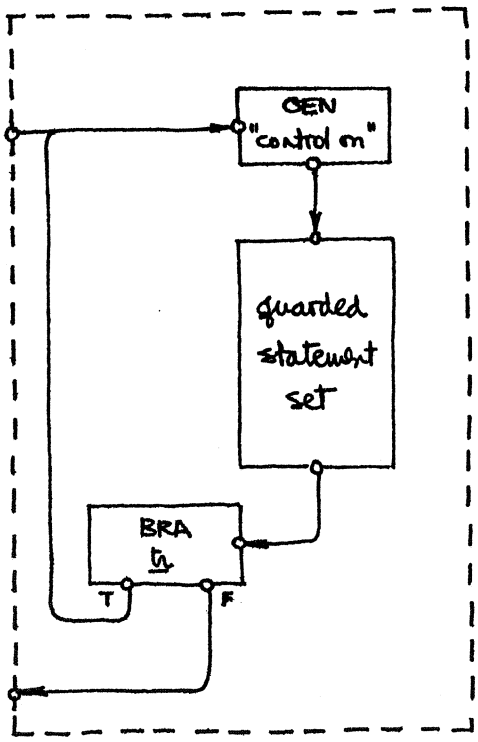


Fig.11 The DO statement

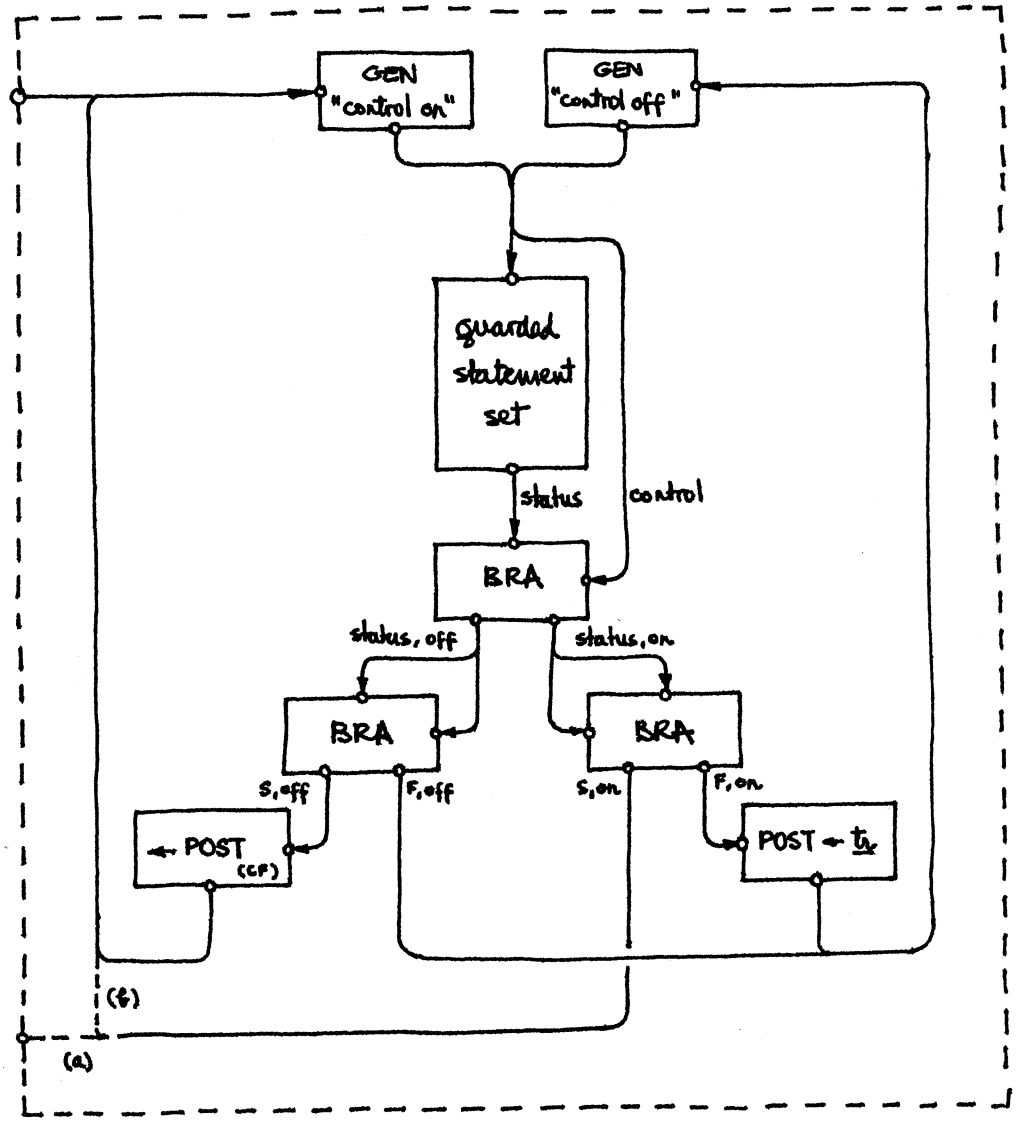


Fig. 12 A when (α) or cycle (β) statement.

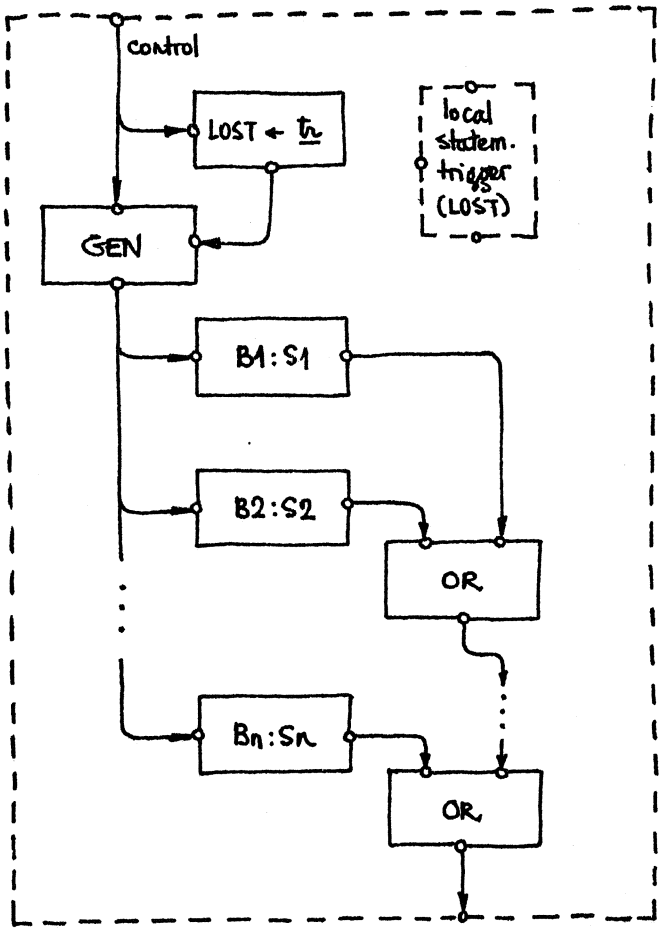


Fig. 13 A guarded statement set

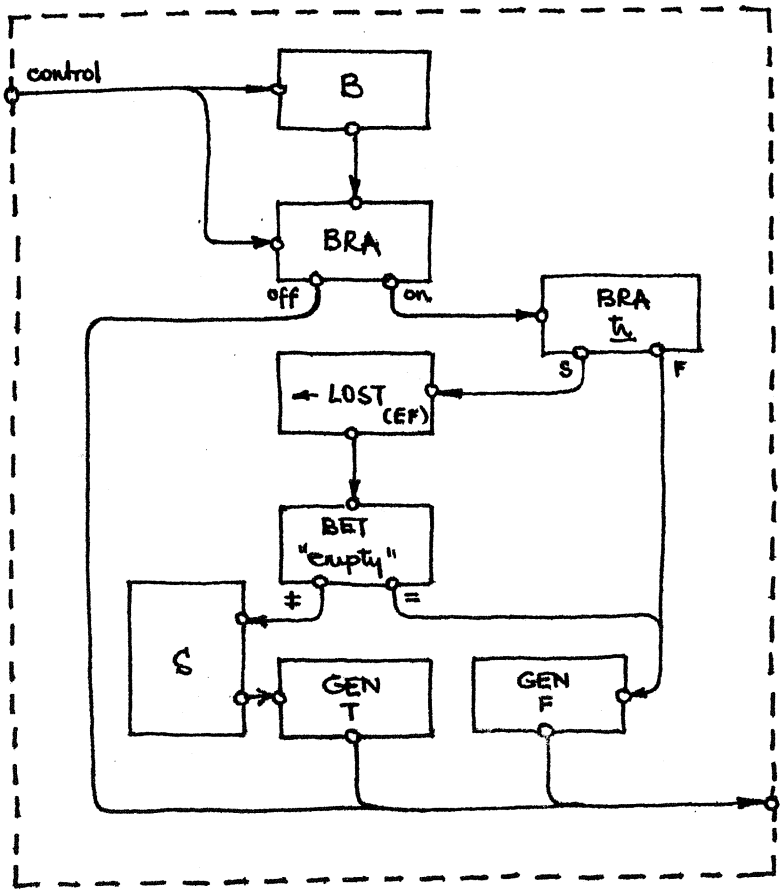


Fig. 14 A B:S guarded statement

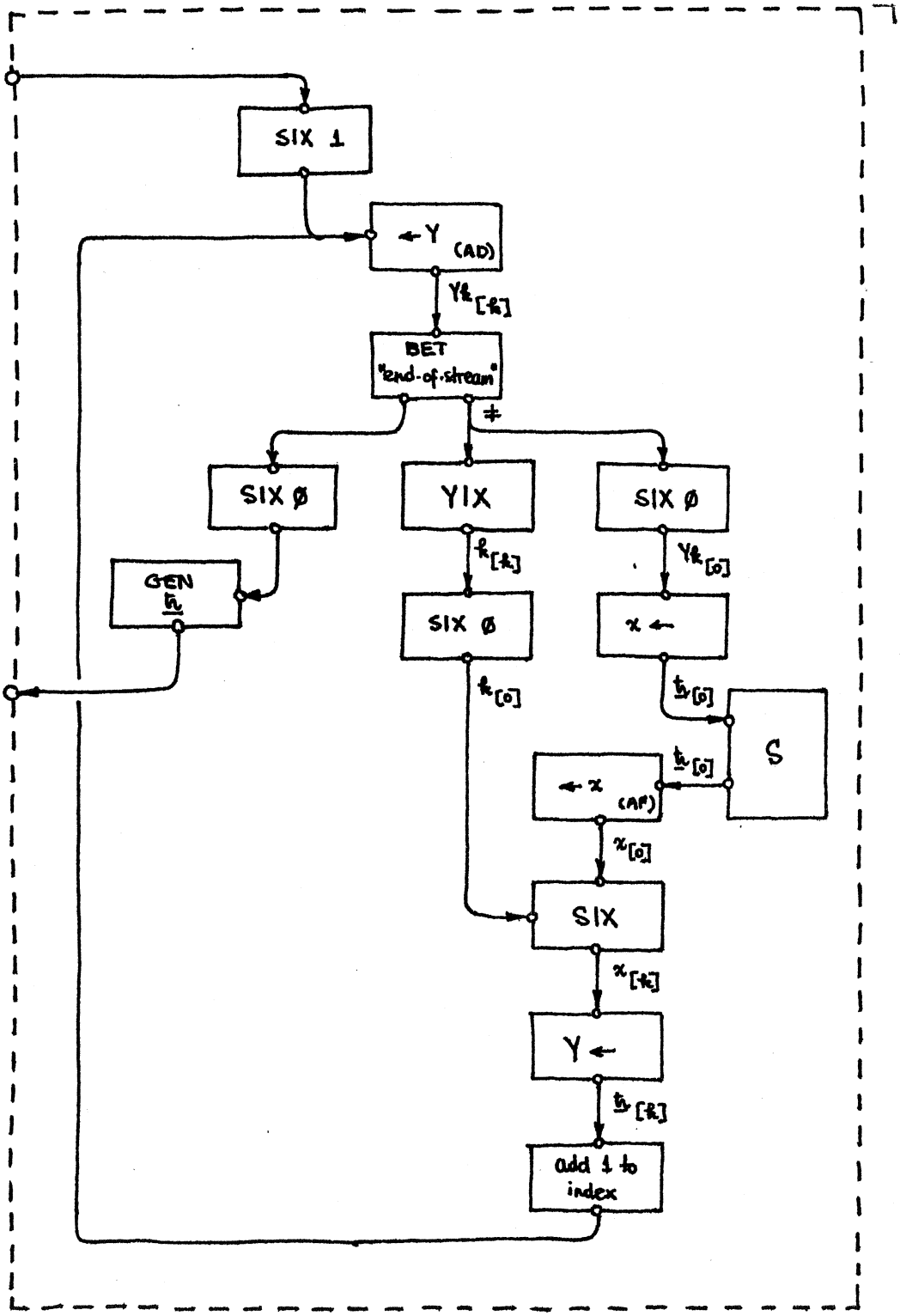


Fig.15 A for x in Y : S end enumeration (Y is an array).

GENERALIZED STREAMS IN THE MATCHING STORE

In DB3, "Generalized Streams on the Data Flow Computer", it was shown that multi-dimensional streams could be implemented on the Manchester Data Flow Machine with two new nodes (See Fig. 1).

Here STORE stores away an entire stream in memory and returns a pointer to it, while FETCH provides the inverse operation. The definition of these new nodes presupposed the addition of some conventional store to the machine.

However, with only a slight modification, these nodes can be redefined as macro nodes which do not require a new store. Instead the data can be held in the Matching Store on a "Set Label and Destination" (SLD) node. Pointers are replaced by <label.destination> pairs.

The STORE macro node is shown in Fig. 2. The stream is held on the node addressed by A: . A node has been invented to combine a destination type with its label to form a <label.destination> pair. The number following each destination literal is the input point.

In the expansion of the FETCH macro (Fig. 3), the <label.dest> of the storage node is circulated around a loop, incrementing the index, until the end-of-stream token is found.

It was suggested (DB3) that the FETCH node could be given a subscripting capability in order to access individual elements of a stream. It seems sensible to give this operation a new name, SELECT, and it can be similarly redefined as a macro node (Fig's 4&5).

Garbage Collection

A problem with these definitions is that streams, once stored, are never destroyed. A method of disposing of unwanted streams is required. In order to be able to decide whether or not a particular stored stream is still wanted, it is necessary to keep a count of how many <label.destination> pairs referring to it currently exist in the machine. Suppose that this count is held on a new SLD node which is situated immediately after the SLD node holding the stream, in the Instruction Store. Then clearly this count must be initialized by the Store macro (Fig. 6).

Assume for the moment that we are dealing only with one dimensional streams or records (the internal representation of records is the same as that of streams, except that the elements do

not in general all have the same type). The reference count is incremented, using the INC macro (Fig. 9), whenever the <label.dest> is duplicated. It is decremented, using the DEC macro (Fig. 10), whenever a copy of the <label.dest> is destroyed. The DEC macro gives the <label.dest> a unique identifier and sends it to the garbage collection routine, which decrements the reference count, and, if it becomes zero, destroys the stream (Fig. 11).

Note that FETCH and SELECT destroy a <label.dest> and hence must decrement the reference count (Fig's 7&8). Note also that FETCH outputs a stream in index order, so that the arrival of the end-of-stream token ensures that the whole stream has been fetched from the SLD node.

Now let us consider multi-dimensional structures. If a stream of <label.dest> tokens is to be duplicated, then each of their reference counts should be incremented. In order to do this, the index has to be removed, and so the INC macro is embedded in a "for each" .. "return all" loop to give the STREAM INC macro (Fig. 12).

When a stored stream of streams is destroyed by the garbage collection routine, its elements, which are stored streams, must have their reference counts decremented. Hence the garbage collection routine calls itself via the DEC macro (Fig. 11).

When a stored stream of streams is FETCHed, the reference counts of each of its elements must be incremented, since a new copy of each <label.dest> is produced (Fig. 13). All the incrementations must be completed before the count for the stream of streams is decremented (otherwise an element could be destroyed when it shouldn't be). In SELECT just one increment is needed (Fig. 14).

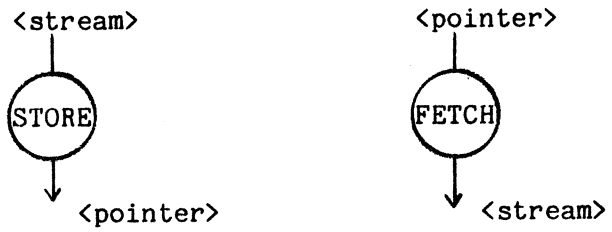


Fig. 1 STORE and FETCH

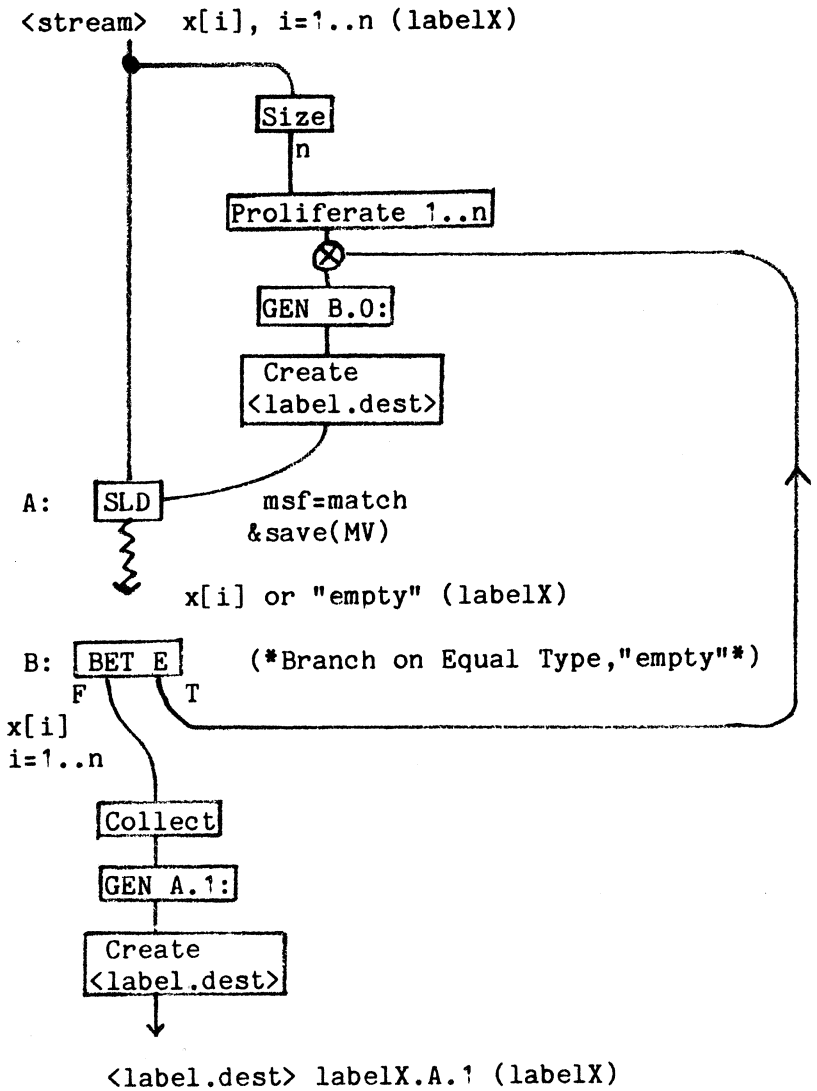


Fig. 2 The STORE macro

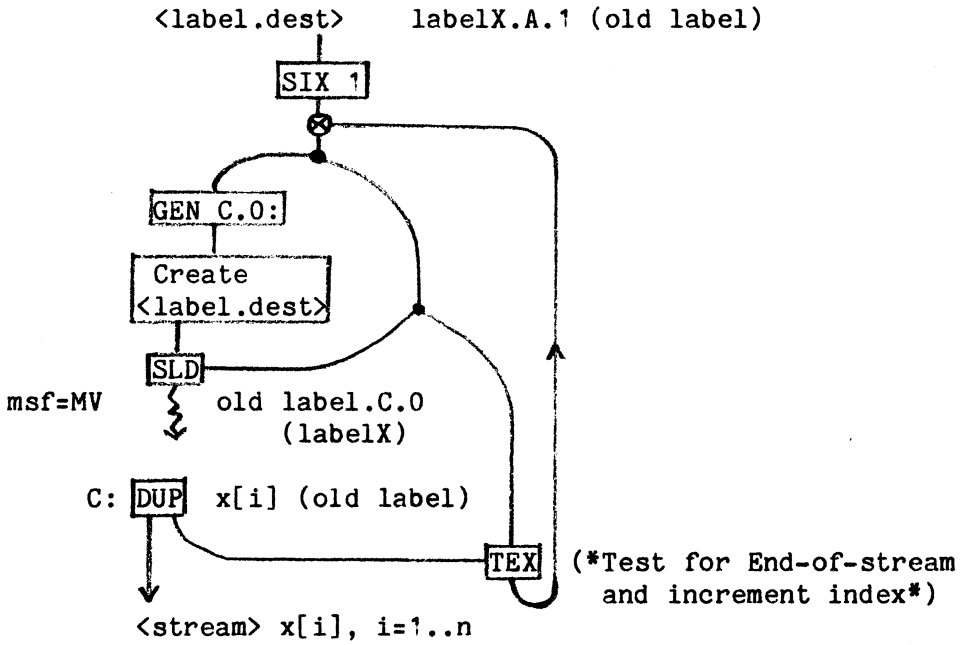


Fig. 3 The FETCH macro

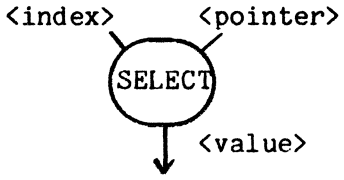


Fig. 4 SELECT

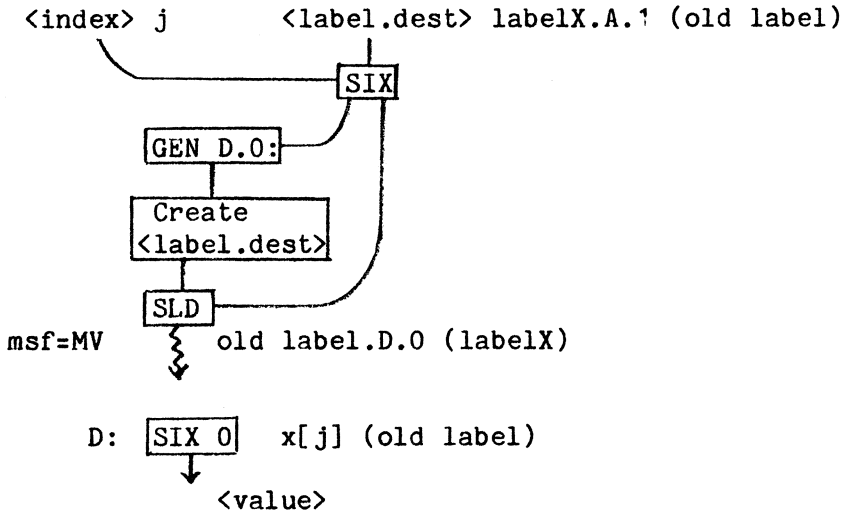


Fig. 5 The SELECT macro

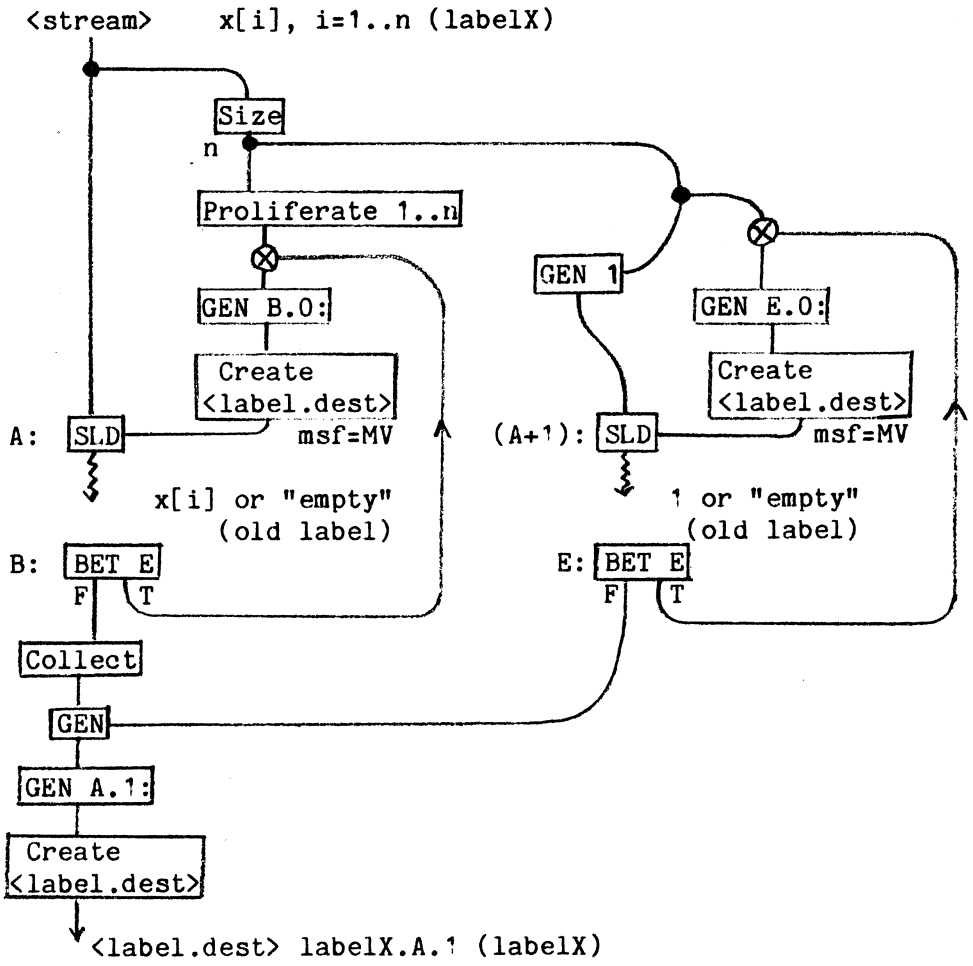


Fig.6 STORE with count initialization

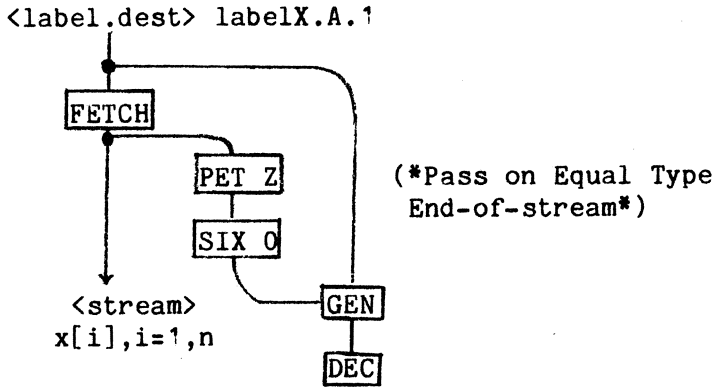


Fig.7 FETCH with count decrement

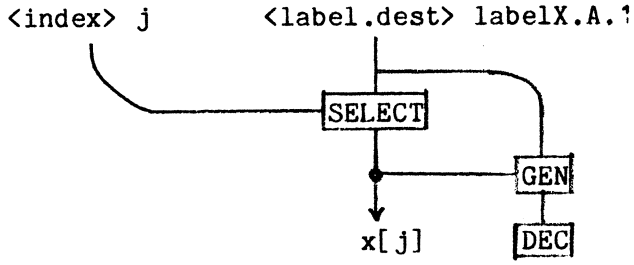


Fig.8 SELECT with count decrement

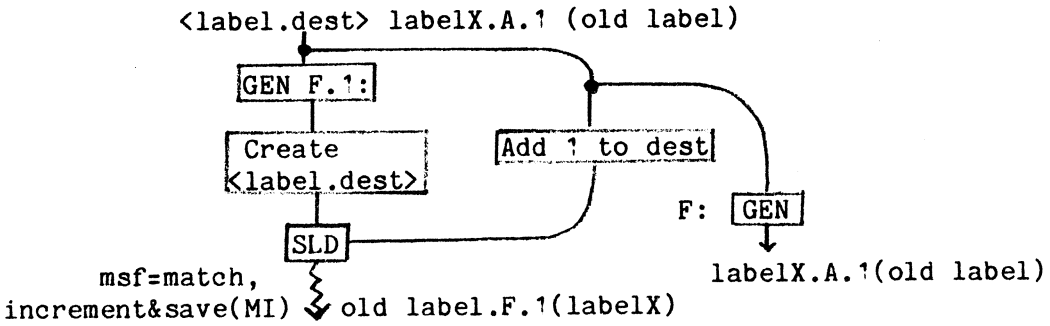


Fig.9 The INC macro

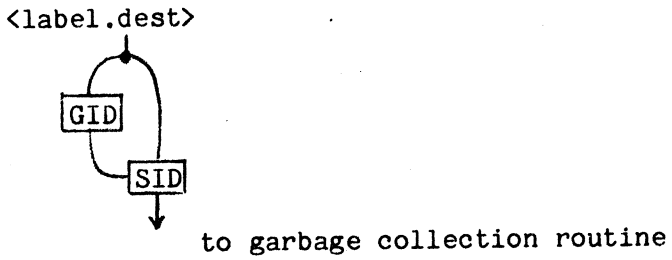


Fig.10 The DEC macro

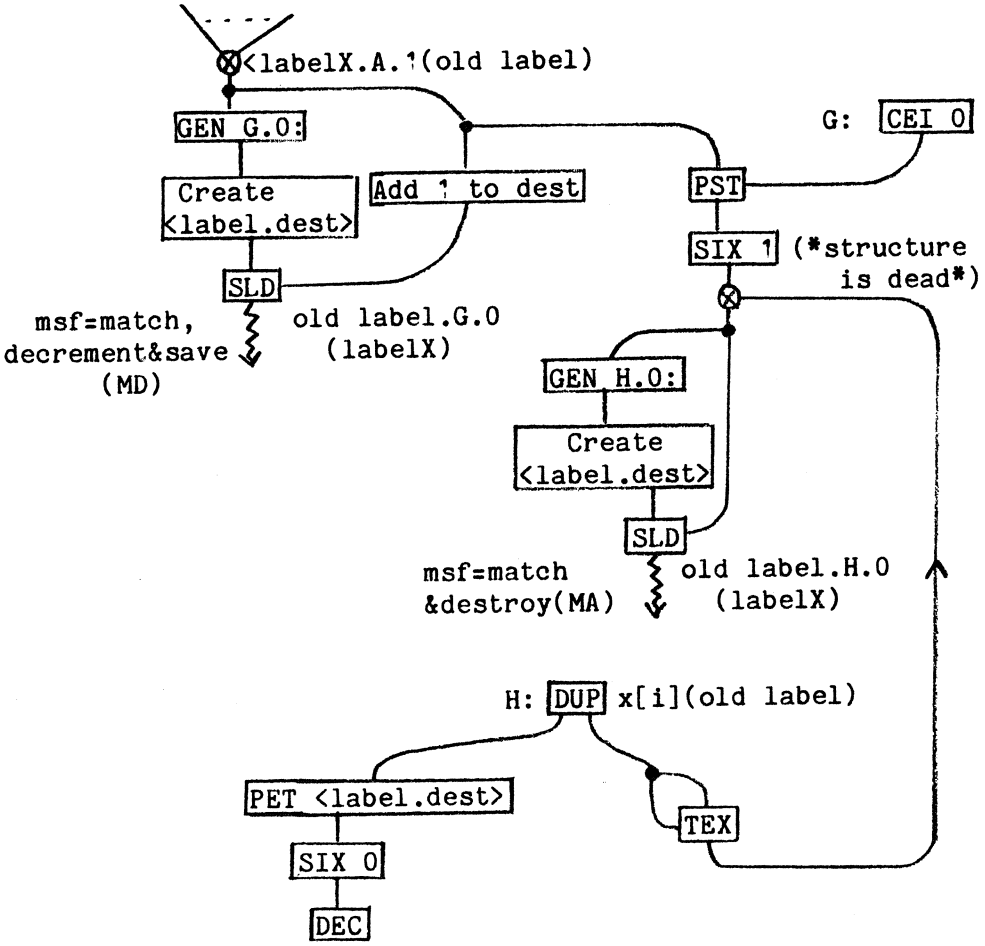


Fig.11 The garbage collection routine

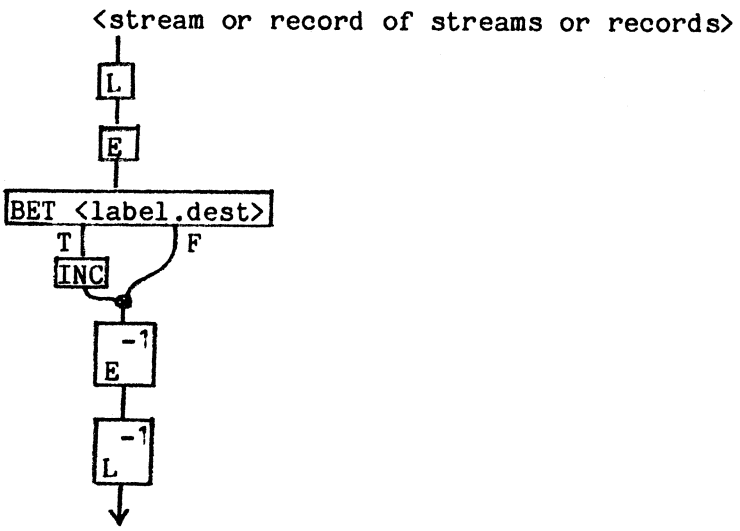


Fig.12 The STREAM INC macro

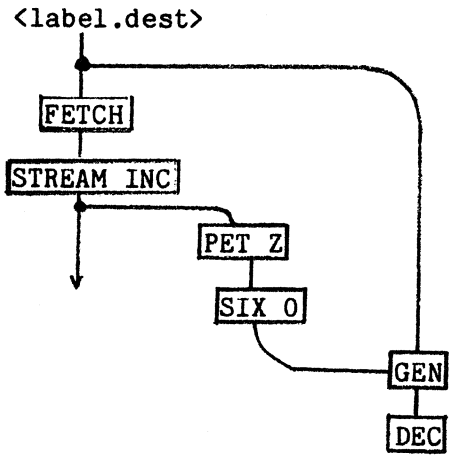


Fig.13 The FETCH macro, multi-dimensional case

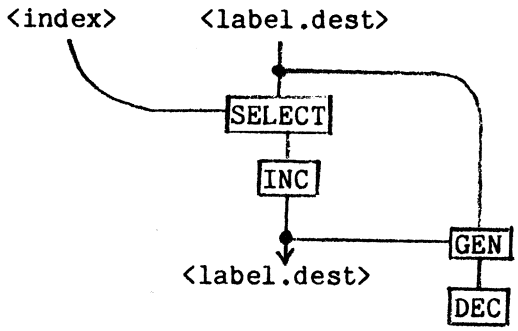


Fig.14 The SELECT macro, multi-dimensional case

GKE2

Data-Flow Machine 4 (2?)

Introduction

I have recently come to the conclusion (belatedly) that it is a very good idea to put thoughts down on paper at the earliest possible time, and circulate them for the possible use of others.

My current interest is in the decentralised control of real-time systems and this has led me to an architecture quite different to the one you have come to know and love (The Manchester Machine TMM). On the following few pages I have attempted to briefly describe DFM4 (three went on the scrap heap) and the way it works.

DFM4 (Fig. 1) consists of a number of communicating modules; for the application area this number could be large (hundreds). Communication paths between modules would most probably be bit-serial, or at best byte-parallel, and the modules themselves may be of limited computational capability. It is important therefore not to overload communication paths and modules with excessive token tagging schemes and their supporting primitives. This does not severely limit us however as sub-graph sharing and even multiple recursion is still possible.

The only tags in this architecture are those associated with token destination (node description location), mode (type and size) and only when sharing a sub-graph, a copy number (GKE1).

Given the above, the main requirement of the hardware is to maintain the strict queueing of tokens on arcs for any given sub-graph invocation. This is not as difficult as it may first appear as we know quite a lot about the pattern of arrival of tokens at any given node type.

How DFM4 Works

The structure of the modules (Fig. 2) attempts to maximise flexibility in the initial experimental configuration; it also allows me to assess the practicality of executing data-flow graphs on conventional systems.

Each module is comprised of several sub-modules and, in the initial implementation, they are as follows:

PE (processing element) A shared controller and execution unit,

NS (node description store) A conventional store containing descriptions of graph nodes,

IQ (input queue) A hardware FIFO buffer through which the module receives tokens from other modules,

LQ (local queue) A software queue which contains tokens generated by the module which are destined for the same module (similar to the agenda queue of Davis's DDM1),

OQ (output queue) A hardware FIFO buffer through which the module transmits tokens to other modules,

AQ (arc queue) A linked list structure containing tokens which are queued on one arc of two-input-arc nodes.

AQ (Fig. 2) is the equivalent of the matching store in TMM and as such deserves some comment. Each two-input node description has a link into AQ ,the first entry of which contains the following:

1) The arc-number on which tokens are currently queued,

2) A pointer to the first token on the arc,

3) A pointer to the last token,

4) If the module supports function invocations, a copy number and a pointer to a similar entry for any other active invocation.

Module operation is fairly simple and proceeds as follows:

PE scans LQ and IQ until a token is present.

The node description corresponding to the token which has just arrived is examined to see if it has one or two inputs.

If the node has one input, PE executes the function and writes any resulting tokens to OQ or LQ .

If the node has two inputs, AQ is accessed via the node link for a matching token.

If the match is successful, the function is executed; otherwise the token is linked to the end of the appropriate arc queue.

The PE then returns to scan LQ and IQ .

The detailed behaviour of a module may vary depending on its primary function ie. input-output, execution etc.

It is quite possible to configure modules which exploit concurrency in arc queueing, node execution and result transmission to a far greater degree than the structure above; a technique I use to identify this concurrency is to construct data-flow graphs which describe module operation.

Shared Sub-graphs

Some modules may not support shared sub-graphs e.g. simple controllers at the periphery of a control system. However for those modules which do support shared sub-graphs, the following mechanism is used to separate different sub-graph invocations. On entry and exit from a sub-graph a copy number is computed by the equations below:

entry	newcopy=oldcopy*maxoccurrence+occurrence
exit	newcopy=(oldcopy-occurrence) DIV maxoccurrence

It is important to note that the copy tag is only appended to tokens actually involved in a shared function invocation. The copy numbers correspond to arcs of an n-ary tree .

Don't-knows (?)

Although the token modes supported by DFM4 were described in GKE1 , I should perhaps expand on one of them ie. the mode ? .

The primary use of this mode is to communicate information about errors during the execution of a graph to the graph itself. In DFM4 errors fall into two classes:

1) Faults in execution or attempted execution of node functions e.g. function argument mode errors (inc. I/O), arithmetic errors, range errors (data windowing) etc. With this class of error a ? token can be safely propagated to succeeding nodes. ? tokens propagated in this manner retain the original reason for error and the destination at which that error occurred. The ? token can not be used as a control token on conditional path nodes (PIT, PIF, SWI).

2) Destination errors e.g. non-existent node description, inactive node input arc etc. Because no successor node exists for this class of error a reserved error node is defined in each module. A token of mode destination is sent to one input-arc of the node and any ? arriving at the error node is sent to that destination.

Other uses for the ? token in areas of partial pattern matching etc. will be described in later documents.

Input-output

Input and output nodes are reserved and are associated with particular devices. The actions of input-output nodes are as follows:

1) Input A response destination , which remains valid until another arrives, is sent to one arc; to the other is sent a token of any mode. Depending on the nature of the device associated with the input node, it will eventually respond with valid data or ? . If no response destination has been specified, a ? is sent to the module's error node.

2) Output A response destination, as for input, is sent to one arc and data to the other. The node responds with a copy of the original data or ? . If no response destination is specified then ? is sent to the modules error node only when the output action fails.

Storage-nodes

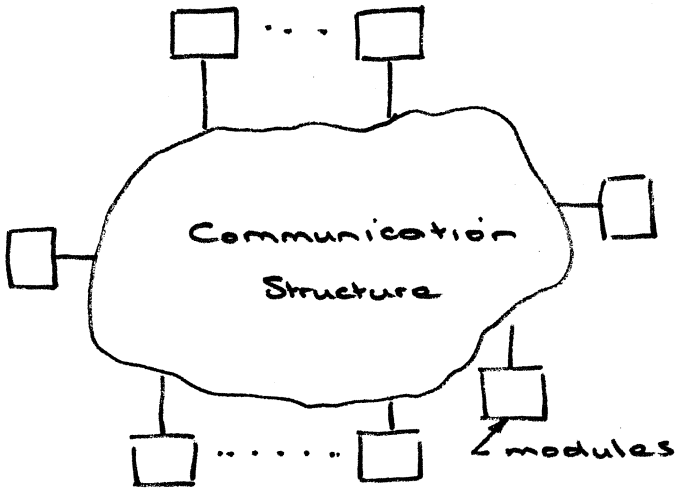
In adaptive controllers it is necessary, from time to time, to update "constants" in the difference equations which represent the digital compensator. While it is possible to retain information by circulating these "constants" it is, to say the least, not very efficient.

The approach I have taken is to provide a storage node. One input-arc of this node receives written tokens while the other when receiving any token causes a copy of the last written token to be transmitted. If no token has been written a ? is issued.

Because read and write operations are not synchronised, graphs using storage nodes may pass through periods of "fuzzy determinacy" when write actions occur.

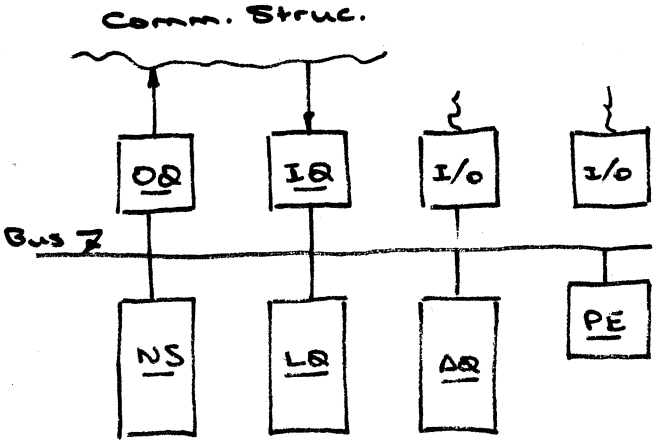
Conclusion

I hope this document has provided some idea of the operation of DFM4. Future documents will describe other features which are perhaps more relevant to the work I am doing; in particular the absence of time from the data-flow model and its effects when tokens "go missing" in the system.



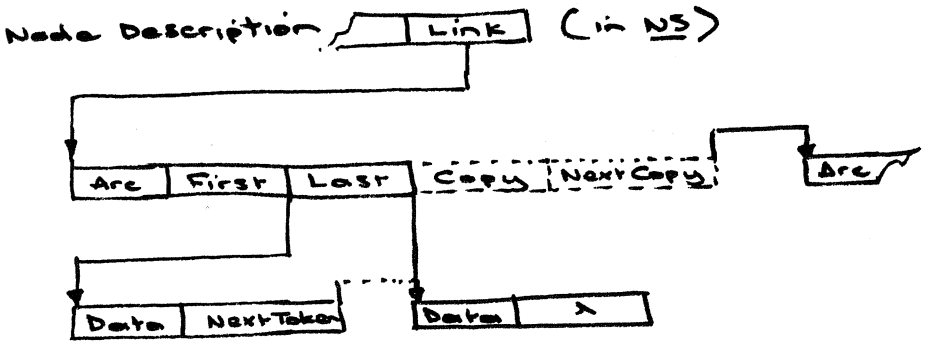
DFM4

Figure 1.



Module

Figure 2.



AQ

Figure 3.

Bibliography relating to data flow

Minsky :
"Computation : Finite and Infinite Machines" - Prentice Hall 1967.

Hopcroft & Ullman :
"Formal Languages and their Relation to Automata" - Addision Wesley 1969.

Ianov :
"The Logical Schemes of Algorithms" - Problems of Cybernetics 1, 1958.

Dijkstra :
"A Discipline of Programming"

Karp & Miller (a) :
"Properties of a Model for Parallel Computations : Determinacy, Termination, Queuing" - SIAM J. Appl. Math., Vol.14, 1966.

Adams :
"A Model for Parallel Computations" in "Parallel Processor Systems, Technologies and Applications" - Spartan 1970.

Karp & Miller (b) :
"Parallel Program Schemata" - JCSS, Vol.3, 1969.

Miller :
"A Comparison of some Theoretical Models of Parallel Computation"
IEEE T-C-22, 1973.

Flynn :
"Some Computer Organisations and their Effectiveness" - IEEE T-C-21, 1972.

Miller & Cocke :
"Configurable Computers : A New Class of General Purpose Machines" -
Lecture Notes in Computer Science 5, Springer-Verlag 1974.

Glushkov et. al. :
"Recursive Machines and Computing Technology" - Information Processing 1974.

Dennis & Misunas :
"A Preliminary Architecture for a Basic Data Flow Processor" - Proc.
IEEE Symp. on Computer Architecture 1975.

Iliffe :
"Basic Machine Principles" - Macdonald 1968.

Dennis :
"First Version of a Data Flow Procedure Language" - Lecture Notes in
Computer Science 19, Springer-Verlag 1974.

Hoare :
"Communicating Sequential Processes" - Draft: Queen's Univ. Belfast 1976.

Tesler & Enea :
"A Language Design for Concurrent Processes" - SJCC 32, 1968.

Chamberlin :
"The 'Single-Assignment' Approach to Parallel Processing" - FJCC 39, 1971.

19th January 1977 .
J. R. Gurd.

DATA DRIVEN SYSTEM FOR HIGH SPEED PARALLEL COMPUTING— PART 2: HARDWARE DESIGN

Prototype design for a data driven computer uses highly parallel hardware to speed up computation and implements a 2-dimensional flowgraph model of computing that departs markedly from traditional approaches to realize increased parallel activity in software

John Gurd and Ian Watson University of Manchester
Manchester, England

A prototype data driven parallel computer being built at the University of Manchester, Manchester, England, implements a comprehensive model of computing based on data flowgraphs. Using the model created in Part 1, Part 2 determines the requirements for a system to execute the labeled flowgraph model. These lead to a circular pipelined architecture in which packages of information representing tokens, arcs, and nodes can circulate.

Investigation of parallelism in hardware as a means of speeding up computations discussed in Part 1 identified the pipeline and the parallel array as two basic hardware structures. A key factor in using such hardware is the ability to express parallel activity in software. Since conventional programming languages, based on the classical von Neumann concepts of sequential memory access using a program counter, are not suited to expressing parallel activity, a data driven model of computing was developed to clarify the dependency relations between data and to expose the potential for parallel activity in 2-dimensional flowgraphs. Computers based on this computational model have proven to be highly concurrent, but quite different in structure from traditional computers.

Studying one particular data driven computer* that implements the model previously developed, Part 2 details the design of individual pipeline stages and estimates system performance. Future developments in high level software and very high performance systems are also considered.

It is apparent from the rules for executing flowgraphs that a data driven computer must be capable of performing the following activities: transferring a token from the tail to the head of the arc on which it lies; grouping together tokens of the same color lying at the heads of arcs that point to a common node, and arranging for the node to fire when all of its inputs are available; and executing operations on input data, erasing the input tokens, and producing output tokens at the tails of the appropriate arcs. These actions must be performed continuously until no tokens remain in the graph. Furthermore, to present data as input and to receive results from the computation, there must be a means of inserting tokens into the system and extracting them from the system.

Data Driven Architecture

The structure in Fig 1 reflects these needs. It comprises five units connected together in a ring. The major traffic around

*The system described is a small prototype machine based on an outline design proposed at Manchester, England, in 1978.[1] It is currently being constructed by the authors with funding from the Distributed Computing Systems Programme of the Science Research Council of Great Britain[2] and is expected to be operational in mid-1981.

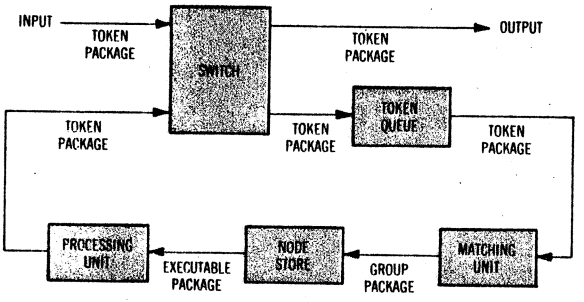


Fig 1 Data driven computer structure. Five major functional blocks communicate in clockwise direction around ring. Token package is main unit of information comprised of data value, label, and destination node pointer. Matching unit groups tokens. When sufficient tokens arrive to fire node, appropriate group package finds destination node description in node store. Executable package containing operator, all operands, label, and pointer(s) to further destination node(s) is sent to processing unit for execution. Switch handles external I/O. Token queue saves excess tokens

the ring is in token packages representing the machine equivalent of labeled tokens lying on flowgraph arcs. Token packages are either sent to the ring as external input or else formed during a computation at the output of the processing unit. They pass around the ring in a clockwise direction, first encountering the switch, which serves to merge the two sources of tokens and to direct output from the system, and then the token queue, which holds the large numbers of tokens that highly parallel programs can generate. The third unit, called the matching unit, groups together tokens with the same label traveling to the same node.

When sufficient matching tokens are present, the node can be fired and all tokens can be sent to the next unit in a group package. Otherwise, each incoming token must be placed in the matching store to await the arrival of further corresponding tokens. Fired groups of tokens find the operation to be performed on them in the node store. For convenience, they are also given the addresses of the nodes to which output should be sent when execution is completed. The node store thus constructs complete executable packages consisting of an operator, operands, common label, and one or more destinations for the resultant values. These packages are sent to the processing unit where the required operation is executed, eventually producing new tokens that start a repeat journey around the ring.

Notational Influences

It is important to reiterate the influence of the labeled flowgraph notation in formulating this structure. Each token carries not only its value, but also a label and a destination, including the place of entry to the node (eg, the left- or right-hand side of an addition operator). This requires each word of token queue memory and matching memory to be much longer than a conventional memory word containing only a data value.

Moreover, the matching operation requires an associative memory for saving unmatchable input tokens. The common name used for association is the label and the destination (except for points of entry, which should of course differ). To conform to the required flowgraph behavior, the matching unit must perform two kinds of access to the associative memory after searching for items with the same name as the incoming token. If the search succeeds, and all matching tokens are present, matched token entries are extracted from the associative memory and sent to the node memory. If the search fails, or one or more required tokens are missing, the incoming token is inserted into the associative memory.

Finally, the processing unit instruction set must include data branch, token relabeling, and token delabeling operations, apart from the usual arithmetic functions. The instruction set naturally excludes conventional control branch or jump instructions and memory to memory instructions.

Parallelism

The ring structure of Fig 1 is a natural candidate for pipeline construction as described in Part 1. Individual units interact only by sending token packages around the ring; hence, the individual switching, queuing, matching, node access, and processing operations implemented by the five functional units can be overlapped and even allowed to overtake one another. The only critical area of the system is the matching unit, where the associative memory must be searched serially. The speed at which tokens can be matched at the associative memory effectively determines the pipeline delay period. Elsewhere, functional units can be pipelined or paralleled as convenient to match this time period. In the prototype system under construction, it has proved necessary to place several processing elements in a parallel array within the processing unit. All other units are composed solely of pipeline stages operating serially at the required rate.

Logical System Design

Two major features of a pipeline are its delay period and its synchronization type. Use of transistor-transistor logic technology determines the 200-ns delay period chosen for the prototype system. Both the modest technology and relatively slow speed are deliberate since the prototype serves only to test out ideas for data driven computing and is not intended as a very high speed production computer.

The choice between a synchronous or an asynchronous pipeline involves a compromise. Synchronous systems are attractive in situations where all operations take similar times to complete, and whenever arbitration between competing tasks is needed. However, the problems of distributing a common-phase clock in a physically large synchronous system mean that asynchronous communication is often preferred.

In the compromise design, local clocks are used within each major functional unit of Fig 1, with synchronous internal arbitration, either pipelining or paralleling, as appropriate. An asynchronous, unidirectional communication

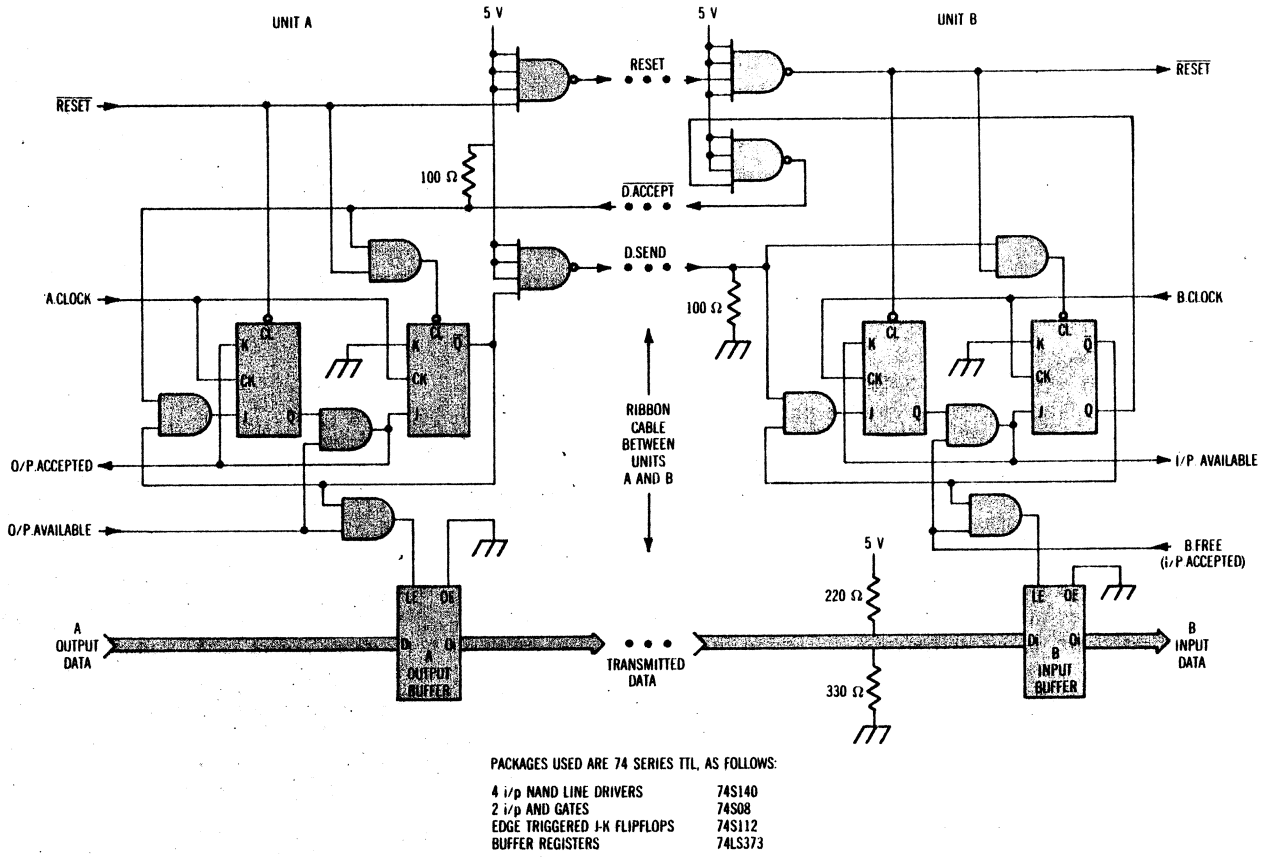


Fig 2 Interunit pipeline communication circuit. Each unit has its own local clock to synchronize internal events. Asynchronous transfer between units is synchronized to appropriate clock at each end by this circuit. Handshake between D.SEND and D.ACCEPT helps transmit data from output buffer of one unit to input buffer of next unit. Circuits are initialized by RESET, which is passed around pipeline from host system

link between each pair of functional units provides buffering and resynchronization at the input and output of each unit. Fig 2 shows logical circuits for controlling the input and output interfaces. An asynchronous handshake arrangement achieves data transfer by using the D.SEND and D.ACCEPT signals. A pair of edge-triggered J-K flipflops synchronizes the handshake to the appropriate clock at each end. While use of two flipflops slows the transfer slightly, it allows resolution of uncertainty when handshake signals change at the same instant as the synchronizing clock. This situation has long been recognized as a major trouble source in systems using asynchronous communication.[3]

Sender and receiver clock signals derive from unrelated 40-MHz sources, except in the matching and processing units, which use 25- and 10-MHz clocks, respectively. Delays through logic at either end of the circuit, together with the need for four delays through the linking cable, imply that communication over distances of up to 10 ft (3 m) is possible within the 200-ns pipeline delay period.

Data Formats

For various reasons such as board size, connector density, expected costs (especially for memory), and anticipated system usage, token packages were designed to have a basic word length of 96 bits. Because all communication in the system takes place via the pipeline, one bit must be used to distinguish between system messages (eg, load a value into the node memory) and computational messages (eg, an actual token). In the case of token packages, the remaining 95 bits are allocated among a 37-bit value field, a 36-bit label field, and a 22-bit destination field. The method of labeling tokens and controlling data flow through program graphs brings about these unusual field sizes. The largest data value recognized is a 32-bit floating point value, and integers are 24 bits long. It can be seen from these figures that the storage requirements for data driven execution with labeling exceed conventional storage requirements by a factor of three or more.

Pragmatic arguments restrict the maximum number of tokens that can be associated or matched up in the matching unit to two. This is not a severe theoretical limitation, since it is possible to simulate any multiple-input node with a primitive graph containing only 1- and 2-input nodes. It is a useful, practical restriction because it permits completion of the matching operation within the required 200-ns time limit. However, it reduces the execution efficiency of some functions.

Similar reasoning limits the maximum number of data value copies that any primitive node can make to two. Once again, this is not a theoretical problem, since a tree of primitive, 2-output copy nodes can produce a multiple set of copies. However, again, efficiency may be affected.

Taken together, these decisions fix the sizes of group packages and executable packages at 133 and 167 bits, respectively. A group package is simply a token package with an extra 37-bit value field. An executable package consists of two 37-bit value fields, two 22-bit destination fields, one 36-bit label field, and a 12-bit operator field. The remaining bit is used to mark system messages.

The node memory has a 35-bit word length, of which 3 bits are used to check the validity of different types of access, and the remaining 32 bits hold information about a node. One or two words may represent a node in the node memory. The first word always defines the operation to be performed at the node and a destination node to which the result token should be sent. The optional second word may define either a second destination node for the result (eg, copy operation) or a literal value to be used as one of the operands at the node. Literals can appear only when the matching unit has not been used to pick up a matching token. They are useful when performing single-input functions that are of 2-input form but with one input constant (eg, increment value, which is equivalent to adding the value to a constant).

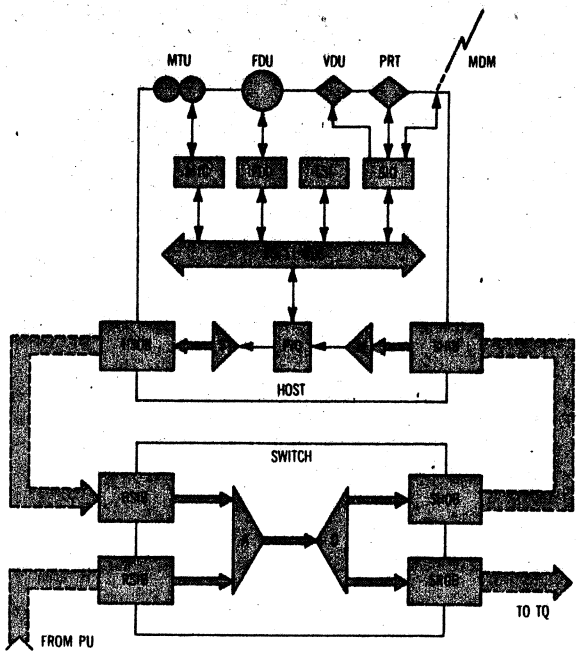
Input/Output and Host System

The prototype system is capable only of data driven computation. Because it cannot control peripheral devices directly, a conventional host system handles file storage and input/output (I/O). Two asynchronous interface links attach the host to the data driven ring (Fig 3). The host machine is a DEC LSI-11 with a range of standard peripheral devices, including a telephone link to external mainframes. Although it is not the ideal system for its position because of its relatively small word length and slow speed, it is adequate as long as I/O traffic remains light. The LSI-11 drives the data driven ring through a 16-bit parallel interface and a 16- to 96-bit word expander. It receives output from the ring via the 96- to 16-bit word compressor and the same parallel interface.

An unusual pipeline unit, the switch must arbitrate between two separate sources of input: the host system and the processing unit. The synchronized nature of the two input buffers simplifies this task. Whenever two packages arrive simultaneously, the switch gives priority to the package coming from the processing unit. To avoid unreasonable delays in processing unit output, the switch operates within a 100-ns period. Output packages are recognized at the switch input and routed to the host exit output buffer. All other packages are sent to the ring output buffer.

Token Queue

Implemented as a first in, first out circular buffer for token packages, token queue (Fig 4) memory is 16k words by 96 bits plus parity, built of 70-ns static random access memory. Total memory access time is 100 ns, and the controller uses this high speed by interleaving read and write cycles to consume input and generate output at the required 200-ns intervals. If the memory buffer is full when a read cycle is ready to start, the controller performs a dummy read cycle instead; it then checks for input and, if input is present, performs a write cycle. No attempt is made to change the pattern of 100-ns alternating read and write cycles. To keep this strategy from causing uneven periods between token packages further down the pipeline, an additional, synchronous, output buffer stage is inserted in the pipeline in this unit.



DOUBLE DOTTED LINES REPRESENT ASYNCHRONOUS COMMUNICATIONS LINKS

MTU = MAGNETIC TAPE UNIT	S = 16-BIT TO 96-BIT DEMULTIPLEXER
MTC = MAGNETIC TAPE CONTROLLER	A = 2-WAY SYNCHRONOUS ARBITRATOR
FDU = FLOPPY DISC UNIT	D = 2-WAY DISTRIBUTOR
FDC = FLOPPY DISC CONTROLLER	SHIB = SWITCH TO HOST INPUT BUFFER
VDU = CONTROL TERMINAL	HSOB = HOST TO SWITCH OUTPUT BUFFER
PRT = 30-CHAR/s DECWRITER	HSIB = HOST TO SWITCH INPUT BUFFER
MDM = 300-BAUD MODEM	RSIB = RING TO SWITCH INPUT BUFFER
SIO = 4-CHANNEL SERIAL INTERFACE	SROB = SWITCH TO RING OUTPUT BUFFER
LSI = LSI-11/2 + 28-k WORD MEMORY	SRHB = SWITCH TO HOST OUTPUT BUFFER
PIO = 16-BIT PARALLEL INTERFACE	PU = PROCESSING UNIT
M = 96-BIT to 16-BIT MULTIPLEXER	TQ = TOKEN QUEUE

Fig 3 Host system interface. LSI-11 frontend computer drives ring via switch module. Multiplexer and demultiplexer convert between 16-bit host word and 96-bit token package lengths. Host then mimics any other pipeline unit with asynchronous I/O communication links. Switch arbitrates between competing inputs internally and synchronously, avoiding the problem of asynchronous arbitration

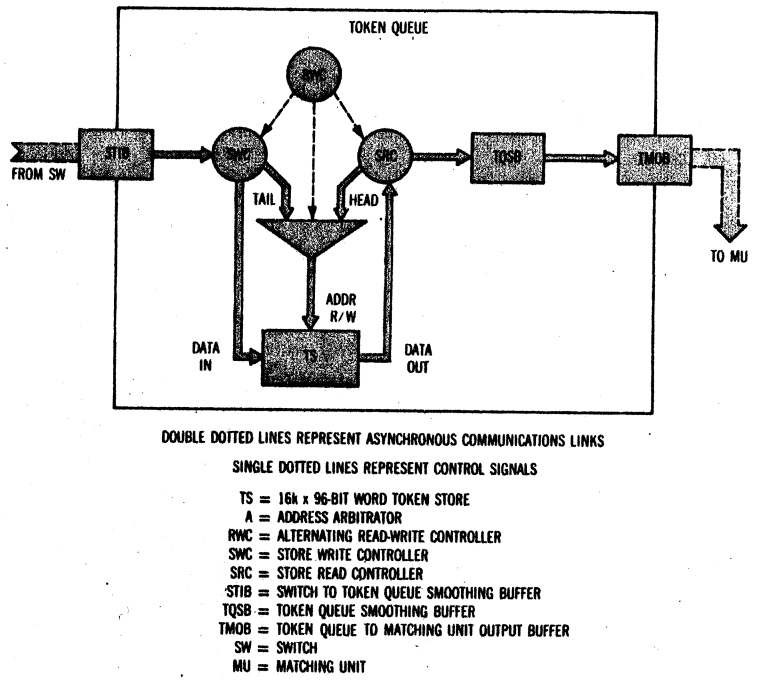


Fig 4 Token queue. Highly parallel programs may generate large numbers of tokens, many of which must be held temporarily. Basis of unit is 16k-word circular queue with 100-ns access time. Alternate attempts to read and write the store are made at head and tail of queue, respectively. Empty store, or absence of input, postpones read or write cycle for 200 ns. Extra synchronous buffer stage smooths irregular output

Matching Unit

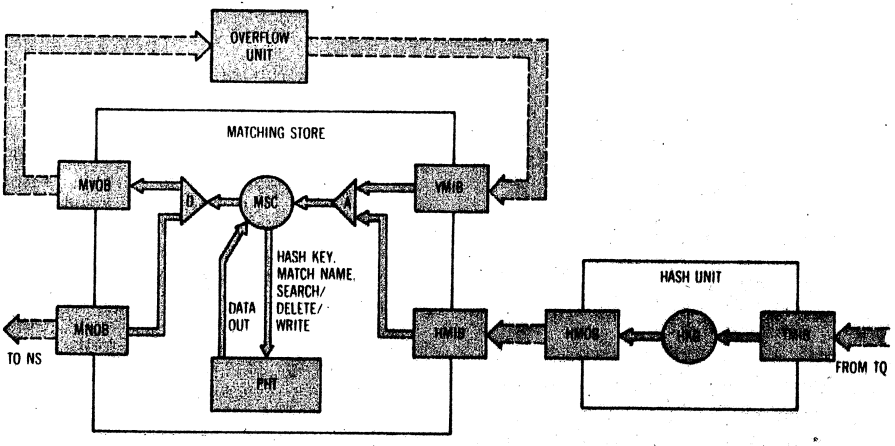
Token matching unit operation is the most critical part of the system. The matching store is relatively large (16k token packages plus parity) and must be accessed associatively on 54 bits of the label and destination fields. Present costs preclude using a true content-addressable memory of this size; therefore, a hardware hashing technique simulates the associative memory by implementing a parallel hashing scheme that searches several hash tables for a matching entry simultaneously, as shown in Fig 5.[4, 5, 6]

A separate pipeline stage precedes the main section of the matching unit. Here, an 11-bit hash key is generated from the label and destination fields of the incoming token package. The token package and the hash key are transferred to the matching memory's input buffer. At this point, examining a bit of the destination field determines whether matching is appropriate for this token package. Single tokens, for which matching is not appropriate, pass directly to the output buffer.

A hash table search locates the partner for other tokens. In this case, the hash key is used as an address to read the eight banks of the hash table in parallel. The label and destination fields of all eight accessed locations are compared with the corresponding sections of the input token package. Successful matches then cause the matching entry to be deleted from the hash table, and the matched token pair, together with its label and destination, is forwarded to the output buffer.

Unsuccessful matches normally cause the incoming token package to be placed at a free position within one of the parallel memory banks still addressed by the hash key. Occasionally, this will not be possible because all words addressed by the key already contain nonmatching token packages. In this last case, the system departs from conventional techniques. Instead of generating a new hash key and starting all over again, the controller routes the unmatched token package to an overflow unit where it can be processed at leisure.[4, 5] One advantage of the data driven notation is that subsequent token packages can be handled out of turn, providing that their hash keys are disjoint, without affecting computation results.

The overflow unit complicates the operation just described. For example, after an unsuccessful match, a token cannot be placed in a free location if an overflow has occurred for its hash key; instead, the nonmatching token package must be referred to the overflow unit. Also, there must be a return path from the overflow unit to the system. This is located at the input to the matching store itself, and provides potential competition for normal ring traffic.



DOUBLE DOTTED LINES REPRESENT ASYNCHRONOUS COMMUNICATIONS LINKS

HKG = HASH KEY GENERATOR
MSC = MATCHING STORE CONTROLLER
PHT = 8 x 2k x 96-BIT WORD PARALLEL HASH TABLE
A = 2-WAY SYNCHRONOUS ARBITRATOR
D = 2-WAY DISTRIBUTOR
TMIB = TOKEN QUEUE TO MATCHING UNIT INPUT BUFFER
HMOB = HASH UNIT TO MATCHING STORE OUTPUT BUFFER
HMIB = HASH UNIT TO MATCHING STORE INPUT BUFFER
VMIB = OVERFLOW TO MATCHING STORE INPUT BUFFER
MVOB = MATCHING STORE TO OVERFLOW OUTPUT BUFFER
MNOB = MATCHING STORE TO NODE STORE OUTPUT BUFFER
TQ = TOKEN QUEUE
NS = NODE STORE

Fig 5 Matching unit. Critical system area contains two pipeline stages and parallel overflow unit. Main matching store consists of parallel hash table. Preceding pipeline stage generates hash key. Access to store may be successful or unsuccessful; successfully matched tokens are extracted from store and sent to node store via output buffer. Unmatched tokens are usually written to spare table location. If hash line overflows, token is referred to separate overflow unit. Provided that hash keys differ, matching unit handles further tokens even before overflow outcome is known. Output occurs once every 300 ns on average

Simulation studies indicate that matching unit throughput remains largely unaffected until the hash table is about three-quarters full.[6] This compares favorably with other hashing schemes.

Viewed overall, the store and release mechanisms in the matching unit alter the nature of the pipeline between input and output. Output is less frequent than input but involves greater amounts of information. Simulation predicts that token packages arriving at 200-ns intervals will produce group packages every 300 ns on the average. This figure, however, depends critically on the form of programs. The basic hash table access time is 160 ns, and the three possible access types, neglecting overflow, take 40 ns for bypass, 240 ns for a successful match, and 320 ns for an unsuccessful match attempt. Given roughly even numbers of each access type, as would be expected, this yields the average figure cited. However, a long sequence of unsuccessful match operations results in inefficient use of later pipeline stages.

Node Store

Two pipeline stages implement the node store (Fig 6). The first stage accesses a segment table containing 64 entries addressed by the six most significant bits of the incoming destination field. Each segment table entry points to the base of a segment of main node storage and holds an indication of the segment length. The second stage of the node store unit accesses the required node using 12 bits of the

destination field as an offset from the segment base. Accesses beyond the end of a segment flag an error. Node entries may be one or two words long; 2-word entries are accessed in sequence.

Depending on the kind of node entry discovered, the node store constructs an executable package for output to the processing unit. These packages are consumed at an average rate of one every 300 ns, the same rate at which input arrives at the node store. The 16-word main node store has an access time of less than 200 ns so that it can handle programs in which up to half of the nodes require 2-word entries.

Processing Unit

Prototype computer system design dictates that all design decisions offer maximum flexibility. In particular, it is prudent to anticipate instruction set changes. This requires development of a microprogrammable processing unit. Because flexible microprogramming is relatively time consuming, it proved necessary to use a parallel array of processing elements within the processing unit to maintain the required throughput by processing one executable package every 300 ns. Fig 7 shows the basic structure of the processing unit.

The first pipeline stage executes certain high speed (200-ns) operations that cannot be performed within the parallel array. These operations control generation of activity names in the system and allow for gathering performance statistics. The main processing area comprises the parallel array of processing elements with appropriate input and output buffers. Each element is built around a 24-bit arithmetic unit constructed from 4-bit AMD2900 series bit slice microprocessors. A microprogram controller with a writable microcode store controls the main processing area to achieve the required, flexible instruction set.

Processing elements have a 200-ns microcycle time, and instruction execution periods that vary from 5 to upwards of 50 microcycles. The average expected execution time for a processing element is 4.5 μs per operation. Hence, the number of elements required to maintain the throughput at one executable token package every 300 ns is 4.5/0.3 = 15. More elements would leave a high percentage of elements idle because the pipeline could not provide input quickly enough to keep them busy; fewer would lead to a general slowing down of the system because the processing array could not cope with input as fast as the pipeline supplies it.

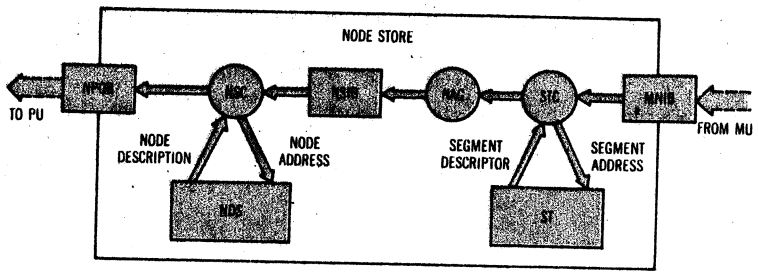
Token package execution results in the production of zero, one, or two result token packages. The expected combination of these different behaviors is such that the average consumption of executable packages once every 300 ns will lead to an average production of result token packages once every 200 ns.

The processing unit is synchronized with a relatively slow (10-MHz) clock to simplify the problems of distributing packages to and collecting packages from the processing elements. Each distribution or arbitration cycle is selected in advance and then executed in synchrony with the clock.

System Performance

The most important feature of the pipeline just described is that all stages are independent from buffer to buffer. Each stage relies only on its input and its internal storage. This means that the system performance is determined by the overall amount of work provided, in the form of token packages, together with the average timing characteristics of the pipeline.

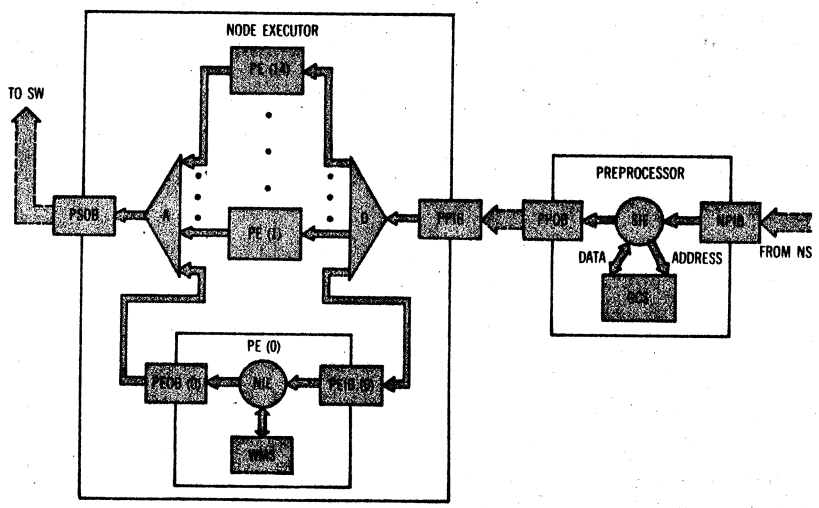
Average delay times associated with various pipeline stages yield a maximum processing rate of one executable package every 300 ns. Using the approximation that one



DOUBLE DOTTED LINES REPRESENT ASYNCHRONOUS COMMUNICATIONS LINKS

STC = SEGMENT TABLE CONTROLLER
ST = SEGMENT TABLE
NAG = NODE ADDRESS GENERATOR
NSC = NODE STORE CONTROLLER
NDS = NODE DESCRIPTION STORE
MNIB = MATCHING UNIT TO NODE STORE INPUT BUFFER
NSIB = NODE STORE INTERMEDIATE BUFFER
NPOB = NODE STORE TO PROCESSING UNIT OUTPUT BUFFER
MU = MATCHING UNIT
PU = PROCESSING UNIT

Fig 6 Node store. Two separate pipeline stages comprise node store. First stage accesses segment descriptor from 64-entry segment table and forms read address of required destination node. Second stage accesses node from one or two words in main node store. Resulting executable package is sent to processing unit via output buffer



DOUBLE DOTTED LINES REPRESENT ASYNCHRONOUS COMMUNICATIONS LINKS

SIE = SPECIAL INSTRUCTION EXECUTOR
GCS = GLOBAL CONTROL STORE
D = 15-WAY SYNCHRONOUS DISTRIBUTOR
A = 15-WAY SYNCHRONOUS ARBITRATOR
PE = PROCESSING ELEMENT
NIE = NORMAL INSTRUCTION EXECUTOR
WMS = WORKSPACE AND MICROPROGRAM STORE
NPIB = NODE STORE TO PROCESSING UNIT INPUT BUFFER
PPOB = PREPROCESSOR TO PROCESSOR OUTPUT BUFFER
PPIB = PREPROCESSOR TO PROCESSOR INPUT BUFFER
PEIB = PROCESSING ELEMENT INPUT BUFFER
PEOB = PROCESSING ELEMENT OUTPUT BUFFER
PSOB = PROCESSING UNIT TO SWITCH OUTPUT BUFFER
NS = NODE STORE
SW = SWITCH

Fig 7 Processing unit. Parallel array of 15 bit-sliced, microprogrammable processing elements achieves main processing capability. Distribution and arbitration are performed synchronously. On average, unit consumes executable package once every 300 ns and produces tokens at 200-ns intervals. Preprocessing stage performs system and monitoring tasks that cannot be distributed to individual processors because they are not strictly functional operations

executable package is equivalent to a conventional machine instruction in a comparable processor with a 32-bit word length and floating point operations, this gives a throughput of roughly 3.3M instructions/s. In terms of useful floating point operations, this has been found equivalent to about 1.1M floating point operations/s for one machine-coded example.

It must be stressed, however, that these are maximum rates that rely on uniform program behavior at various stages of the pipeline. Three secondary factors will cause performance degradation from the optimum. First of these is a long sequence of unmatchable tokens arriving at the matching unit. High performance relies on an equal mix of 1- and 2-input nodes in the program and a smooth distribution of matching store accesses resulting from these nodes. Secondly, the node store should receive no more than half of its total requests for double-length entries. These are typically used for instructions with literal constant operands and for copy operations; they occur quite frequently in practice. Finally, the processing unit requires an even mix of different kinds of executable packages to maintain the average execution period. A long sequence of lengthy instructions (eg, floating point division) will lower the throughput.

The exact effect of these factors has not yet been determined, but it is expected to be relatively small when compared with the primary problem of providing enough work to keep the whole pipeline occupied. Given an average, well-behaved program, it can be seen from the diagrams that there are a total of 16 pipeline stages plus 15 processing elements to keep occupied if maximum throughput is to be achieved. Thus, the program should provide at least a 31-fold degree of parallelism. The measurement of parallelism in programs is an open problem that has received a great deal of recent attention.[7, 8]

Future Developments

Although the Manchester University research group is engrossed in practical details of machine implementation at the moment, there has been ample opportunity to anticipate

future challenges when the system is completed in 1981. Two interesting areas of study have emerged. One is the development of richly expressive, and yet efficiently translatable, high level languages for programming data driven systems. The second is realization of an extensible computer whose power may be increased without theoretical limit simply by adding more and more hardware modules.

High Level Languages

It is no longer practical to design computer systems without regard for the programs they are intended to run, and most programs currently in use are written in a high level language. Consequently, many machines, including data driven machines, are designed around specific programming languages, and others, such as the one described here, are developed in close conjunction with a programming discipline.[9, 10, 11]

For general purpose systems, there is a broader requirement to run a variety of different languages, and it is interesting to see how versatile the data driven system can be in implementing these. Programming languages can be classified into three groups for this purpose.

The first group comprises those languages developed around data flowgraphs, usually for high speed parallel computing. These are expression oriented, or single-assignment languages with parallel semantics based on a data driven view of program execution. Of course, they are therefore a natural high level vehicle for data driven computers, as Part 1 of this article demonstrated in discussing the ease of translation into graphical machine code.

More of a challenge is presented by the second group of high level languages. It consists of the traditional, von Neumann languages with their semantics based on variables in fixed storage locations and sequential execution of program statements. Sequential flowgraphs for such programs can be generated by a straightforward, if somewhat tedious, translation.[12] However, the challenge is to unwind the sequential execution completely so that as much work as possible may be performed in parallel. This requires the use of sophisticated loop unraveling techniques together with software flow analysis.[13, 14]

Languages whose semantics are purely mathematical and not based on any particular view of program execution comprise the third group. These are founded mainly on the theory of mathematical functions (the lambda calculus) with the important exception of the language LUCID, which is based on an algebra of histories.[15] Because these languages are not machine oriented, they are difficult to implement efficiently on conventional computers. The principal problem is that of minimizing the amount of computation performed. Although the problem is also present for data driven implementations, there is some indication that the parallelism available to be exploited in data driven systems allows rapid execution of functional programs.

At present, high level software for data driven systems is at an early experimental stage. Pilot compilers for all three groups of high level languages have been written, but there is no clear indication of an optimal language among the candidates. On the other hand, most researchers agree that languages based on the von Neumann model are not appropriate for specification of highly parallel algorithms.

Extensible Hardware

Data driven notation was developed as a means for expressing highly parallel computational activity. However, the architecture described here is limited in the amount of parallelism it can exploit by the need for sequential token matching within a serial pipeline. Since the prototype system has proven to have just over 30-fold hardware parallelism, problems with much greater amounts of parallelism will run relatively slowly. Two possible improvements to the prototype architecture may allow much higher rates of computation.

The first improvement would not alter the basic pipeline structure at all, but would instead increase throughput by making the primitive machine operations much more substantial than the simple arithmetic operators suggested in Part 1. Each executable token package could then represent a much larger fraction of the whole computation, and the fixed rate of package execution would increase the overall processing rate. While this might require a larger complement of more complex processing elements, the biggest difficulty lies in deciding which powerful primitive operations should be included in the instruction set.

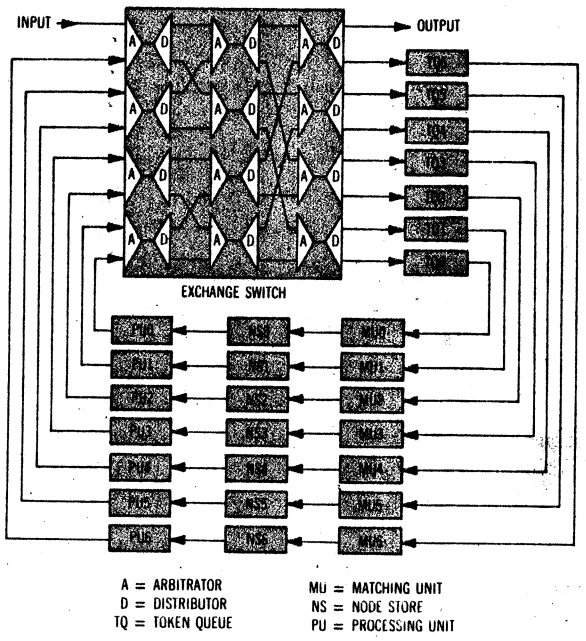


Fig 8 Parallel array of data driven rings. Very high processing rates might be achieved by connecting large numbers of pipelined rings in this kind of structure. Unidirectional, pipelined exchange switch is modularly extensible and of relatively simple form as compared to crossbar switch, for example. In very large systems, major problem is to distribute workload evenly among data driven rings

The second improvement involves changing the system structure so that several serial pipelines can operate simultaneously in a parallel array. The nature of the matching operation is such that tokens with similar colors and destinations could be isolated in separate pipelines as long as the output from any processing unit can reach the input to any matching unit. The unidirectional nature of the pipeline makes this feasible with relatively low cost in time and hardware complexity. The resulting system has a number of pipelined ring structures connected in layers to a greatly extended switch unit, as shown in Fig 8. The time delay through the switch increases logarithmically with the number of rings.

Consequences of this multilayered structure are far reaching. It seems capable of indefinite expansion, providing continually increasing computing power for problems with sufficient parallelism to keep the hardware busy. Of course, there are many practical difficulties that are largely concerned with the problems of distributing work evenly between layers of the system.

Conclusions

Starting with the observation that the speed of computation for some problems needs to be much higher than is presently achievable, and that the best hope for achieving the required speed is through the use of parallel hardware, a novel parallel computer architecture with some interesting properties has been developed. The architecture is based on a data driven, graphical model of computation that views program structure and execution in two dimensions rather than one. A prototype system that uses low key technology and yet is capable of executing about 3M instructions/s is currently under construction.

Several other systems are at comparable stages of development. In particular, researchers at the University of Utah and Toulouse already have operational systems, and a group at MIT is about to start construction of a machine.[9, 16, 17] This article did not attempt to compare these systems with the one described because they are not based on the same type of data driven model. An outline system design based on a similar model has been developed at Irvine, but because details of the system are not yet known, again no comparison was attempted.[10]

It is expected that some alteration in programming techniques will be needed to use this kind of system efficiently. However, the changes may be of a kind that already are being introduced for reasons of reliability, portability, and maintainability of software. If the anticipated increase in throughput can be achieved in multilayered data driven structures, it is likely that the required changes will present a minor obstacle to development of such a system.

Acknowledgments

We would like to acknowledge the assistance of all the members of the Data Flow Research Group at Manchester, whose many ideas have contributed continually to the development of the data driven computer. Chris Kirkham also was kind enough to read and comment upon the draft manuscript. In addition, the financial support of the Distributed Computing Systems Programme of the Science Research Council of Great Britain is gratefully acknowledged.

References

1. I. Watson and J. R. Gurd, "A Prototype Data Flow Computer with Token Labelling," *AFIPS Conference Proceeding,* NCC, June 1979, pp 623-638
2. Science Research Council, "The Coordinated Programme of Research in Distributed Computing Systems: Annual Report," Science Research Council of Great Britain, Dec 1979
3. G. R. Couranz and D. F. Wann, "Theoretical and Experimental Behaviour of Synchronisers Operating in the Metastable Region," *IEEE Transactions on Computers,* Feb 1975, pp 604-616
4. E. Goto and T. Ida, "Parallel Hashing Algorithms," *Information Processing Letters,* Feb 1977, pp 8-13
5. T. Ida and E. Goto, "Performance of a Parallel Hash Hardware with Key Deletion," *Information Processing 77,* North Holland, New York, 1977, pp 643-647
6. J. G. D. da Silva and I. Watson, "A Pseudo-Associative Matching Store Using Hardware Hashing," Dept of Computer Science, University of Manchester (Submitted for Publication)
7. S. Winograd, 'On the Speed Gained in Parallel Methods," *New Concepts and Technologies in Parallel Information Processing,* E. Cainaniello (Ed), Noordhoff, 1975, pp 155-166
8. D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys,* Mar 1977, pp 29-59
9. J. C. Syre, *et al,* "LAU System: A Parallel Data Driven Software/Hardware, System Based on Single Assignment," *Parallel Computers—Parallel Mathematics,* M. Feilmeier (Ed), 1977, pp 347-351
10. Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Information Processing 77,* North Holland, New York, 1977, pp 849-853
11. J. R. W. Glauert, "A Single Assignment Language for Data Flow Computing," MSc Dissertation, Dept of Computer Science, University of Manchester, Jan 1978
12. R. E. Miller and J. D. Rutledge, "Generating a Data Flow Model of a Program," *IBM Technical Disclosure Bulletin,* Apr 1966, pp 1550-1553
13. D. J. Kuck, Y. Muraoka, S-C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *IEEE Transactions on Computers,* Dec 1972, pp 1293-1310
14. L. Lamport, "The Parallel Execution of DO-Loops," *Communications of the ACM,* Feb 1974, pp 83-93
15. E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM,* July 1977, pp 519-526
16. A. L. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," *AFIPS Conference Proceedings,* NCC, June 1979, pp 1079-1086
17. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," *Proceedings of the Second Annual Symposium on Computer Architecture,* IEEE, Jan 1975, pp 126-132

THE DATA FLOW PROJECT

A document was distributed approximately one year ago outlining a possible parallel computer architecture using a 'data driven' concept.

Since that time work has been progressing on several aspects of the idea. A simulator has been written to verify the operation of the architecture and to enable an investigation of programming such a machine.

Several people in the department have expressed an interest in the progress of the idea, this document is intended to provide some of that information. Since the circulation of the original document was limited, a description of the aims and principles of data flow is presented. This is followed by a description of the architecture as it is envisaged at present, and some efficiency results obtained with the help of the simulator are included. Clearly there have been several intermediate stages in the evolution of the designs and further stages are yet to come. It is hoped however that a description of the present design will be of interest and highlight the problems and potential of data flow architecture.

Ian Watson
John Gurd
25/5/77

PROCESSOR PARALLELISM - A DATA FLOW SOLUTION AT SINGLE INSTRUCTION LEVEL

Introduction

One of the major roles of the computer designer during the last two decades has been the incorporation of new technologies into the design of computers. This together with many architectural improvements has resulted in a ten thousand fold increase in the speed of computing machines since their initial conception. In addition the cost of components has been reducing rapidly until at the present time powerful computing devices, in the form of microprocessors, are available at a cost which will permit their use in a very wide range of applications.

The fastest machines are still constructed from many discrete components, although the level of integration is increasing rapidly. If, as the technologists believe, the size of integrated circuits can still be increased significantly, then very fast computers with very large amounts of random access storage will be available as 'ready made' components. Furthermore, as the speed of machines increases, limitations imposed by the speed of light will necessitate the miniaturisation of computing devices. It is probable therefore, that the further development of conventional computing machines will become a job for the technologist. Unless the designer can produce significant architectural improvements, his influence will decline. Processor Parallelism is a possible candidate for the designers attention.

A Case for Parallelism?

It has been almost universally assumed that increases in the speed of computing have been worthwhile in that they have enabled the solution of more complex problems in a realistic time. Although the power available in the microprocessor is sufficient for a large proportion of present applications, it is difficult to believe that mankind has anywhere near the capability to solve all computable problems. The"PDP-11 in every room" philosophy may provide the majority of users with the facilities which will satisfy their present needs but it will do little for the advancement of computation.

What is the nature of problems which are likely to be computation limited? In the present situation, large amounts of computing time are spent in solving problems in pattern recognition, artificial intelligence, atomic and crystal structure analysis, weather forecasting,finite element analysis of mechanical structures and many other similar fields. One common denominator is that all can be specified as array computations where processing can take place on elements of the array in parallel. It would seem that a significant increase in computing speed would result if a machine were produced which could exploit this problem parallelism.

There are wide ranging views as to how significant parallelism at a single problem level will become in the future of computing. From those who believe that parallelism only exists in a very small number of problems and even then it is swamped by intervening serial computation, to those who consider that all problems would have significant parallelism if only the correct algorithms were devised.

The truth almost certainly lies somewhere between, little effort has been made to investigate parallel algorithms for standard computing tasks but many will remain essentially serial. However there is no doubt that in the areas already mentioned there are vast parallel tasks which are at present using several hours of machine time for an individual computation.

Most of these computations are performed on machines which provide a central computing service to a research establishment. If such a machine were to be replaced by a parallel one then its power must be usable when large parallel programs are not running. As long as a parallel machine is flexible there is no reason why the parallelism exhibited by many users running concurrently should not be exploited.

Given these conditions there would appear to be a case for examining parallel computing as a method of extending the limit to computing speed.

Possible Parallel Machine Configurations.

Single Instruction Stream Multiple Data Stream (SIMD) Machines

These machines operate as a very simple observation that in some highly regular parallel problems, each point in the array is undergoing exactly the same computation but on a different piece of data. The parallelism can then be exploited by having many machines each executing the same set of instructions at the same time. Several machines notably the ILLIAC IV and the ICL DAP have been constructed using this principle.

Most parallel problems are not perfectly regular, and the performance of the machine in an irregular parallel mode is little better than a single data stream machine. Although such a machine has a limited use it is by no means the ideal implementation.

Multiple Instruction Stream Multiple Data Stream (MIMD) Machines

It is clear that the ideal parallel machine must be capable of supporting many processors each executing a separate instruction on a separate piece of data. If any decent efficiency is to result, every processor must be kept fully occupied and the overheads of managing the system must not be excessive. One approach (that takes in the Carnegie-Mellon C.mmp) is to connect several processors, each with their own local store, to a central common store. An operating system will then be written to allocate tasks to processors; the usage of machines and the overheads will be found when and if the system starts to run. A possible approach but not one which appeals to the cautious.

A more satisfactory approach would seem to involve a clearer understanding of the problems involved in the allocation and synchronization of parallel processors. If the hardware structure of the machine can then incorporate features which appear necessary to such a system, then a more efficient structure should result. This is the approach taken by Data Flow.

In a C.mmp like system, the overheads of management are likely to become smaller if the parallel tasks allocated to processors are large. Communication and synchronization will then take place less often. However the allocation of parallel parts of a program at a large section level (often referred to as large scale 'granularity') requires the problem to be specified in this form for the machine to be able to make any attempt at efficient task allocation. This suggests that the programmer must have a knowledge of the machine structure in

order to optimize the specification. A far more flexible system would result if the machine were capable of exploiting parallelism at a much lower level of granularity. A parallel machine design should therefore attempt to operate at a low level, ideally the single instruction level.

The Data Flow Principle

Consider the problem of evaluating a simple sum of products (AxB) + (CxD), this could be written

X := AxB

Y := CxD

Z := X+Y

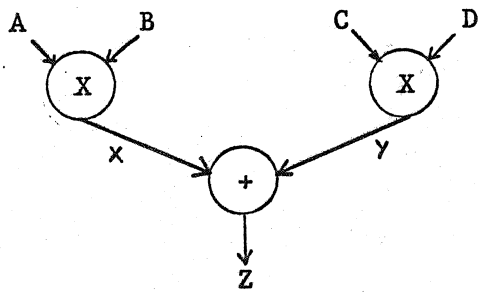
Assume that two or more processors are available each of which can execute an addition or multiplication between two operands. The evaluation of X and Y can be performed in parallel on two processors. When the values of X and Y both become available, the final answer Z can be evaluated. In the general case, any controlling hardware or software must continually scan the results of current operations to find out which results have been produced. When they have been produced it is then necessary to find the instructions which use them and send these instructions to a free processor. It is clear that in any large problem this scanning mechanism would become very complex.

There are two important points to note from this example:-

(i) It is the availability of the data X and Y which dictate whether the add instruction is executable.

(ii) If the instructions producing X and Y had fields which pointed to instructions in which they were used, the requirement for a search would be eliminated.

The principle of Data Flow is based on these simple observations, the formulation of the problem in Data Flow terms is best indicated by a directed graph.



The circles in the diagram are 'nodes' these represent the basic operations. A node is activated when both its inputs become available (NOTE: a node does not have to be a simple instruction, nor does it have to have only two inputs). Clearly each node must contain a pointer to the input of the node to which its output is connected.

The effect of specifying the computation in this way is to achieve automatic synchronization between all operands in the problem. Any node can be executed as soon as its inputs become available although if no processing resources are free, it can be held up without any loss of synchronization. Additionally any nodes which have inputs available and are thus executable can be executed in any order. This property is invaluable in the design of a parallel machine, it allows not only random allocation of instructions to any free processing resources but also the use of multi-stack (and multi-ring - see later) storage techniques.

How can this structure be implemented in hardware?

In order to simplify the following description it will be assumed that:-

(i) Nodes are simple instruction

(ii) They have two inputs and one output.

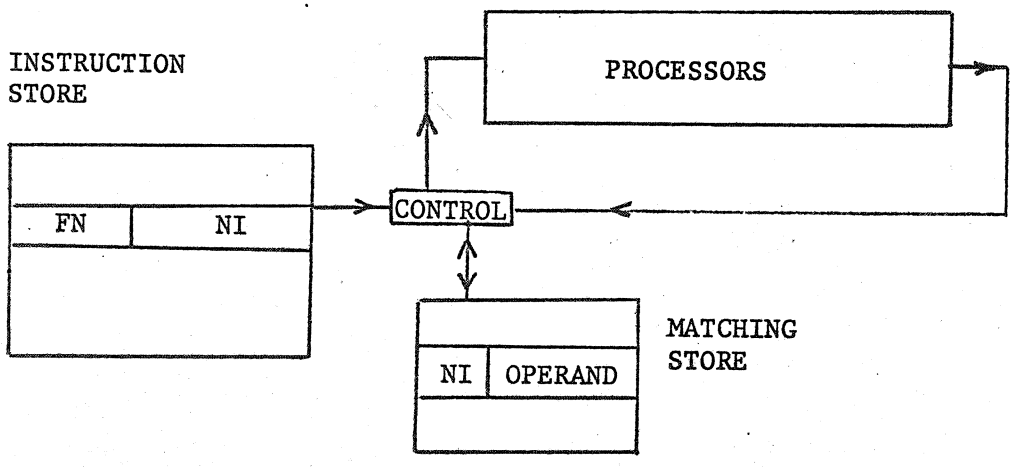
What should exist in the hardware?

(a) Processing elements

(b) A store for node instructions

(c) A matching mechanism

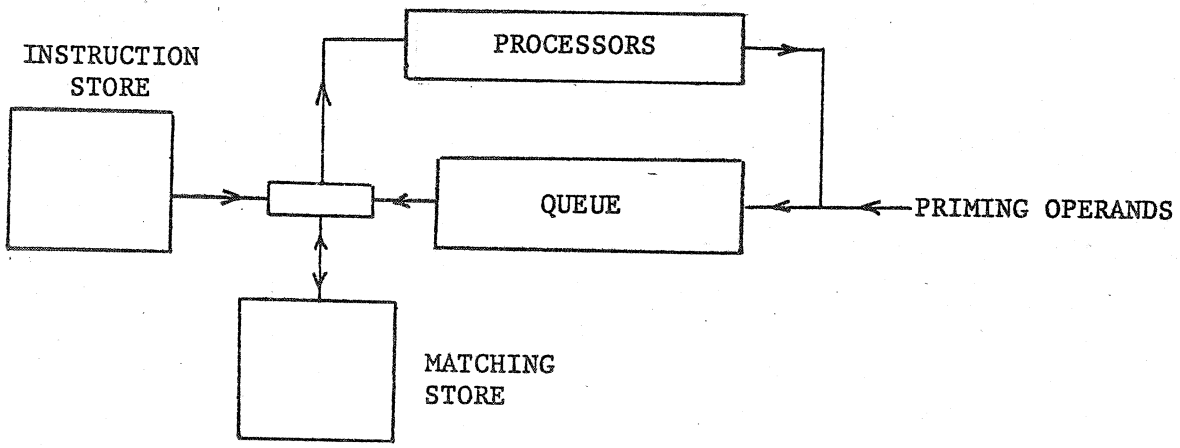
A block diagram could then be drawn:-



This simple structure contains no initial starting mechanism, however the operation is clear. An operand exits from a processor with a pointer to the instruction which will use it next (NI). It is presented to the matching store using this NI as an address. If another operand already resides there, it must be the matching one for the node and the node can be fetched from the instruction store and sent for processing. Otherwise the operand is placed in the matching store to await a partner. A node is sent to the processors complete with function and Next Instruction; the NI must be attached to the result operand when the processor has completed the function.

A Mechanism must exist in the processing elements to ensure that one free processor accepts the instruction and signals to the control that it has done so, this can be arranged by a "daisy chain" system.

The only addition required to produce a workable design is a priming (initial starting) system. This can be implemented as a First In First Out (FIFO) buffer between processors and control:-



The initial operand values can now be placed in this queue with NI values pointing to the first instructions. A more detailed description of a practical design is given in Appendix A1.

Limitations

This design will certainly work but it has severe limitations.

(i) The instructions can only be used to direct one set of operands, if the sum of products of four different numbers were required in the example, it would be necessary to duplicate the code.

(ii) Although the output of a node can be fed back forming a loop it would be impossible to tell which operands belonged to which execution of the loop and the operation could be in error.

(iii) Procedures are impossible for the same reasons as in (i) or (ii).

Solutions

Take the first case of a parallel stream of operands all wishing to use the same set of instructions. These may be elements of an array in a regular array computation. By attaching to the operands of each point of the array, a unique index identifier, the problem can be solved. The addressing of the matching store must now be performed using a concatenation of the Next Instruction and the Index Identifier.

The problems of iteration and recursion can also be solved using identifier fields attached to the operand, although in these cases the identifiers must be capable of being manipulated by the processors. For example a simple recursive procedure mechanism only requires a field which is incremented at each procedure entry and decremented at the procedure exit to ensure that operands at a particular level are uniquely identified.

Further manipulations and tests on the identifier fields are necessary in a practical machine, for example manipulations on an index identifier permit complex array addressing sequences to be implemented.

A Practical Design

In order to produce a workable machine design there are several points concerning the simple model which must be examined.

The processing units, instruction store and result queue can be implemented as described, however the matching store requires further consideration. The size of the identifier field will be sufficiently large that a directly addressed store would be impractical. An associative or pseudo-associative memory is the obvious answer.

The 'single ring' unit described may easily become store limited. If the system were constructed from microprocessors each with an average instruction time of 5μS, it would be difficult to support more than 10 to 12 processors with stores of 200nS access time. A multiple 'ring' structure has therefore been devised. The rings intersect at the FIFO queue which now becomes a queue/distribution mechanism. A detailed description of the multi-ring structure is given in Appendix A2.

Other Considerations

Apart from the problems of constructing hardware, there are of course many other features of Data Flow computation which must be considered. In order to use such a machine it must be possible to specify ones problem in a Data Flow form and have a language with which to communicate this specification to the machine. Both these areas require a considerable amount of research. It appears at present that both problem specification and languages will require a novel approach. This may deter those who believe that the inertia of the present is insurmountable, but if parallelism can provide a method of significantly increasing computing power, to abandon it on those grounds would be unwise.

Performance

In order to assess the possible performance of such a system, a simple simulator has been written which models the multi-ring structure. Several programs including two dimensional solutions of Laplace's Equation, singly recursive and doubly recursive evaluations of factorial and parallel evaluation of series, such as e^x for several value of x, have been run to verify the operation of the system.

In order to estimate the efficiency of the interconnection method, the performance of a single processor on a single ring has been compared with many other shapes and sizes of machine. Although the figures are certainly problem dependant, a fair estimate of the efficiency can be gained by considering the figures obtained for 4 rings each of 8 processors.

On a problem with a total parallelism of 50, these 32 processors completed the computation 23 times faster than a single one, giving an efficiency of over 70%. These figures were sufficiently encouraging to prompt the writing of a more comprehensive simulator which will take account of practical store delays etc., this is under way at present. If these figures can be realized in practice, a very modest machine of 32 microprocessors each with an instruction time of 5μS could achieve an overall rate of 5 MIPS.

It may be premature to speculate on the ultimate performance of such a machine but it is interesting to do so. RAM storage is now available with access times of 50nS, this suggests that a single ring might achieve an instruction time of 100nS. A machine comprising 16 such rings could achieve a rate of 100 MIPS, a worthwhile target with present technology.

Areas of Current Work

This document has only outlined the possibilities of Data Flow computer architecture, there are several large areas where much useful work can be done:-

Specification of problems in Data Flow Form.
Languages for Data Flow Machines
Operating Systems for Data Flow Machines
Input/Output
Storage - Pseudo - Associative Matching Stores,
Backing Storage, Multi level Storage
Overall Architectural improvements

Most of these areas have received some investigative effort and no totally insurmountable problems have been encountered, although it is believed that a deeper understanding of the specification of problems may lead to changes in the architecture.

APPENDIX 1

Details of a 'Single Ring' Architecture

For simplicity the main text of this document has previously assumed that all nodes in a data flow system have one input and two outputs. In practice this is not sufficient for comprehensive problem specification, however the provision of nodes with one or two inputs and one or two outputs is acceptable. This modifies the machine structure and controller function slightly.

The implications of this and a more detailed design of other parts of the machine are discussed with reference to Figure A1.

The diagram is essentially the same as that discussed previously with the following exceptions:

(i) An IDENTIFIER field has been introudced into the operand for use as already described.

(ii) A Content Addressable Memory has been used as the matching store.

(iii) A Simple 'Daisy Chain' instruction allocation mechanism is shown. Each processor has a T(Take) and a B(Busy) line. When the controller has an instruction available it raises the T line, this propogates past all busy processors until the first non-busy processor to see a 'one' on its T line strobes the highway. (A 'handshake' reply is obviously required in case all processors are busy).

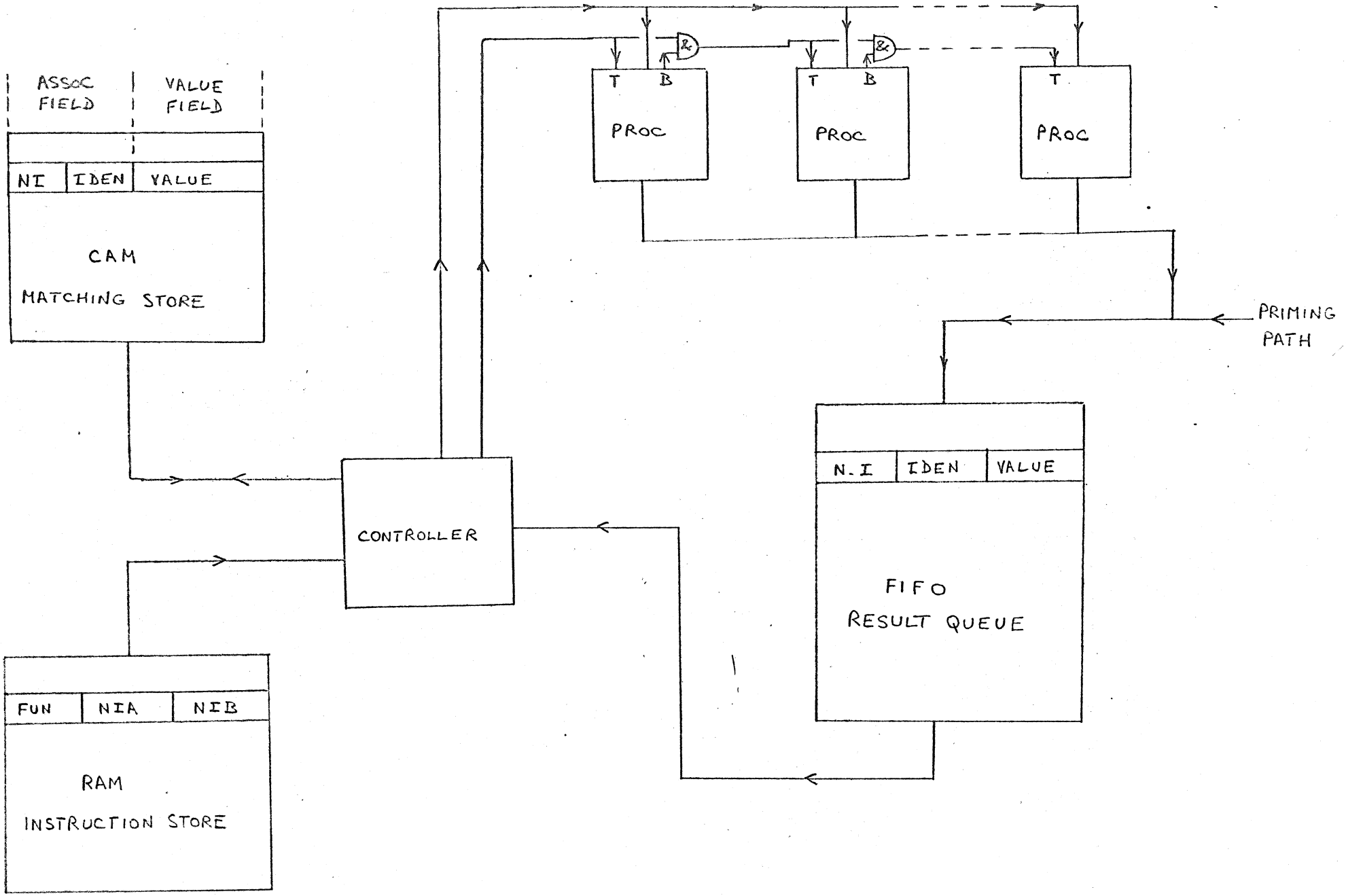


FIGURE A1 A SINGLE RING

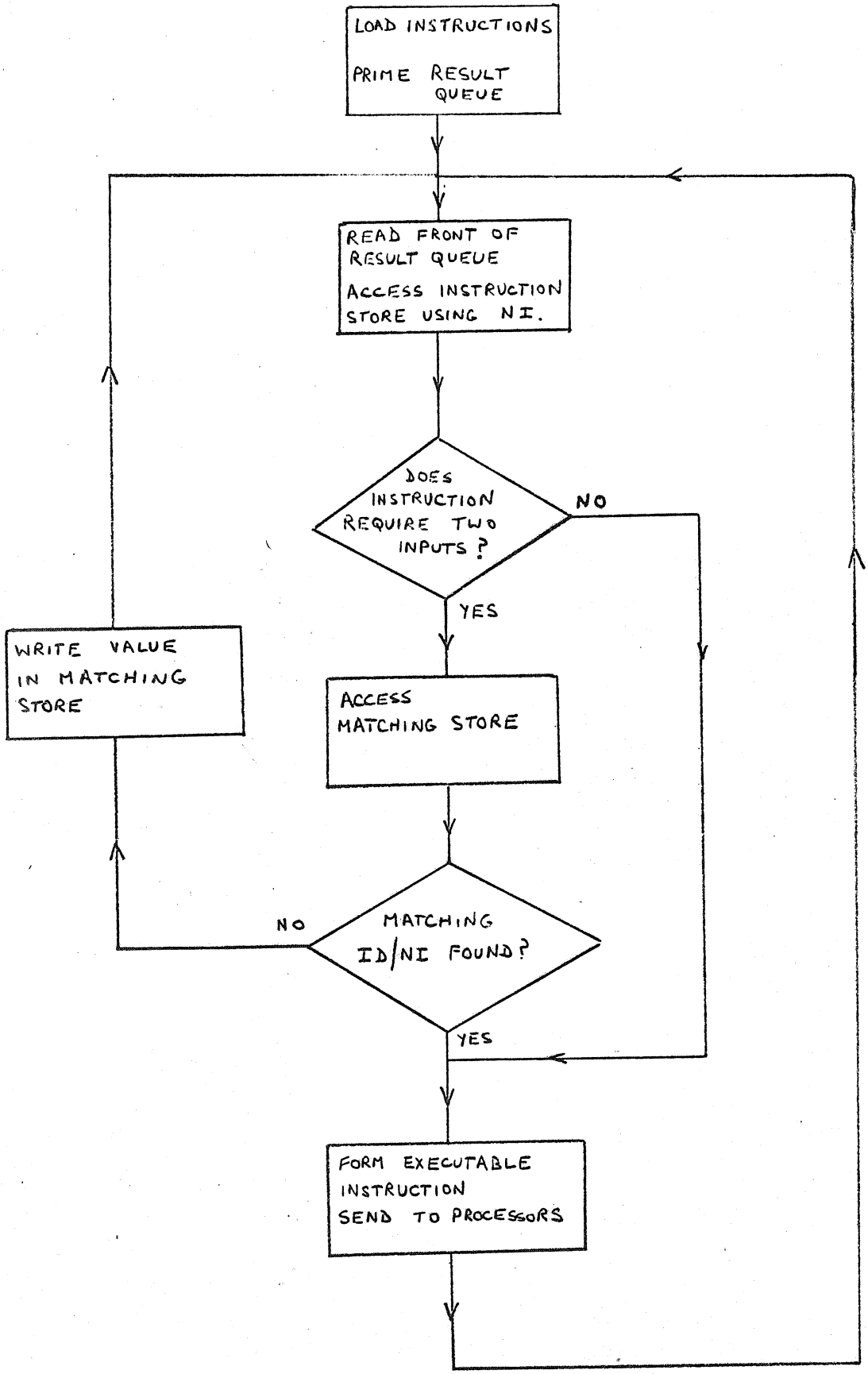


FIGURE A2 CONTROLLER ACTION.

Detailed controller operation is best understood from the flowchart of Figure A2. Meanwhile the processors are producing results and placing them on the end of the result queue (a bus controlling mechanism is obviously required for this).

In order to assess the limitations of the structure, it is necessary to make some assumptions about store/processor speed. Figures of 200nS store read/write time and 5μS average processor instruction time would seem to be representative of present medium speed technolgy. (This assumes that microprogrammed microprocessors are used and that a suitable pseudo-associative store mechanism can be devised).

The controller beat time is dictated by the store accesses. Each time a result is taken from the Result Queue it is necessary to access both instruction and matching stores (this can be done in parallel). If all instructions have two inputs, then on average every second operand will require writing to the matching store. The average operation will then take approximately 300nS, and an executable instruction will be produced every two of these, i.e. every 600nS. For instructions with only one input, an executable instruction would be produced every 200nS. Assuming a value (from simulation experience) of 1:1 for the ratio of two inputs to one input, an average executable instruction time of about 400nS will result. With an average of 5μS for the execution time, it would appear that twelve processors could be supported.

The input and output bus systems must be able to cope with maximum rates which may occur. For the input bus this must be dictated by the minimum executable instruction time, i.e. 200nS.

The output bus rate is related to the minimum processor instruction execution time. If this is assumed to be 2μS and each processor produces two output operands, a bus beat time of 100nS would be necessary. Of course this would rarely happen and as long as a suitable handshake control system existed between input/output bus systems a time of 200nS could probably be assumed for each with little effect on performance.

From these rough figures it would seem that a system could be built with an instruction execution rate of 2 MIPS using conventional medium speed technology.

APPENDIX 2

A 'Multiple Ring' Architecture

As stated in the main text the 'Single Ring' architecture becomes limited by storage speeds. It would be possible in such an architecture to incorporate overlapped stacks of fast 'cache' memory to produce a higher instruction rate, but problems would then occur with distribution on the bus system. From initial investigation it appears that a simple yet satisfactory solution exists by duplicating these 'Single Rings', but with an intersection and switching mechanism incorporated in the result queue. This is shown in Figure A3 for an eight ring system.

Operands will appear at the inputs to the switch from eight sets of processors. It is necessary to ensure that operands with the same identifier - instruction address are directed to the same ring output (and hence controller/matching store/instruction store). This can be achieved using three bits of the identifier - instruction address to control the path through the switch.

At the first level of the switch the first bit is used to control whether the operand is placed in the higher or lower FIFO buffer, the second bit is used at the second level etc. The path of an operand with the chosen three bits of the identifier equal to 001 appearing on input seven is indicated for example.

The mechanism has imposed a restriction on the design that operations are layered and any executable instruction cannot be sent to any free processor. It also assumes that there is a sufficient number of results in the result queues to enable a particular result to filter through the switch before the output channel becomes empty. Simulation has indicated that, with the program examples used so far, although they have some effect, these factors do not reduce the overall system efficiency below acceptable levels.

The switching system used could be replaced by a single 8 x 8 crossbar switch at one of several places in the ring. However such a switch has two significant drawbacks.

(i) Its complexity as a function of the number of processors N, is N^2. This compares with a complexity N $\log_2$ N in the suggested scheme; a significant reduction in a large system.

(ii) If all inputs to an N x N switch have an identifier which directed them to the same output, one output line would need to distribute these at N times the input rate. By using a sequential binary switching system coupled with the existing FIFO queueing buffers, the maximum rate at which the buffers must accept operands is reduced to twice the input rate.

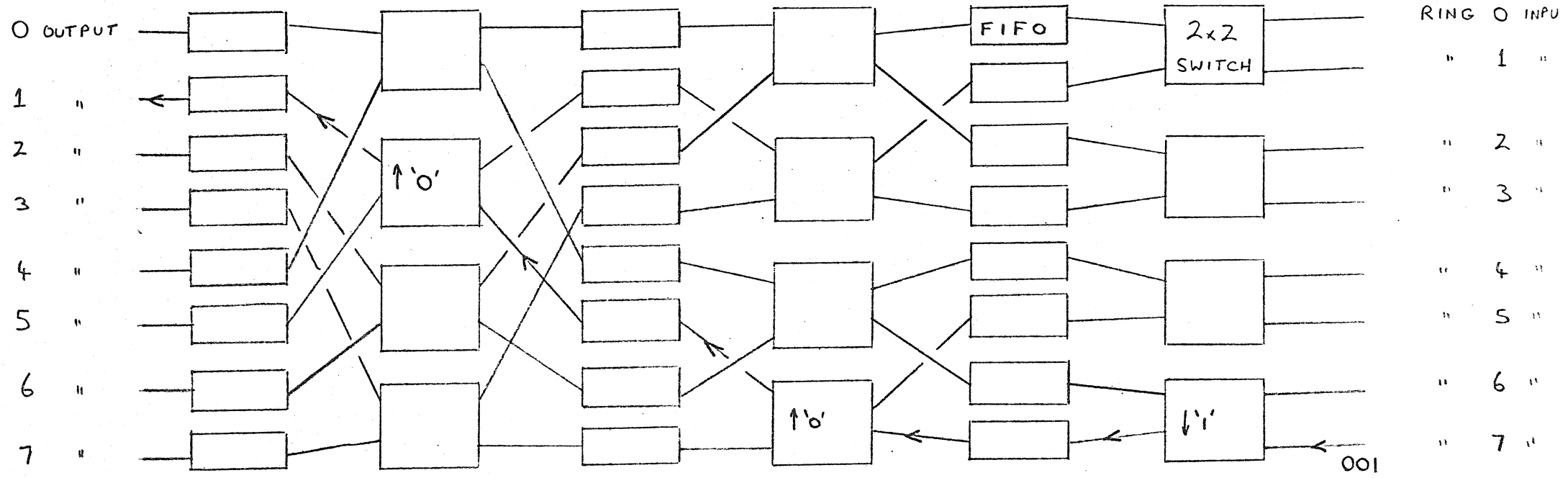


FIGURE A3 A 'MULTI-RING' SEQUENTIAL SWITCH

The number of buffers required between each set of switches is dependant on the randomness of identifiers which occur at the switch input. Initial theoretical probability calculations have shown that three levels will result in over 90% of the total maximum throughput for random input identifiers.

In the model which has been simulated, the ring selection is done purely on the Next Instruction address. In highly parallel array problems it would clearly be better to select on the index identifier although this requires duplication of instructions in the individual ring instruction stores. It is thought that this ring selection should be easily alterable and that an instruction distribution/cache mechanism might be desireable.

This is one possible method of overcoming the speed limitations of a single data flow 'ring'. It is clear that this structure is highly dependant on the ability of data flow instructions to be executed in any order. This seems to be a prime requirement for any single instruction level parallel system and with this in mind, it is probably possible to propose several different variations on the architecture described.

1 Fundamentals of Dataflow

J. R. Gurd

1.1 INTRODUCTION

It is becoming apparent that future requirements for computing speed, system reliability, software manageability and cost-effectiveness will entail the development of alternative computer architectures to replace the traditional 'von Neumann' organization on which our present computing practices are based. Dataflow architecture is one possible alternative which aims for high-speed computing via efficient exploitation of software parallelism in a highly parallel system of processing hardware. The name 'dataflow' is derived from the graphical model of computation which is used to describe how programs are executed. In this model data is active and flows asynchronously through the two-dimensional program, activating each instruction when all the required input data has arrived. This is in direct contrast to the 'von Neumann' model in which data passively resides in store whilst instructions are executed one-at-a-time according to a defined sequence controlled by a 'program counter'.

Dataflow architectures, as described in this part of the book, are only one alternative to traditional computers. Several other models with similar characteristics are emerging, and these are sometimes confused with dataflow systems, usually because they too are driven by their data. In particular, string reduction and graph reduction systems fall into this category. In the following we will concentrate on 'pure' dataflow architectures.

This part of the book is divided into four chapters, covering fundamentals, hardware techniques, machine-level programming and high-level software. This first chapter opens with a discussion of the nature of software parallelism, the possible ways of representing it, and some implications for parallel machine-code design. This provides an introduction to dataflow notation and also demonstrates the important distinction between static and dynamic dataflow systems. The chapter concludes with a discussion of techniques for compiling from various high-level programming languages into dataflow object-code.

In Chapter 2 on hardware we consider the requirements for executing dataflow code and exploiting the exposed software parallelism. We then study three different system designs which have been, or are being, constructed as experimental research vehicles for further work applying and refining dataflow techniques. The chapter closes with a discussion of dataflow system performance.

Chapter 3, on machine-level programming, studies the languages which are used to specify graph programs for the Manchester Dataflow Machine. The lowest-level interface

is via a compact textual representation of the binary messages which are sent to load the program store. This is difficult for humans to use as a programming vehicle, and it is more normal to use the Template Assembler (TASS) which is also described.

Chapter 4 describes a specific high-level language for dataflow programming, SISAL, illustrated by a number of examples of language constructs and some complete programs. SISAL is a single-assignment language with Pascal-like syntax. It is currently being used for evaluation of a variety of multiprocessing strategies.

1.2 PARALLELISM IN SOFTWARE

Two kinds of parallelism can be found in software. The first kind occurs when a common operation (or set of operations) is to be applied to many separate sets of data. An example is the element-wise addition of several arrays, as in the Fortran program:

```
      DO 10 I = 1,100
      F(I) = A(I) + B(I) + C(I) + D(I)
10    CONTINUE
```

The second kind is found when different operations (or sets of operations) are to be applied to separate (or even common) sets of data. This may be found in many blocks of assignment statements, for example, the following Fortran code:

```
A = E - G
B = H * Z
C = E * H + F
D = E + G
```

These forms of parallelism have been known for a long time and their importance in influencing parallel hardware design has been recognized. Flynn [1] classified hardware systems as SIMD (single-instruction-stream, multiple-data-stream) if they exploit the first kind of software parallelism, and MIMD (multiple-instruction-stream, multiple-data-stream) if they exploit the second kind.

Nowadays this classification is considered overly simple, but no generally accepted alternative taxonomy is emerging. The difficulty seems to be that parallel hardware may be deployed at a different level of 'granularity' to the obvious software parallelism. For example, in an instruction pipeline, small parts of the execution of successive instructions are processed concurrently by overlapping, regardless of any program parallelism at the instruction level, or above. In the absence of a level-independent taxonomy of parallel systems comparison of different architectures is by *ad hoc* methods. We have found it useful to distinguish between 'regular' and 'irregular' parallelism when comparing the abilities of dataflow systems with those of more conventional parallel systems.

Regular parallelism exists wherever the same task is to be performed many times over, usually on disjoint data. With connected data it may be necessary to exploit regular parallelism via a pipeline, as in the instruction pipeline cited above. With unconnected data, as in the case of the first (SIMD) kind of software parallelism, a lock-step parallel array of hardware can be used, as in the DAP [2] or ILLIAC IV [3]. In either case, the actions to be performed concurrently are highly regular, and the performance of the systems depends critically on whether or not the program can provide sufficient work with the required amount of the required form of regularity.

Most of the parallel computers so far constructed exploit regular parallelism of one form or another. In practice it has proved surprisingly difficult to arrange for programs to provide, continuously, sufficient parallelism of the desired nature. Consequently applications run at variable speed, the regular parts executing rapidly, whilst other sections

are necessarily slower. In many cases the slow segments dominate overall performance and reduce the total speedup of programs to a small fraction of that intended.

Irregular parallelism exists wherever different tasks are potentially concurrently executable, sometimes on common data. This corresponds to the second (MIMD) form of software parallelism. An independent array of parallel hardware, such as in the CDC 6600 [4] (on a small scale) or the C.mmp [5] and Cm* [6] multiprocessors (on a large scale), is needed for implementation. Where common data is involved complex interlocking mechanisms are necessary to prevent unintentional accesses being made (e.g. reading data before it has been defined, or writing before all prior reads have been completed). Note that hardware mechanisms which exploit irregular parallelism will also be able to handle regular parallelism. The reverse is not usually the case.

Few systems have been constructed to exploit irregular parallelism on a large scale, and it is in this area that many interesting experiments in computer architecture are now being conducted. The best known examples use parallelism at the 'process' level, derived from programming languages such as Concurrent Pascal [7], Modula [8], Distributed Processes [9], and Communicating Sequential Processes [10] and implemented on shared-memory or message-passing multiprocessors. Dataflow systems exploit irregular parallelism at a lower level, approximating to the conventional machine-code instruction-level.

Whether parallelism is regular or not, the key issue in developing a system to exploit it is to provide an effective notation for expressing potential parallelism in programs. In the following section we develop a notation for instruction-level irregular parallelism by examining the nature of inherent parallelism in a small segment of conventional Fortran code.

1.3 PROGRAMS AS GRAPHS

Consider the following set of Fortran assignments which multiply together the 'variables' I1, I2, I3, I4, I5 and I6 and put the result in 'variable' K:

```
L = I1 * I2
M = I3 * I4
N = I5 * I6
K = L * M * N
```

To discover the potential software parallelism we must discard the traditional view of a program as a list of instructions which manipulate data in fixed storage locations in a defined sequence. Instead we need to concentrate on the role the individual storage locations play as they temporarily hold data values whilst the latter pass between operations in the program. The pattern of store accesses brought about by the sequence of activation of instructions is normally contrived by the programmer to achieve the combinations of data with operators dictated by the particular problem being solved. The fact that this is specified as a one-at-a-time process owes more to the history of the development of computers than to inherent constraints in the problems that computers are used to solve.

1.3.1 Data Dependence Graphs

An alternative view of the combination of data with operators is obtained by constructing a data dependence graph for the program [11, 12]. Algorithms for this task are in common use for conventional machines in optimizing compilers. In the example above, we simply draw a number of arcs over the program, one arc for each variable. The tail of an arc shows where the variable is assigned, and the head shows where the variable is

consumed (by appearing on the right-hand side of an assignment statement). In more complex examples more than one arc may be required for a variable when it appears on the right-hand side of more than one assignment statement. Multiple assignments, where a variable is assigned a value at more than one point in the program, can be dealt with by systematically renaming the variables so that a version is created without multiple assignments, but with the same meaning as the original. Where variables appear only on the right-hand side they are assumed to be input data to the program segment. The resultant graph for our example is shown in **Figure 1-1**.

This diagram is more visually attractive if it is rearranged to show enforced sequence down the page, with potential concurrency across the page, as shown in **Figure 1-2**.

In this graphical form it is possible to omit all the variable names as they are now superfluous, being constrained to be the same at head and tail of each arc. If names are required (as an aid to understanding, or for writing a textual version of the graph), they can be written just once, alongside the appropriate arc. Each assignment statement can be simplified to a description of the expression to be computed. In many cases this will be a simple arithmetic operation, as in the case of the multiplication in our example, shown in **Figure 1-3**.

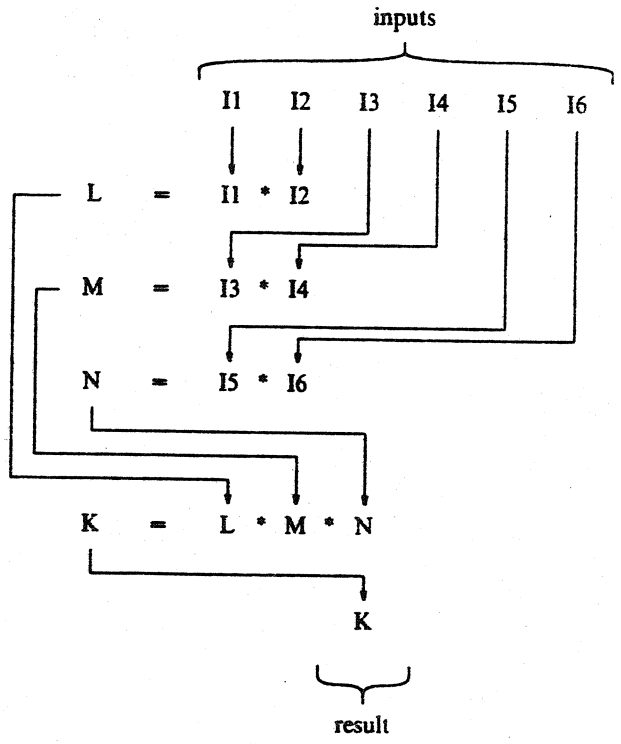


Figure 1-1

18. J. B. Dennis, "First Version of a Dataflow Procedure Language," *Lecture Notes in Computer Science* 5, p.187 (1974).

19. D. P. Misunas, "Structure Processing in a Dataflow Computer," *Proceedings of Sagamore Conference on Parallel Computation* (August 1975).

20. Bowen D.L, "Implementation of Data Structures in a Dataflow Computer," Ph.D. Thesis, Department of Computer Science, University of Manchester (May 1981).

21. Arvind and R. A. Ianucci, "A Critique of Multiprocessing von Neumann Style," Technical Memo 226, MIT Laboratory for Computer Science (1983).

22. W. B. Ackerman and J. B. Dennis, "VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual," Technical Report 218, MIT Laboratory for Computer Science (June 1979).

23. W. B. Ackerman, "Dataflow Languages," *IEEE Computer* **15**(2), p.15 (February 1982).

24. J. R. W. Glauert, "High Level Languages for Dataflow Computers," *Pergamon-Infotech State of the Art Report on Programming Technology*, p.173 (March 1982).

25. J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. R. W. Glauert, I. Dobes, and P. Hohensee, "SISAL - Streams and Iteration in a Single-Assignment Language," Language Reference Manual Version 1.0, Lawrence Livermore National Laboratory (July 1983).

26. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM* **21**(8), p.613 (August 1978).

27. G. Richmond, "A Dataflow Implementation of SASL," M.Sc. Thesis, Department of Computer Science, University of Manchester (October 1982).

28. J. Darlington and M. Reeve, "ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Languages and Computer Architecture* (October 1981).

1.7 REFERENCES

1. M. J. Flynn, "Some Computer Organisations and Their Effectiveness," *IEEE Transactions on Computers* C-21(9), p.948 (September 1972).
2. S. F. Reddaway, "DAP - A Distributed Array Processor," *Proceedings of 1st ACM Symposium on Computer Architecture* (December 1973).
3. G. H. Barnes et. al., "The ILLIAC IV Computer," *IEEE Transactions on Computers* C-17(8), p.746 (August 1968).
4. J. E. Thornton, *Design of a Computer: The CDC6600,* Scott Foresman & Co (1970).
5. W. A. Wulf and C. G. Bell, "C.mmp - A multi-mini-processor," *Proceedings of AFIPS FJCC* 41, p.765 (September 1972).
6. R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* - A Modular Multimicroprocessor," *Proceedings of AFIPS NCC* 46, p.637 (June 1977).
7. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* SE-1(2) (June 1975).
8. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software - Practice & Experience* 7(1), p.3 (January 1977).
9. P. Brinch Hansen, "Distributed Processes : A Concurrent Programming Concept," *Communications of the ACM* 21(11), p.934 (November 1978).
10. C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* 21(8), p.666 (August 1978).
11. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Communications of the ACM* 19(3), p.137 (March 1976).
12. D. J. Kuck et. al., "Dependence Graphs and Compiler Optimisations," *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, p.207 (January 1981).
13. J. L. Peterson, "Petri Nets," *ACM Computing Surveys* 9(3), p.223 (September 1977).
14. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," *Proceedings of 2nd IEEE Symposium on Computer Architecture*, p.126 (January 1975).
15. G. S. Miranker, "Implementation of Procedures on a Class of Dataflow Processors," *Proceedings of International Conference on Parallel Processing*, p.77 (August 1977).
16. Arvind, K. P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Technical Report TR114a, Department of Information and Computer Science, University of California at Irvine (December 1978).
17. I. Watson and J. R. Gurd, "A Prototype Data Flow Computer with Token Labelling," *Proceedings of AFIPS NCC* 48, p.623 (June 1979).

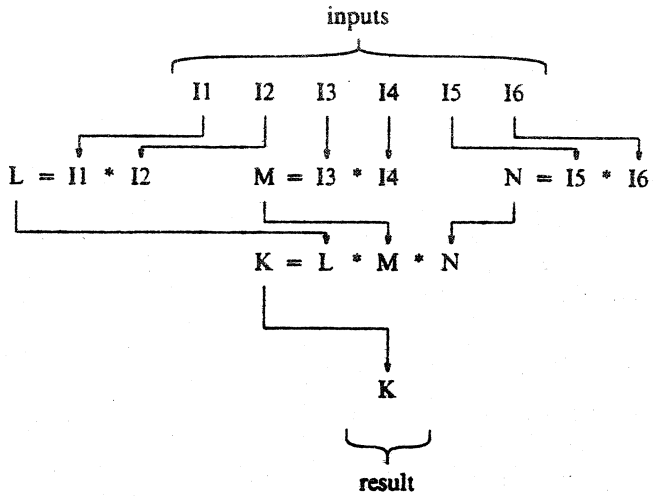


Figure 1-2

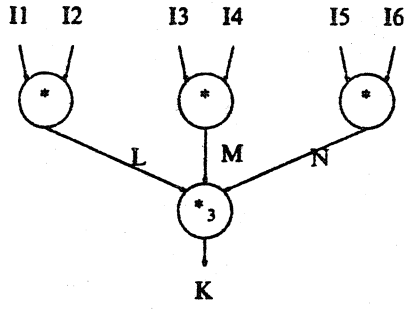


Figure 1-3

1.3.2 Machine-Level Graph Programs

We have now constructed a simple statement-level data dependence graph. Note that it retains the meaning of the original program, but it also shows potential parallelism and enforced sequence in a two-dimensional format. In order to illustrate all the program parallelism available for exploitation by instruction-level parallel hardware it is necessary to decompose the program even further. Of course the level to which we descend is completely arbitrary. We could build a system capable of multiplying three values together in one instruction (in which case the above graph would not need further reduction), or we could go to the extreme of implementing only boolean operators (AND, OR, NOT, etc.) in hardware, and building up more complex operators using standard techniques (in which case our example graph would require considerable further decomposition). Most of the dataflow computers currently under construction use an instruction-level comparable to that of a 16-bit minicomputer with extended arithmetic capabilities. We shall assume this level in the remainder of this part of the book. This implies the availability of straightforward monadic and dyadic arithmetic operators on integer and floating-point numbers, and we will also assume the existence of operators which generate and

manipulate boolean values.

In our example program it will be noted that the lowest expression evaluation in the graph is not a machine instruction at this level. Consequently it must be implemented by a subgraph of instructions such as either of those shown in **Figure 1-4**.

In this particular example it is immaterial which of these alternatives is used, and a compiler could choose between them arbitrarily. In other cases there will be efficient and inefficient options and compilers will need to be sensitive to the assessment criteria if they are to produce optimal code under a wide range of conditions. To develop such assessment criteria we need to know how programs will actually execute on a specific parallel hardware configuration. This is too difficult to discuss in detail here, but we shall finish this chapter with a brief description of an abstract dataflow implementation model from which the basic principles of execution may be derived.

1.3.3 Execution of Machine-Level Graphs

Consider a complete machine-level program graph for our example in which each multiply instruction is given an identification number, as shown in **Figure 1-5**. Remember that the purpose of this notation is to allow all potentially concurrent instructions to execute simultaneously. In the original sequential program we would expect the multiplications to be performed in the order {1}, {2}, {3}, {4}, {5}, producing the answer in five multiplication times. On the graph above we can see that either of the parallel execution orderings {1, 2, 3}, {4}, {5} or {1, 2}, {3, 4}, {5} will produce the answer in three multiplication times (given at least three and two multipliers, respectively). The problem for the parallel execution model is to cause one of these parallel execution orderings to be followed.

It is difficult to arrange activation of instructions by some parallel equivalent of a program counter. In the first place such program counters would have to be associated with processors, and the variable amounts of parallelism that could occur might require large numbers of these processors, many of which could frequently become idle. Secondly, the idea of a program counter is closely linked to the concept of a linear data store with fixed locations for each program variable. Large numbers of active instructions would

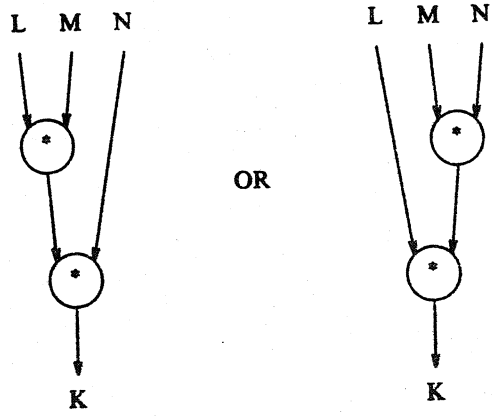


Figure 1-4

1.5.2 Single-Assignment Languages

Single-assignment languages (SALs) have no concept of sequential execution and no direct control statements such as the GOTO. To combat the ambiguities that might arise from reassigning values to variables, the languages allow each variable to be assigned just once in a program. Constructs which permit controlled reassignment in special cases, such as loops, are provided. SALs tend to use data structures, such as arrays and streams, that can be readily implemented in dataflow graphs. There are often strict type and scope rules. In particular, it is common to prohibit all forms of side-effect in reentrant constructs. The net results are languages that provide ideal textual syntax for the description of dataflow graphs [22, 23, 16, 24, 25].

Many SALs were developed without reference to dataflow execution, and they are similar to the *functional* or *applicative* languages which have been developed without reference to any particular means of execution.

1.5.3 Functional Languages

Functional languages are based on the mathematics of functional algebra and have no concepts of storage state and assignment [26]. They are sometimes referred to as *zero-assignment* languages. In fact, if assignment is restricted to occur only once for each variable in a program, the effect is the same as if there were no assignment at all and 'assignment' statements were treated as *definitions* of the variables. In this sense SALs and functional languages are identical and it should come as no surprise to find that absence of GOTOs and side-effects are common to them both. However, functional algebra allows more powerful programming constructs than are used in SALs because it permits construction of higher order functions and abstract data structures. Consequently the two groups are not directly equivalent. Nevertheless they have enough in common to make it attractive to implement functional languages on dataflow systems.

Several attempts have been made to compile dataflow code from higher order functional languages [27]. These indicate that it is possible to implement such languages fully, but there are many doubts as to the efficiency of programs produced in this way. Recent research has concentrated on developing mixed data-driven/demand-driven architectures for such languages [28].

1.6 SUMMARY OF DATAFLOW GRAPHS

Dataflow graphs are a convenient notation for representing parallel computations. They permit conditional constructs, loops, functions (including recursion), and data structuring. Translation to dataflow graphs is feasible from a wide range of high-level programming languages.

There is a natural classification for dataflow systems according to the way they handle reentrant code. The three classes of system are known as *static, dynamic code-copying,* and *dynamic tagged* schemes.

entry to, and to restore old tags at exit from, the reentrant code. Of course, tokens must carry extra bits to denote the tag.

Note that token-tagging can be used to distinguish data belonging to different cycles around a loop. For example, in the program shown in **Figure 1-11**, assuming all input tags have value zero, the tags could be incremented each time round the loop at the points labelled '+ +', and zeroed on exit from the loop at the point labelled 'ZZ'. In this case it is no longer necessary for the arcs to act as first-in-first-out queues, and the 'firing rule' can be derestricted.

Systems using the above schemes to implement concurrently reentrant functions are known as *dynamic* dataflow systems. The first scheme is called the *dynamic code-copying* scheme. The second scheme is known as the *dynamic tagged*, or *dynamic code-sharing* scheme. Hybrid dynamic systems use both code-sharing and code-copying in order to limit the size of the tag.

1.4.4 Structured Data

Compact programs can also be written using data structuring, by which a single variable name is used to refer to a large collection of simple data items. Two schemes have been developed to implement data structures in dataflow graph programs.

The first scheme uses separate storage to hold the structures and represents each structure travelling in the program graph by a *pointer* token. A specialized *structure store* is responsible for executing read and write operations on structures, and also for issuing the appropriate pointers [19]. All other instructions are as described above, and operate on scalar data, or control the flow of pointer tokens through the program graph.

An alternative scheme uses the tagging system described in the previous section [20]. Each element of a data structure is a normal token which carries a unique tag defining the position of the element in the structure. Tag-sensitive instructions are used to manipulate the structure in the required way. This scheme is particularly useful for implementing regular structures, such as arrays, whose elements are all subject to continuous processing (as, for example, in signal processing applications).

1.5 COMPILATION OF GRAPH CODE

The examples introduced earlier demonstrate that it is possible to generate dataflow graphs from a conventional high-level programming language such as Fortran. However, the analysis algorithm that forms data dependence graphs from such languages is highly complex and takes a long time to execute. There exist other languages which are easier to translate and these are receiving the majority of attention in dataflow research projects.

1.5.1 Conventional Languages

The principal difficulties with conventional languages reside in possible side-effects due to *explicit* use of storage locations (accessed by the programmer as 'variables'). Data dependence analysis is often hampered by obscure array index expressions which are impossible to analyse at compile-time and thus requires some assistance from the programmer to indicate how the arrays will be accessed. However, the worst problem is that of aliasing via the use of unbounded arrays or arithmetic operations on pointers. No amount of compile-time analysis can help unravel devious or undisciplined use of such language 'features'. The only method of control is to ban the facilities from the language [21].

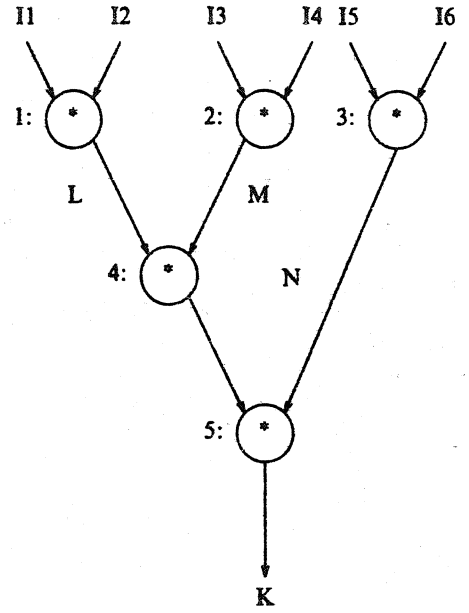


Figure 1-5

imply large numbers of active store locations with attendant problems of multiplexing the required accesses. In addition to this each horizontal 'band' of instructions would have to be synchronized so that the next lower band could not start processing until all current instructions had terminated. This implies that a program would proceed at the speed of the slowest operation in each band. Apart from these problems, the task of allocating instructions to processors would be extremely difficult.

These arguments constitute a compelling reason for abandoning program counters in instruction-level parallel computers. The key to making this transition is to notice that a data dependence graph shows how *instructions are dependent on data.* It is not sensible to execute an instruction before all the data it requires is available. Conversely, once an instruction has finished executing, all other instructions that are waiting for its output data can be activated safely. The simplest way to execute a graph program so as to obey these rules is to send data directly from instruction to instruction according to the data dependence arcs, and to allow each instruction to execute when and only when it has all its required input data available. In this way the graph program execution will be *data-driven.*

We can illustrate data-driven execution of graph programs by introducing data-carriers, known as 'tokens' after Petri-net notation [13], onto the data dependence graph. Each token carries one data value. A token is constrained to move (at any speed it can) from the tail to the head of one data dependence arc. Tokens wait at the heads of their dependence arcs until all other arcs (if there are others) pointing to the same instruction also have tokens at their heads. At this time this instruction can be executed, taking an arbitrary amount of time to complete, after which its result token(s) is(are) placed on its output arc(s). The tokens causing the execution are no longer needed, and so they will be removed from their (input) arcs.

The sequence of 'snapshots' in **Figure 1-6** shows how our example program could be used to evaluate 6! by sending tokens with integer values 1 to 6 to the program inputs I1 to I6, respectively. Tokens are shown on the dependence arcs as black discs with the associated values written alongside. The way in which the data appears to flow through the program graph during execution is the reason for the name 'dataflow'.

1.4 GENERALIZED DATAFLOW GRAPHS

The multiplication program considered above is not a general example of conventional computing practice. The only arithmetic operation used is multiplication and there are no control structures, such as conditionals or loops. In this section we consider enhancements to the dataflow notation which help to accommodate more general programs.

The first point to be made is that *any* form of machine instruction can be represented by a node in a dataflow graph and could therefore be executed in parallel with other instructions. This property makes the graph notation useful for exploiting irregular software parallelism. The simplest case in which this is advantageous is in the evaluation of general arithmetic expressions in which any arithmetic machine instructions could be

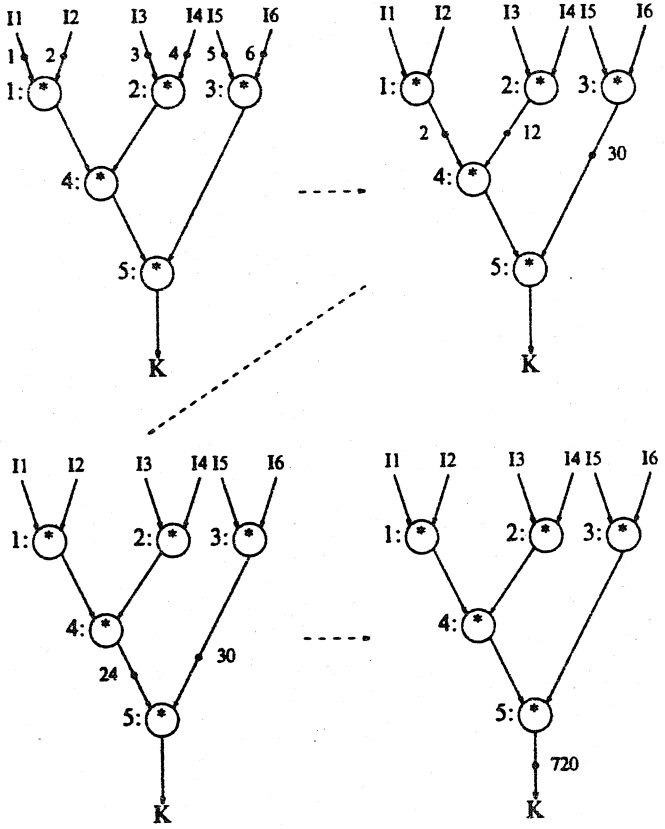


Figure 1-6

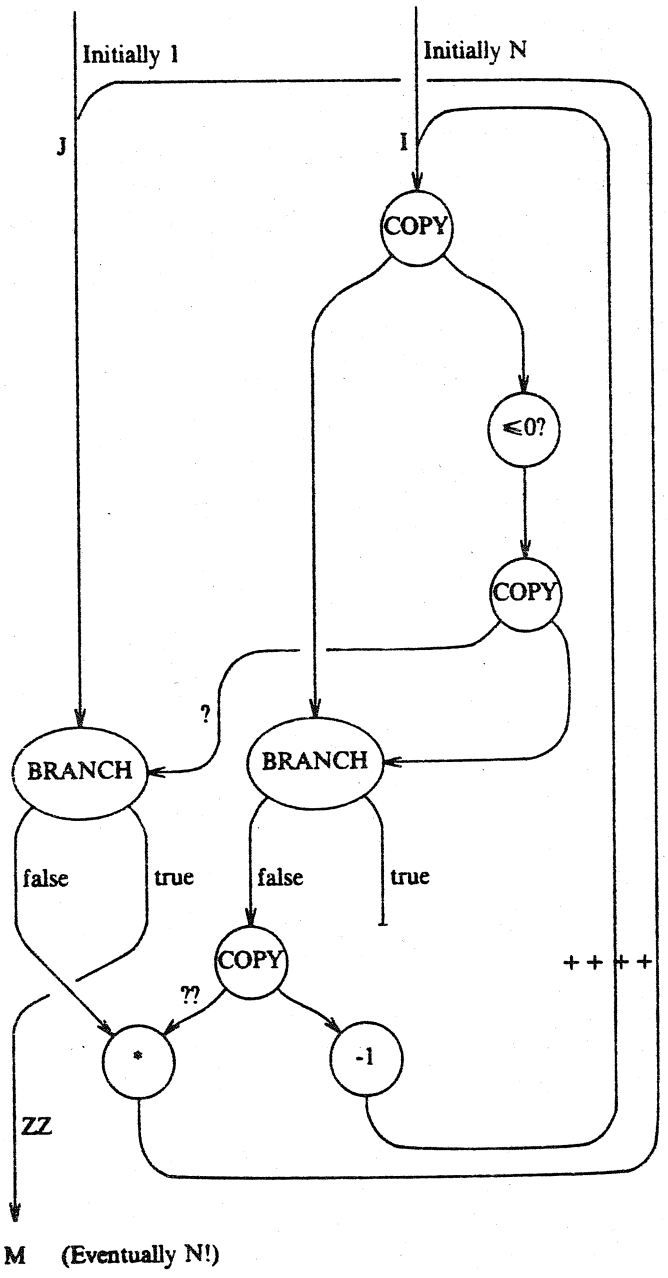


Figure 1-11

currently active data space. In visual terms tagging can be thought of as the process of *colouring* the data tokens [18]. The graph execution rules need to be modified so that only tokens of the same colour (i.e. those carrying identical tags) can group together to cause execution of an instruction. Special instructions are needed to create new tags at

As an example, consider the Fortran program segment below:

```
      I = N
      J = 1
10    IF (I .LE. 0) GOTO 99
      J = I * J
      I = I - 1
      GOTO 10
99    M = J
```

This is an iterative program which computes values of N! for variable N (i.e. not just 6!). It translates into the machine-level graph of **Figure 1-11**. Detailed analysis of this graph reveals that it is possible for more than one token to occupy the arcs labelled '?' and '??'. Consequently, it is essential that the arcs of the graph behave as first-in-first-out queues (otherwise the loop could terminate early because of overtaking on the arc labelled '?'). Unfortunately implementation of unbounded queues proves to be difficult, so it is usual in practical dataflow systems to restrict the normal 'firing rule' so that instructions can only be executed when their output arc is empty.

This is the simplest way of implementing reentrant graph programs, but it is not completely general because it prohibits concurrent reentrancy. It only permits loops which are reactivated in strict sequence. Although a limited amount of parallelism can be obtained by pipelining within the cycles of a loop, there is often further parallelism which can only be extracted by a more general scheme (as described in the next two sections). Systems which implement this first scheme, allowing only sequential, cyclic reentrancy, are known as *static* dataflow systems [14].

1.4.3 Functions

A typical case in which concurrent reentrancy is required is when the programmer defines a *function* (i.e. a user-defined subgraph) which is to be called from several places within the program. This is somewhat similar to a Fortran subroutine. It is, of course, possible to create many copies of the machine code representing the function and to plant them 'in-line' at the appropriate places. However, this is wasteful of instruction storage for large functions and those which are called frequently. It also prohibits the use of recursion since this implies provision of infinitely expanded program graphs. Consequently, two alternative implementation schemes for reentrant programs have been proposed.

The first such scheme permits concurrent reentrancy via an *apply* instruction, planted at the start of a user-defined subgraph, which creates a new copy of the subgraph each time it is activated [15]. All input tokens to a subgraph activation are gathered together at the *apply* instruction and are then transferred to the unique new copy of the reentrant code. An *exit* instruction, placed at the end of the copy of the subgraph, gathers together all the output tokens for the activation and transfers them back to the output arcs of the appropriate *apply* instruction. The copy of the reentrant code is then destroyed. The operation of this scheme is analogous to conventional macro-expansion in that extra code and data space is allocated whenever it is called for. This avoids data having to share code concurrently.

An alternative scheme allows data to share code by 'tagging' tokens as they enter into and exit from the reentrant areas [16, 17]. This system is similar to the use of a stack for implementing procedures and functions on conventional machines, except that the concurrent activation of shared graph code requires that each token be *individually* tagged with the appropriate 'name-base' instead of using global stack registers to identify the

used. Such expressions can be converted easily into graphs. In fact most conventional compilers already generate 'expression evaluation trees', when parsing high level programs, before they generate the required linear object code. The dataflow execution model demonstrates how such trees may be evaluated directly, in time proportional to their height, using parallel instruction execution. At a higher level, the model also allows whole expressions to be evaluated concurrently. Additional parallelism can be found when control structures are invoked.

1.4.1 Conditionals

The simplest control structure is the conditional (**if** ... **then** ... **else** ... **fi**). We can construct a data dependence graph for a conditional statement using *conditional dependence arcs* which are controlled by the runtime evaluation of a boolean predicate. These arcs are implemented using the two 'switching' machine instructions, known as *branch* and *merge*, shown in **Figure 1-7** and **Figure 1-8**.

These may be visualized as two-way switches inserted into the arcs of a standard dependence graph. Each switch selects one of two possible routes for an incoming data token, the other route being left inactive. The route is selected according to the value of

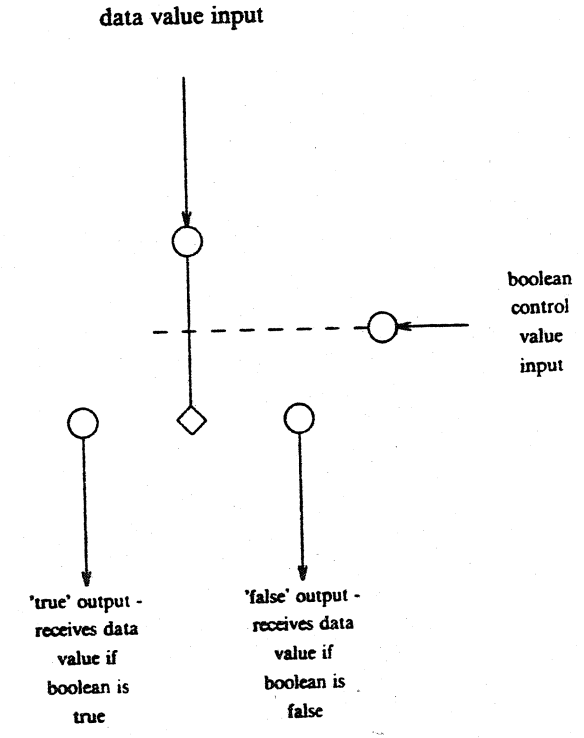


Figure 1-7

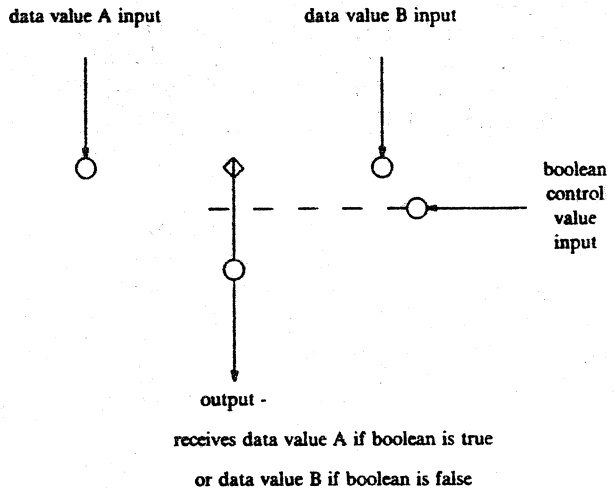


Figure 1-8

a boolean control token. The data and control tokens wait for each other at the inputs to the switch exactly as they would at a dyadic or triadic arithmetic instruction. Where it is certain that only one of the data inputs to a *merge* instruction will be generated, and in proper correspondence to the associated boolean (e.g. from a previous *branch* instruction using the same control value), the *merge* may be omitted from the machine code and the two data arcs conjoined, as shown in **Figure 1-9**.

Using the extended instruction set we can implement a conditional Fortran statement such as:

```
C = A
IF (I .EQ. J) C = F
```

by the graph shown in **Figure 1-10** in which '⊥' indicates that tokens travelling down this arc will be destroyed, and the '=?' instruction generates a boolean value indicating whether its two data inputs are equal.

1.4.2 Loops

Switch instructions are most powerful when used to implement graphical loops and functions. These are important because they allow complex computations to be defined by relatively small programs, in the same way as conventional loops, subroutines or procedures. However, these reentrant constructs pose substantial implementation problems in a parallel computer because of the possibility of simultaneous activation of the reentrant code.

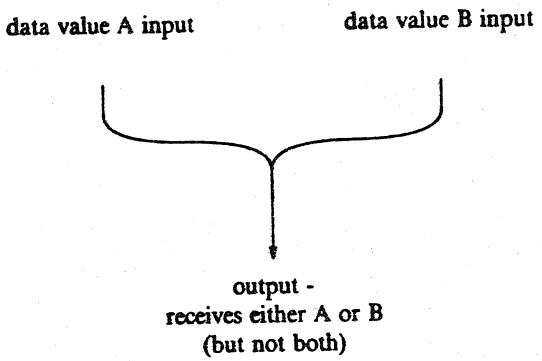


Figure 1-9

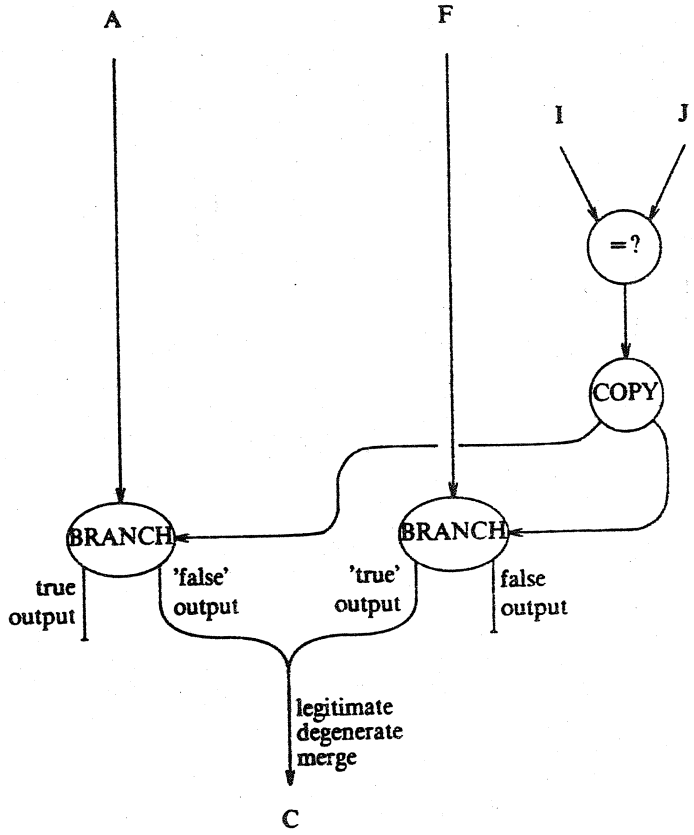
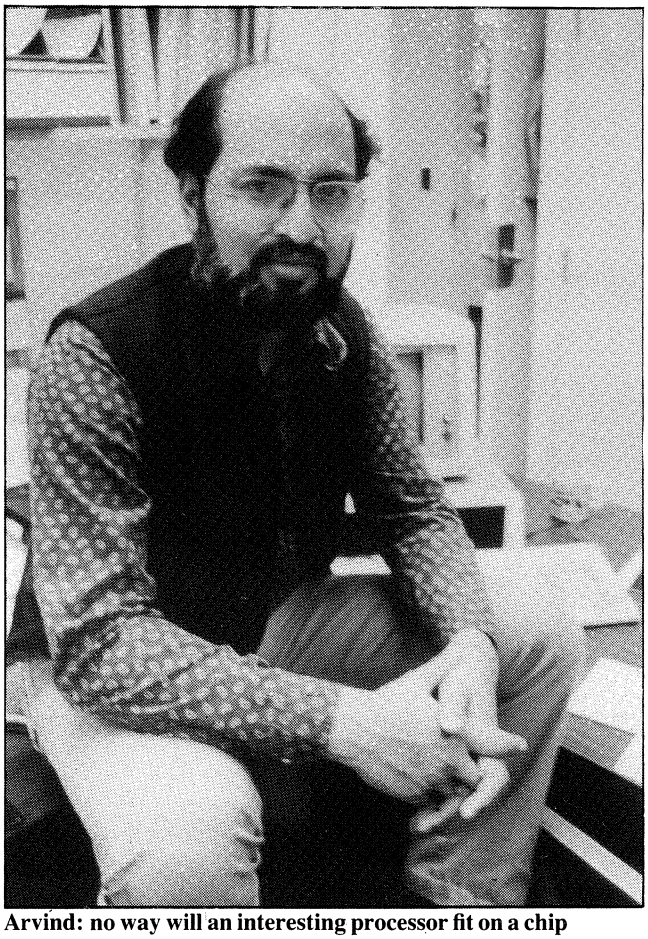


Figure 1-10

The future will run along parallel lines

The quest for faster machines is concentrated on parallel architecture, with a number of projects under way to develop a dataflow machine. Tony Durham looks at the work of an MIT team in this area



Arvind: no way will an interesting processor fit on a chip

Why are computers so slow? Because, runs an oft-rehearsed argument, the conventional von Neumann machine treats a programme as a sequence of operations to be performed one after another.

Programs would run faster if the machine could carry out many operations in parallel. But two things prevent this. One limitation is imposed by the physical resources of the conventional machine. It only has one processor. Its memory can only handle one request at a time. In principle at least, these bottlenecks could be eliminated by expanding the machine's resources.

Another, more fundamental, speed limit remains. Assume, for the moment, that the program is still seen as a body of operations to be applied to data. Each operation has inputs which may themselves be the outputs from other operations. Clearly those other operations must be completed first. They in turn must wait for the operations which compute their own inputs, and so on.

In the dataflow machine architecture, attention is focused on these data dependences. In fact they largely govern the machine's behaviour. The programmer does not have to tell the machine the order in which to execute instructions.

In a dataflow machine, an operation may be executed as soon as its inputs are ready. The precise order in which instructions are executed may depend on physical details but the result of the computation will not be affected.

Dataflow machines are being developed in a number of centres, including Manchester and Toulouse. One of the most ambitious dataflow projects is led by Arvind, a professor in the Laboratory for Computer Science at the Massachusetts Institute of Technology (MIT).

The project has attracted $6.7 million funding from the Defence Advanced Research Projects Agency.

It involves a dozen graduate students, many undergraduates, four IBM engineers, and one visitor from Ericsson, the Swedish-based telecommunications company.

Arvind's parallel machine will have 64 or 128 processors, each of which is a powerful computer itself.

Arvind says he gave up the idea of designing a special processor chip three or four years ago. In his opinion 'there is no way that an interesting processor will fit on a chip'.

He looked at the possibility of designing a dataflow processor board, using commercial chips, as a building block. He decided this would involved too much work.

Instead, Arvind's group has decided to buy ready-made, powerful processors which run the Lisp language. Their own efforts can then be concentrated on the problem of connecting the processors together. Software and microcode will be used to make the Lisp machines behave as dataflow processors.

A big advantage is that complete programming environments already exist for such machines. The $80,000 plus Symbolics 3600 was Arvind's original choice, but now he says his group is thinking of using the recently unveiled Texas Instruments Explorer.

Packets of data called tokens will flow between the machines on a high-bandwidth network. Each token will contain 8 to 12 bits specifying its destination.

Arvind plans to use a network in the shape of a seven dimensional hypercube. In this highly symmetrical arrangement, a token can travel to any destination in, at most, seven steps.

Tokens will be routed by a specially designed board, plugged into the backplane of each processor. This board will handle the processor's own communications. It will also store and forward packets which are merely passing through.

Each board will hold a table of routing instructions. Altering the tables will make the hypercube behave like networks of various other shapes.

Arvind's proposed machine is described as a multiprocessor emulation facility. Its role is exploratory. It will not be a machine for end users. It will be a testbed for parallel architectures.

Future parallel machines may be purpose-built for anything from signal processing to artificial intelligence (ai). Their processors will be hardwired together in arrangements which suit the application. It is argued that failures may be avoided if there is a system on which new designs can be tested before they are built.

Until 1978, Arvind was at the University of California. There he developed the Irvine Dataflow (ID) language. The original ID compiler generates Lisp code. It is still used for debugging.

A new compiler, written at MIT, generates dataflow machine language. Conceptually, a machine language program is a dataflow group in which boxes represent operations and arrows represent data dependences. At the moment it is a machine language without a machine. But an abstract dataflow machine has been specified in approximately 300 pages of Lisp code. This code can be regarded as a hardware specification; it can also be run as an emulator, on existing hardware.

'We have a compiler, we have the emulator, and they work together once in a while,' says Arvind. Debugging the two programs has been difficult since they cannot be debugged separately.

Arvind's group has adopted a method of computing known as dynamic dataflow. It requires a more complex machine than Jack Dennis's original static dataflow concept, but it permits more parallelism in the computation and allows the programmer to use procedure calls.

Some observers think that dynamic dataflow machines may be suitable for ai work, while static dataflow machines may offer higher performance in scientific computing.

According to Arvind, dynamic dataflow was invented independently at MIT and by John Gurd's group in Manchester. 'In static dataflow,' he explains 'there is a restriction which says you cannot have more than one token on an arc. In dynamic dataflow there is no restriction. You have as many tokens on an arc as you like.'

The problem is that tokens may arrive in the wrong order. Confusion is prevented by a system of tagging tokens which Arvind compares to colour-coding.

To exploit the available parallelism, the program (a dataflow graph) must be split up and distributed between many processors. Finding a good 'decomposition' of a program is a very hard problem, according to Arvind.

'These are things we want to study,' he says. 'We have lots of ideas, but we are not at all clear which is an acceptable one.'

He believes that the machine itself should decide the division of labour among its processors. This contrasts with the approach of Occam and other languages where the programmer can explicitly assign work to a particular processor.

Instead, Arvind argues that one should choose a language which makes it easy to find parallelism in a program. Functional languages are the most suitable, he says.

A functional program is almost a software model of a dataflow machine, with arguments or inputs going into functions and results coming out of them, to be used as arguments by further functions. Of well-known languages, Lisp comes closest to the functional ideal.

All the arguments of a function can be evaluated in parallel, Arvind points out. 'When you write a program in a functional language you don't obscure the parallelism that was in your original algorithm.'

For the moment he is content to use the data-driven style of computation, in which the program computes all the results it might need later, even though some of them may prove to be unnecessary.

He admires the 'futuristic' work of UK computer scientists like David Turner, John Darlington and Peter Henderson, who favour demand-driven computation. Here, the program computes only the results it needs. It can, therefore, handle infinite data objects without foundering in infinite computations.

But demand-driven computing requires extra bookkeeping. 'The way demand-driven evaluation is usually done,' says Arvind, 'is so inefficient that it doesn't have a chance of succeeding.' He has just written a paper suggesting a method of reducing the bookkeeping.

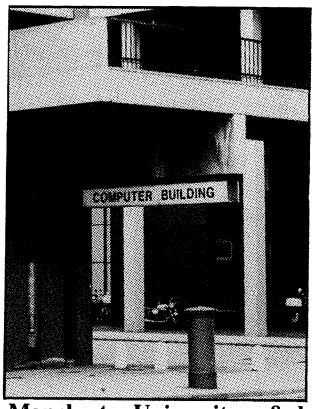
Dataflow computing assumes that functions are to be evaluated, programs are to be executed. But there is an alternative – to simplify the program by some kind of algebraic operation, until there is nothing left and the result simply falls into your lap. This is the method of reduction.

In reduction machines, too, the program is represented as a graph. But no tokens flow along the graph. Instead, it is gradually pruned. This may be done sequentially or in parallel.

Arvind feels there may be a deep connection between dataflow and reduction. He believes dataflow may be moving closer to parallel reduction, with the addition of data structure storage. When tokens become too big they can be stored in memory. Instead of the tokens themselves, pointers to their locations flow through the graph.

'If someone asked me to bet money today,' he says 'I'd guess that hardware-wise, those two things will turn out to be the same.'

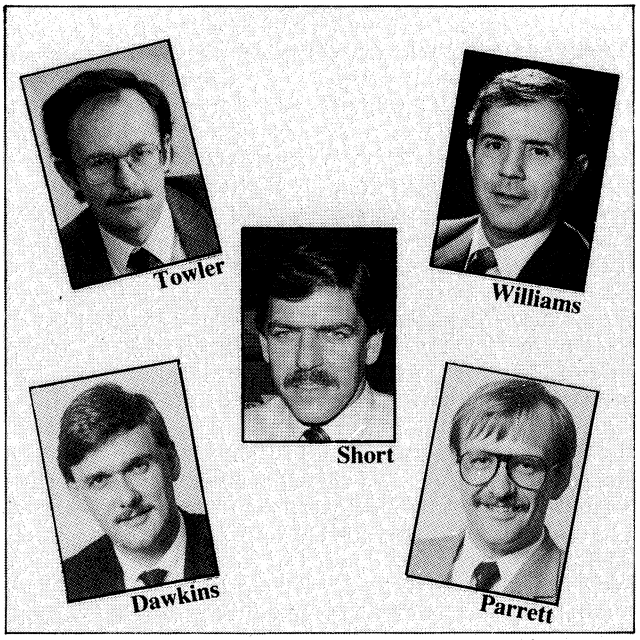
Tony Durham is a freelance journalist. ■



Manchester University: r & d

Arvind's machine will not be for end users, but will offer a testbed for parallel architectures

PEOPLE



Graham Davies is the new marketing and sales director at STC Components, taking over from Ronnie Luckman who retired this month. He joined STC in 1971 after 10 years with the Royal Naval Scientific Service, where he was involved in the design of radar and missile control systems.

At STC, he progressed from the position of general manager of switching installations to director of operations for ITT Africa and the Middle East, based in Brussels. Two years ago, he moved to the Industrial Products Group. To prepare himself for his new job, he is currently carrying out a programme of visits to STC's manufacturing units in the UK, and its marketing subsidiaries in France, Germany and North America.

Chris Towler has been appointed principal consultant in the systems division of Software Sciences, after 17 years in the computer industry. He has worked in both technical and commercial computing for users, manufacturers and systems houses, and before joining Software Sciences was technical services manager in British Nuclear Fuels' financial directorate.

Towler will be based at Software Sciences' Macclesfield office, and will be responsible for project management and consultancy.

Software Sciences has also appointed **Neville Williams** as sales manager for its air defence and electronic warfare systems. Williams previously worked for British Aircraft as chief communications engineer, and Plessey Radar as manager of the advanced systems group.

Altergo Products has appointed two new systems engineers. They are **Steve Dawkins**, who previously worked for the London Borough of Harrow, and **Steve Parrett** formerly of Kodak.

They will be responsible for supporting the sales and marketing division, as well as providing service to Altergo's UK customers and overseas agents.

Martin Short has left STC Business Systems to join Mitel Telecom as national dealer manager. He has 11 years' experience in the business systems market, with 3M and Philips Business Systems as well as STC. At Mitel, he will build up the dealer network for the Kontact multi-tasking work station.

Lester George, formerly managing director of Ferranti Instrumentation, has been appointed to the Ferranti main board and has moved to head office. **Albert Dodd** takes over from him at Instrumentation, and also becomes managing director of Ferranti Engineering Holdings.

Roy Boyle becomes Ferranti's new personnel and industrial relations manager. ■

INTRODUCTION

1 Brief summary of Dataflow techniques

2 The Manchester Prototype - hardware & software

3 Evaluation Method - measurements & meaning

4 Evaluation Results

5 Interpretation & Futwe Work

J.R.Gurd
21st October 1982

Dataflow Program Graphs

Instructions appear horizontally if they can be executed concurrently, and vertically if they must be executed in sequence.

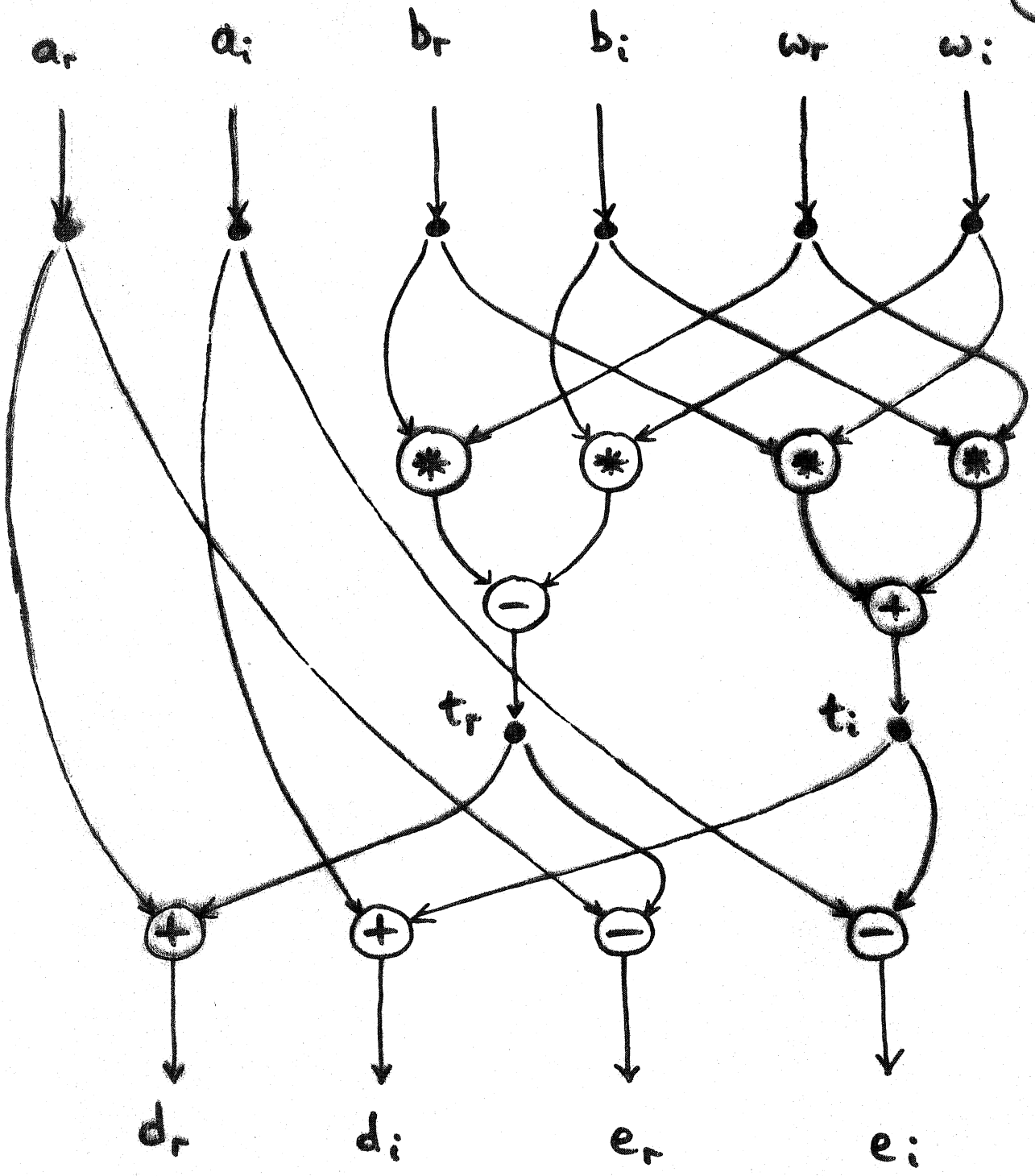
Sequencing constraints are indicated by

data dependence arcs

drawn between the instructions (or **nodes**).

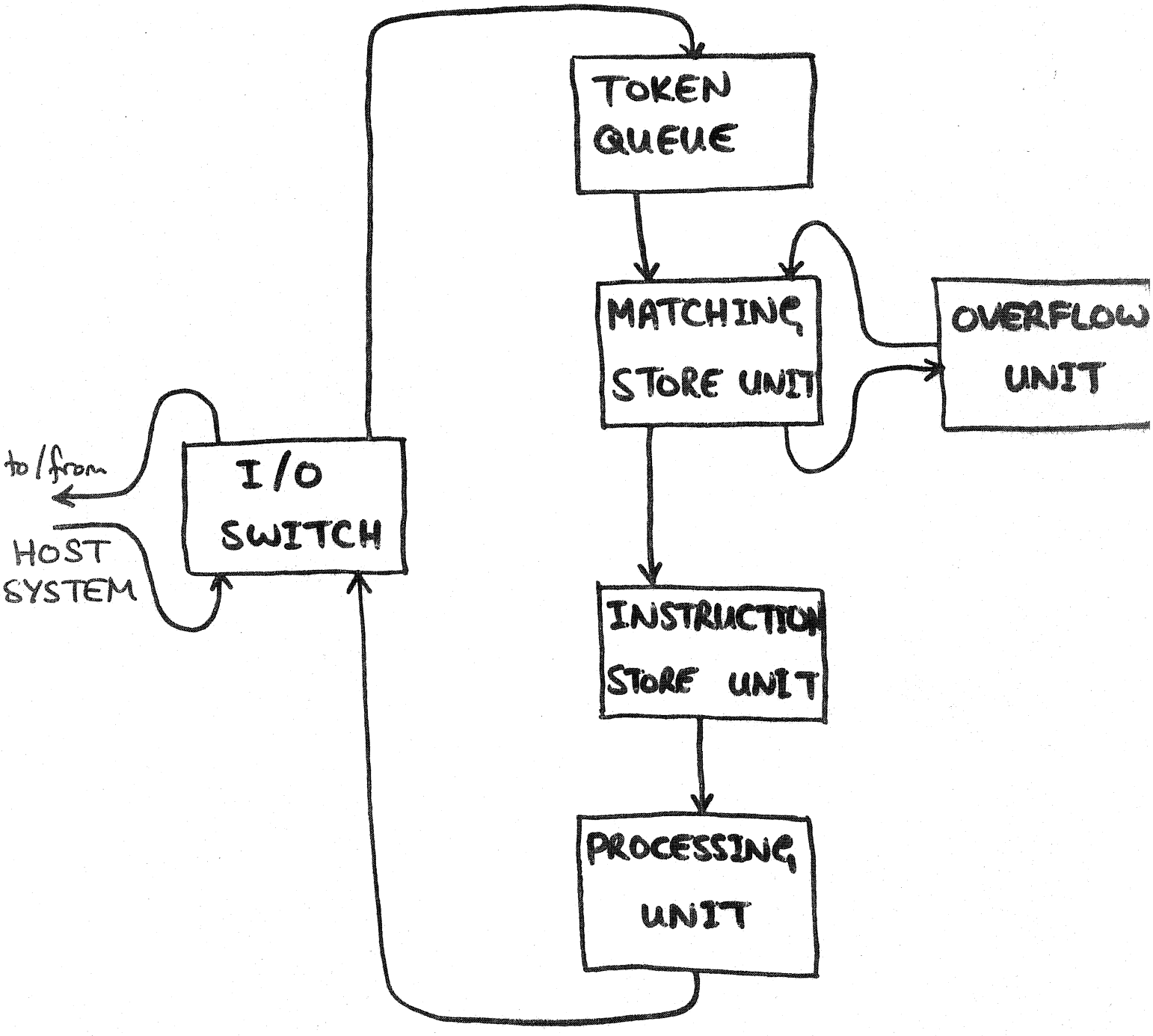
Instructions do not reference memory - data (in the form of **token** packets) is sent directly along the data dependence arcs.

Instruction execution is prompted by the presence of data tokens at <u>all</u> input points.



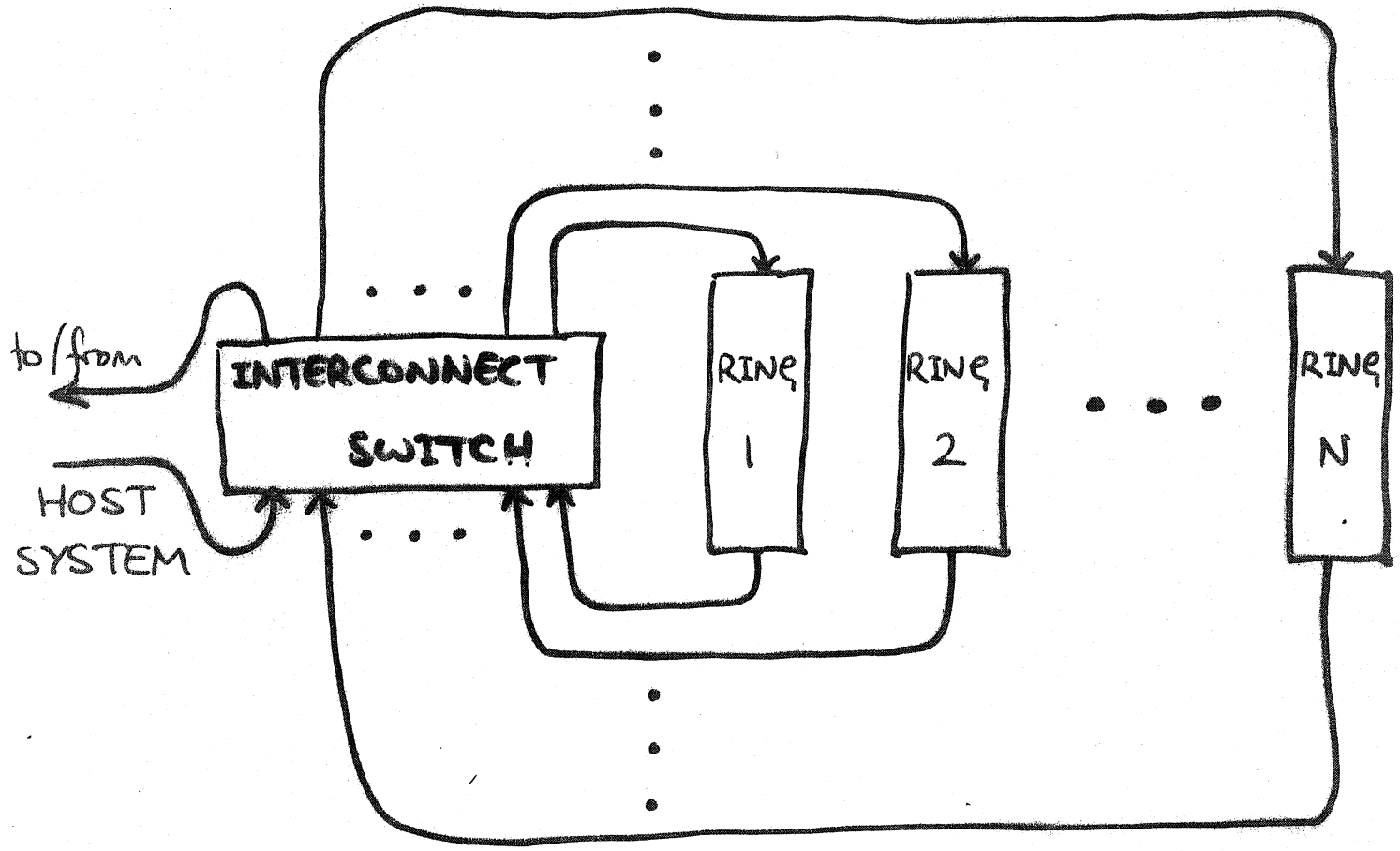
FFT Butterfly

1-ring Dataflow Processor

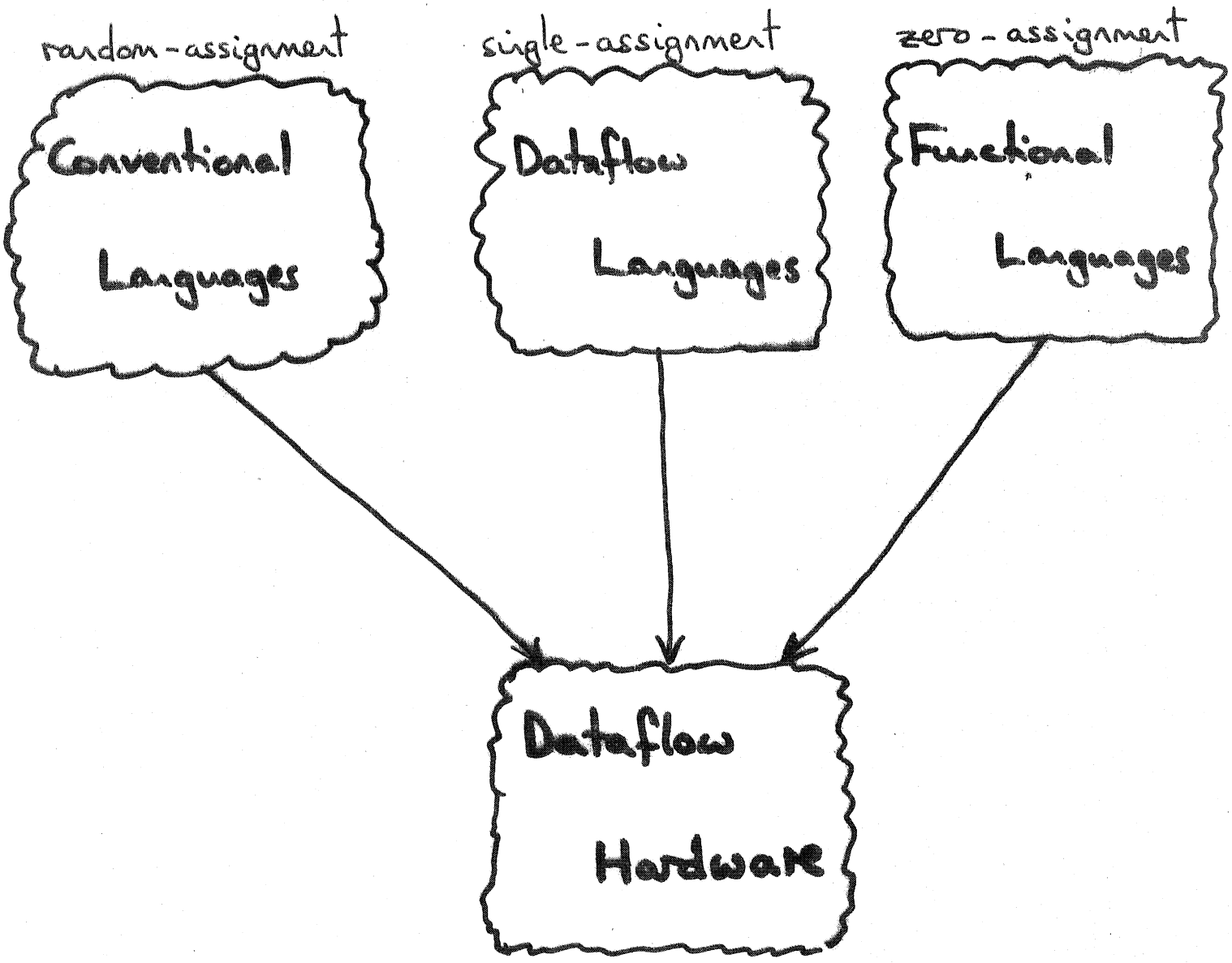


Multi-ring Dataflow Processor

'add-on' processing power



Dataflow Programming



The Manchester Dataflow Project
a brief history

date	activity	staff
1976 -	start date : April	2 lecturers
	paper system design ; 1st simulators	
1978 -	1st SRC grant : £75k over 3 years	+1 RA
	to build prototype 1-ring hardware	+1 lecturer
1980 -	2nd SRC grant : £25k over 3 years	
	to develop software tools	+1 RA
	1st major SRC project review : December	
1981 -	prototype hardware ran first program : October	
	3rd SRC grant : £400k over 4 years	
	to evaluate & enhance 1-ring hardware	
	to extend to 4-ring hardware	+1 RA
	to make hardware available to external users	
	to develop applications codes	
1982 -	matching store upgrade	
	2nd major SRC project review : September	

---

total staff : 6 lecturer/RA

+ 6 grad. students

Achievements at Manchester

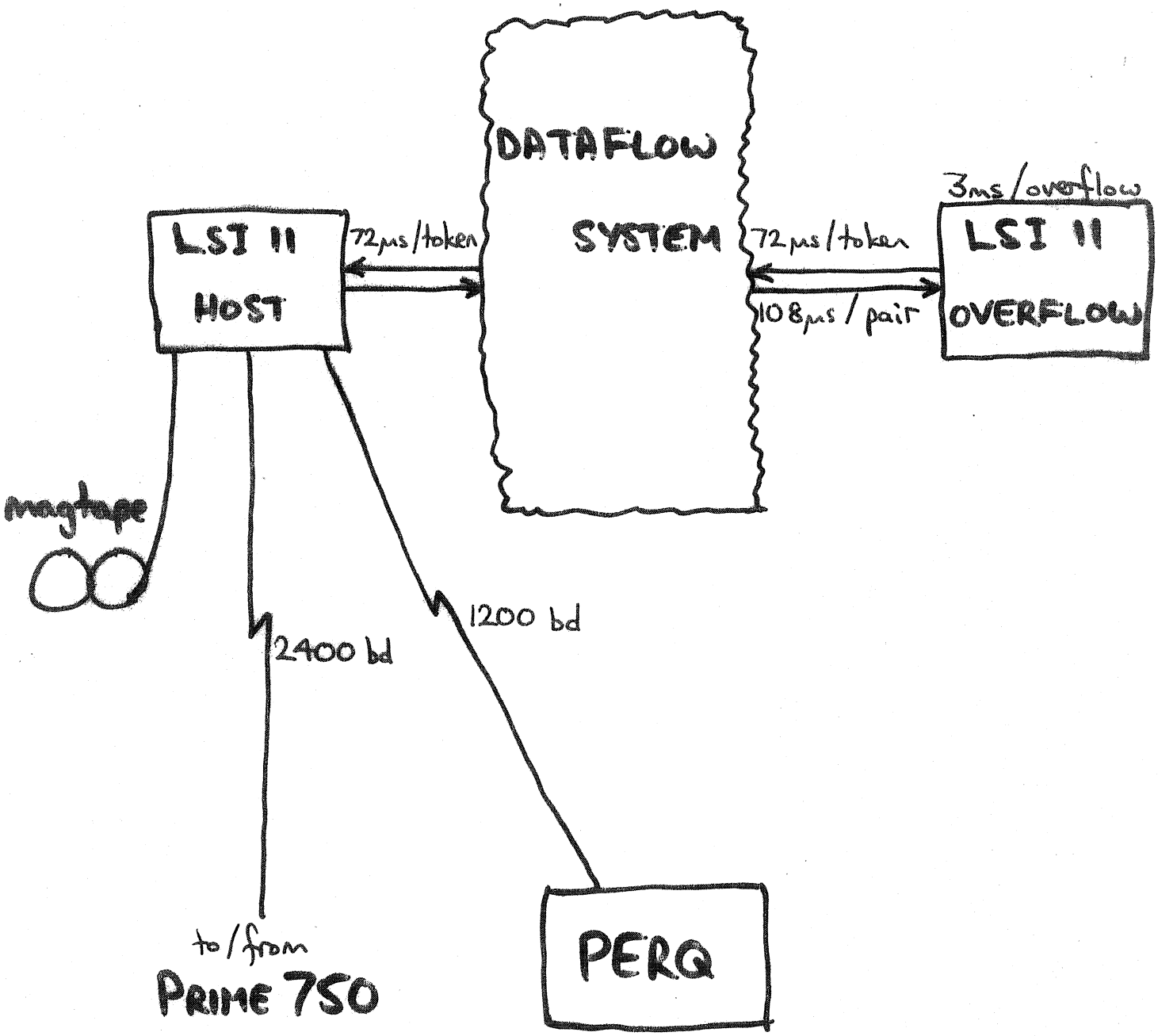
overview

1. Prototype dataflow hardware running 1 - 1.5 MIPS

2. Prototype software support system in operation

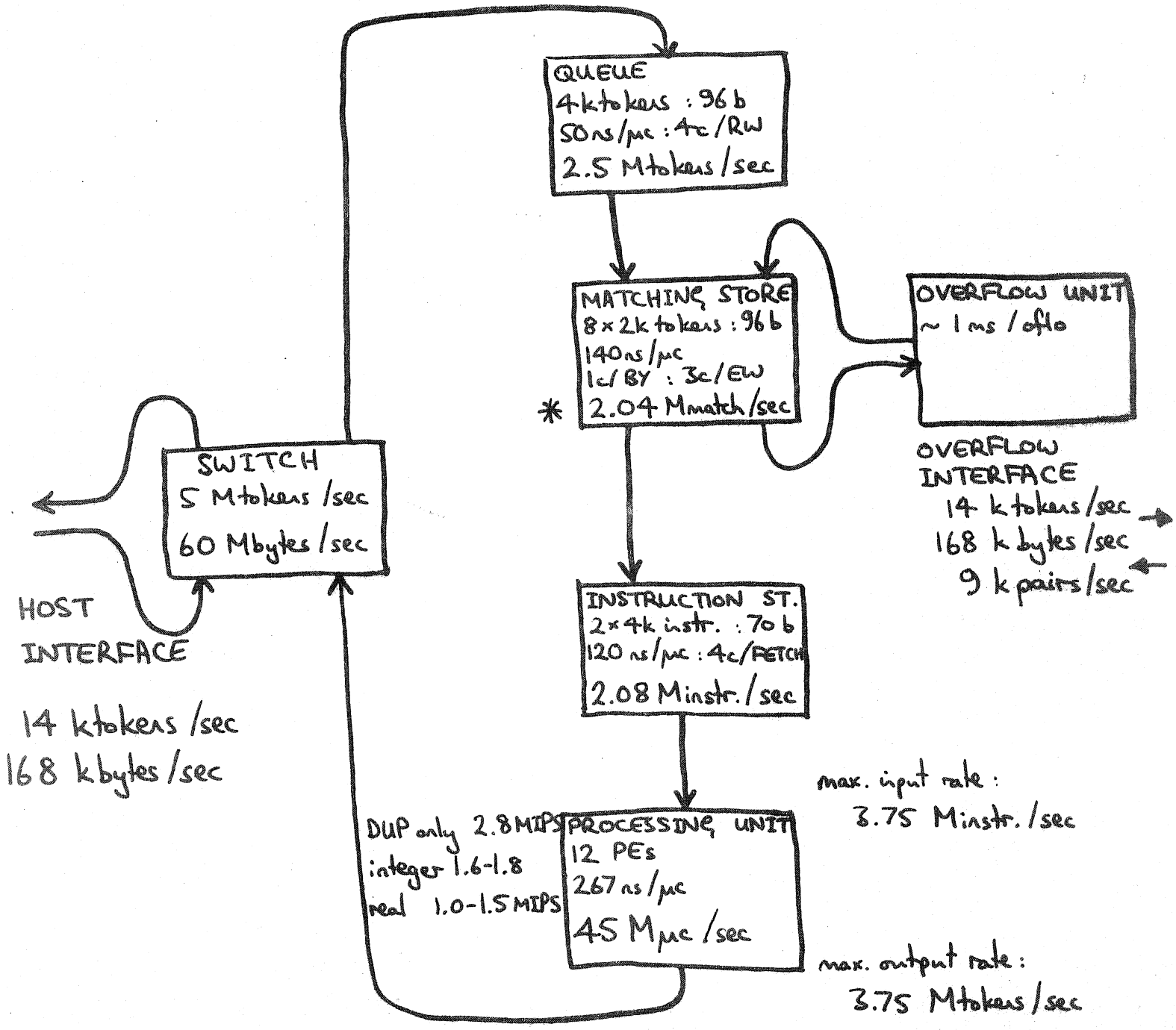
3. Preliminary performance evaluation in progress

The Hardware Environment



current status

1-ring Dataflow Processor



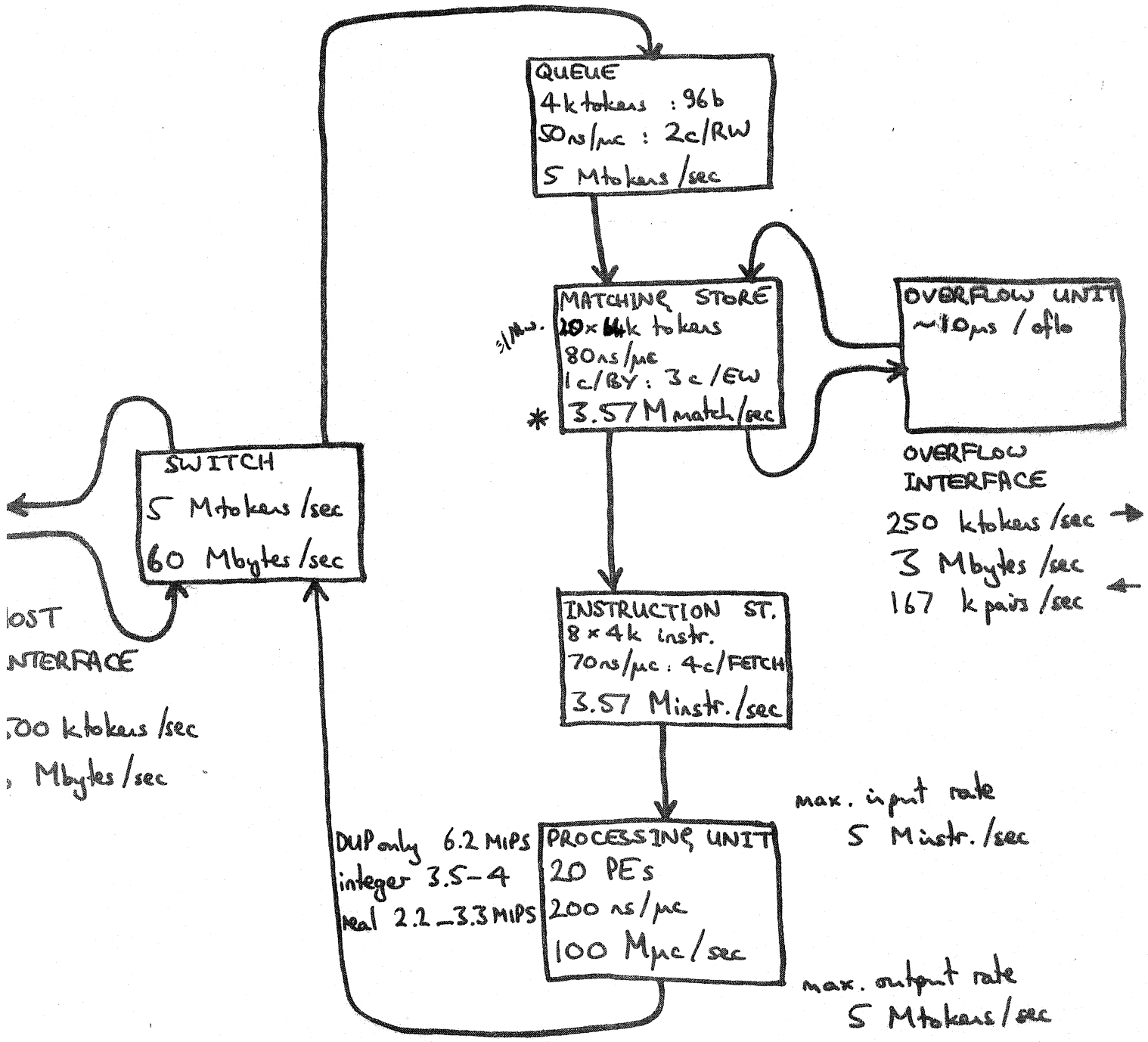
* is average rate - all others are maxima

~ 4000 debugged MSI TTL ICs

90% of opcodes implemented (equivalent to simulator)

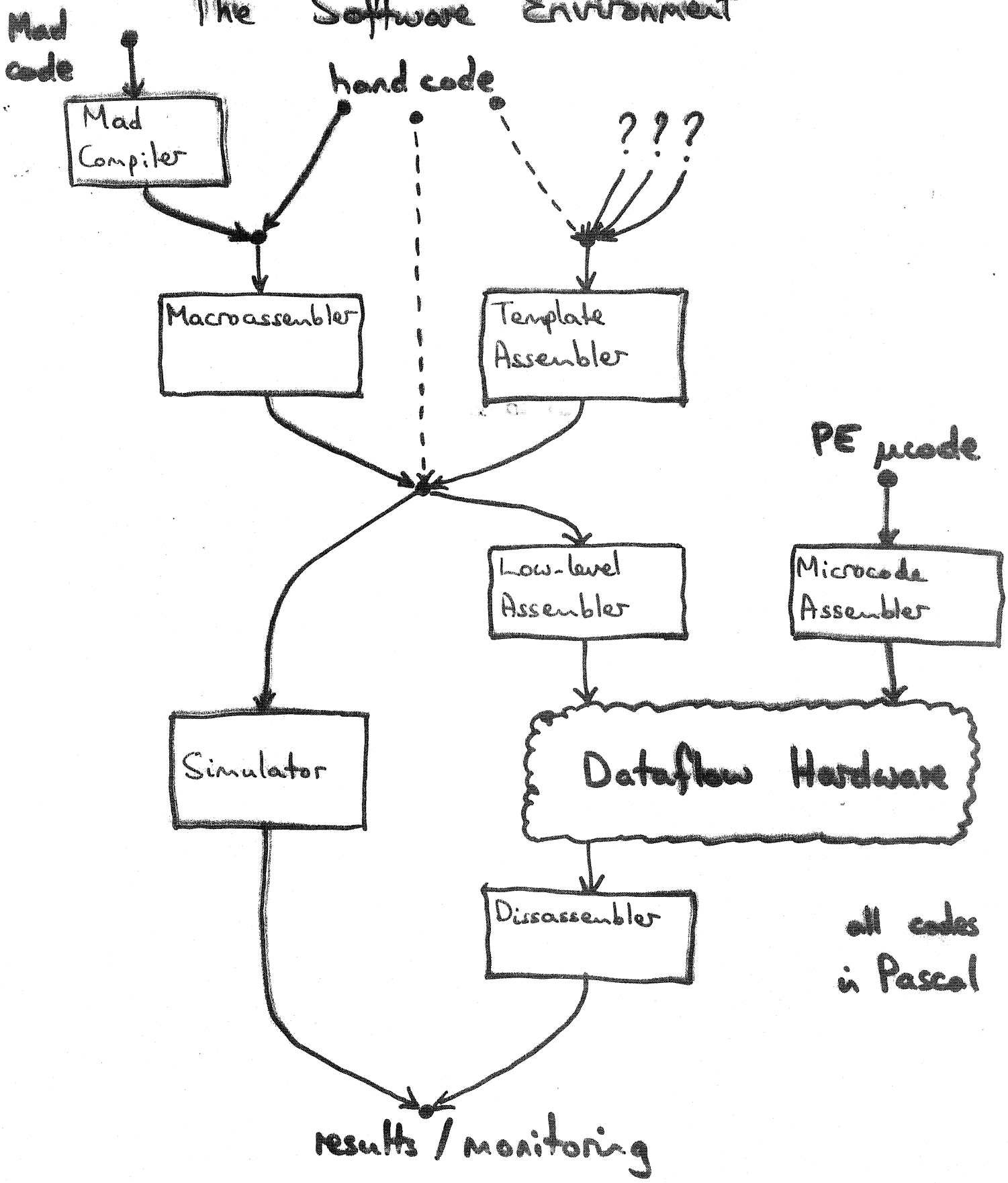
proposed upgrade

1-ring Dataflow Processor



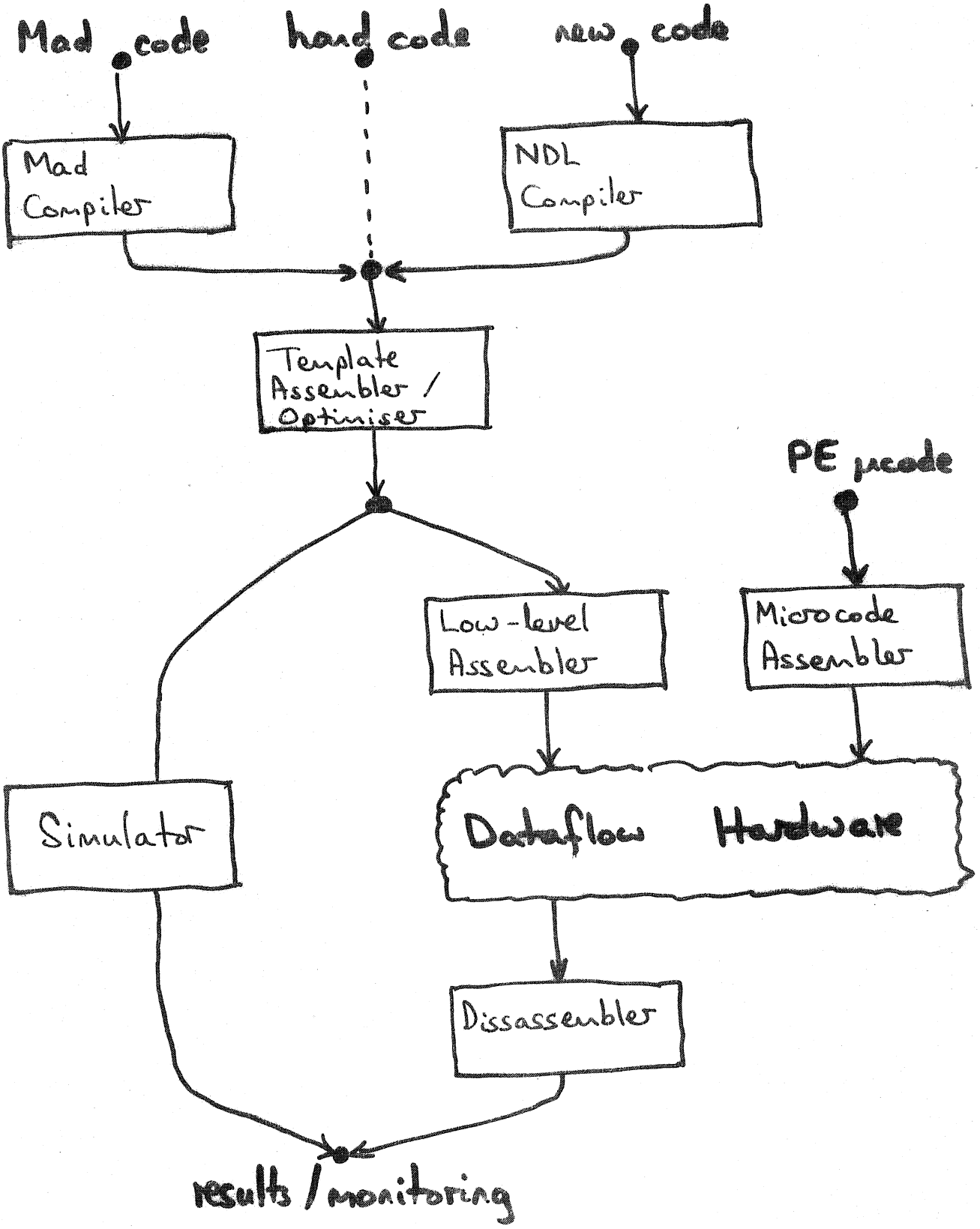
* is average rate - all others are maxima

The Software Environment



proposed upgrade

The Software Environment



Evaluation Objectives

1. Tune prototype hardware for optimum performance.

2. Determine the nature of software parallelism that can be exploited by the hardware.

3. Establish the value of a dataflow MIP.

Evaluation Method

1 For programs that do not overflow

* Plot speedup curves
* Interpret results

Rectify problems in processor & pipeline hardware

2 For programs that overflow a little

Determine nature of bottlenecks in overflow loop

Rectify problems in hardware / software

3 For programs that overflow a lot

Design & implement a hierarchical memory

Evaluate & optimise performance

Basic Program Measurements

S_1 total number of instructions executed

$\equiv$ number of execution steps with 1 PE

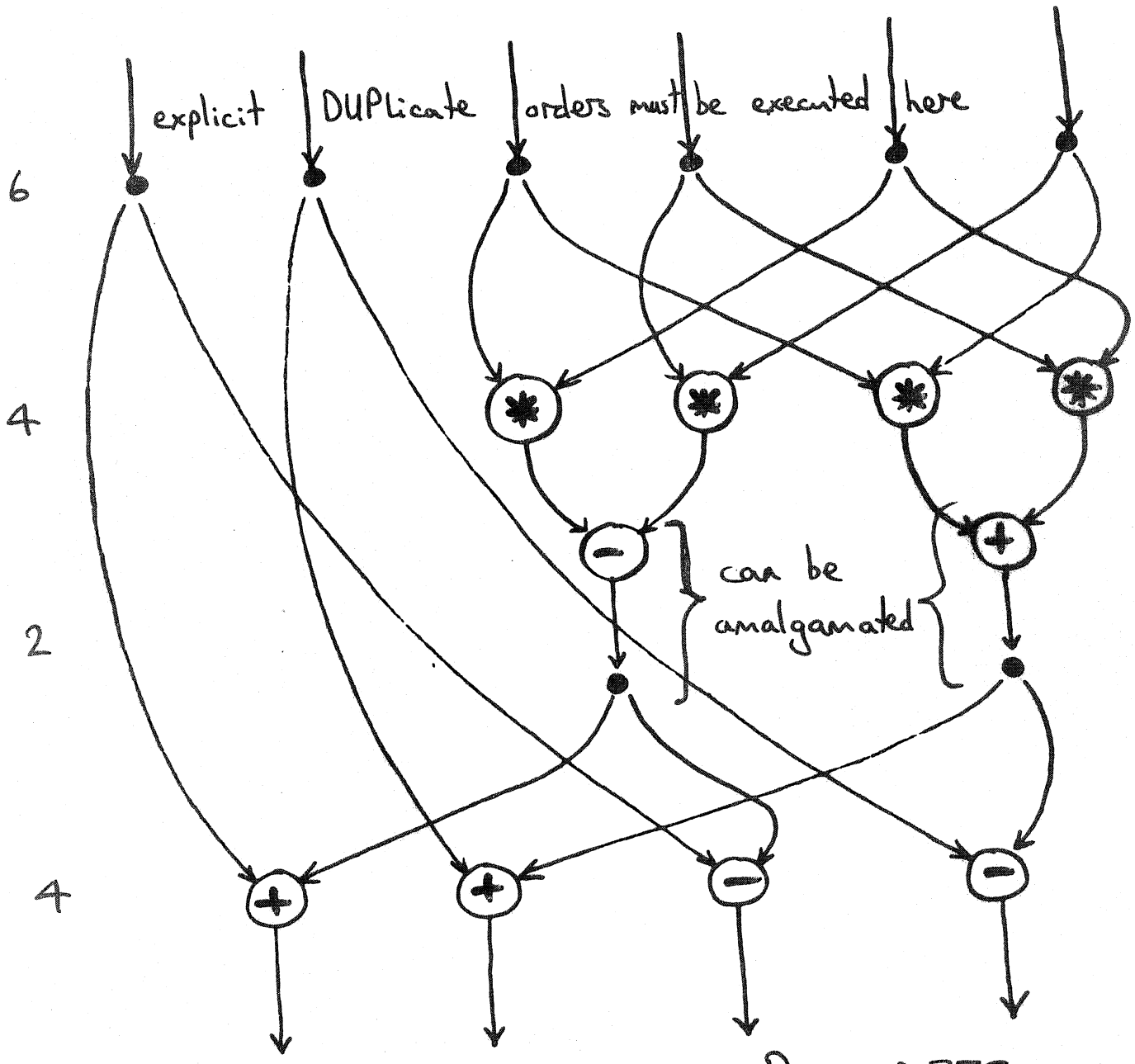
S_∞ number of execution steps with unlimited PEs

(N.B. notional only : steps are not of equal time)

$\pi = \frac{S_1}{S_\infty}$ a rough measure of program parallelism

(but does not account for time-variance of parallelism)

P_{BY} proportion of executed instructions which have one input (i.e. BYpass matching function)



$S_1 = 16$

$S_\infty = 4$

$\pi = 4$

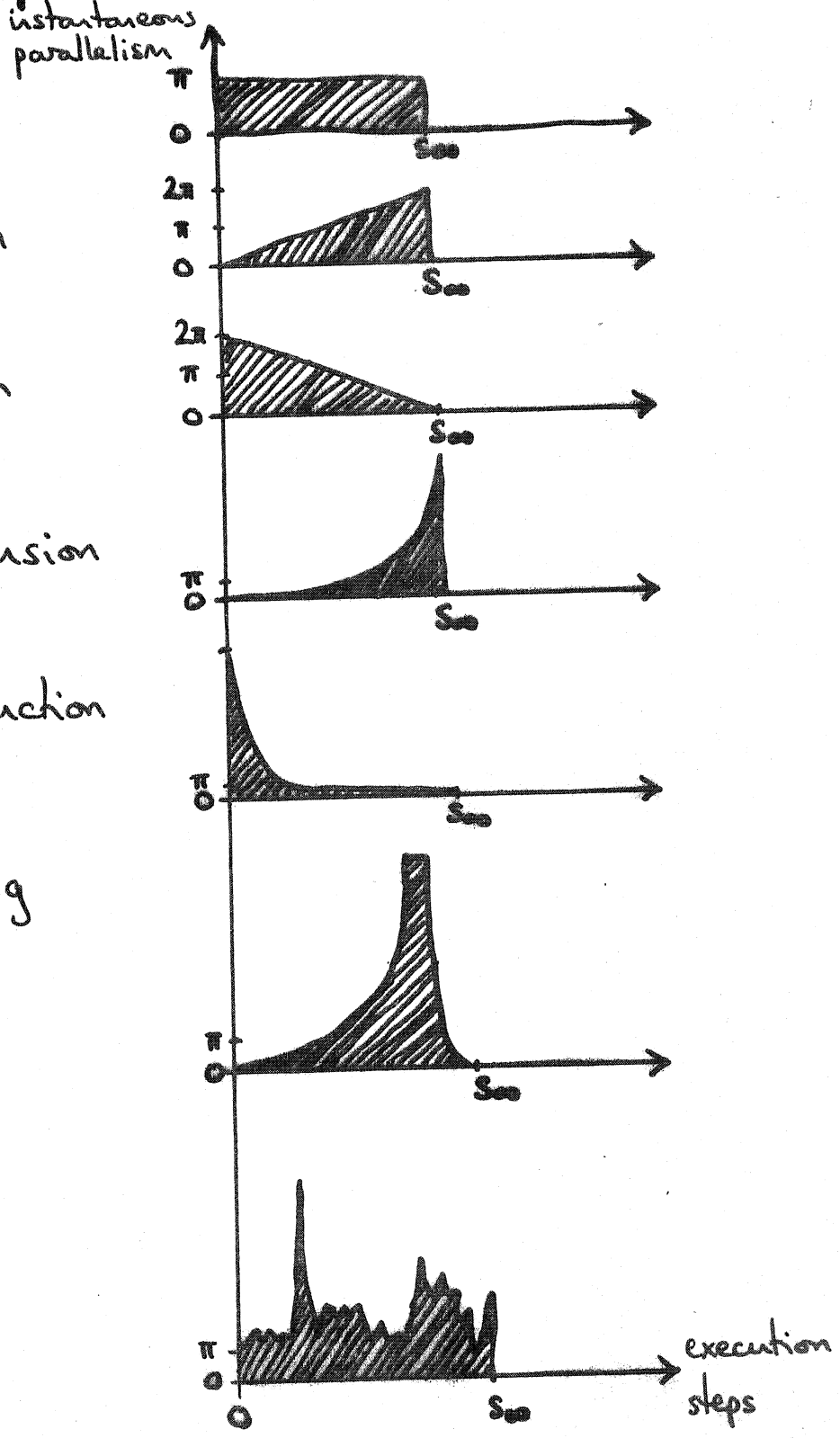
$P_{BY} = 0.375$

e.g. FFT Butterfly

Time-variance of Program Parallelism

(area of each block = S_1)

1. constant
2. linear expansion
3. linear reduction
4. exponential expansion
5. logarithmic reduction
6. recursive doubling (4 - 1 - 5)
7. irregular (mixed)



Basic Hardware Measurements

T_1 total execution time for 1 PE

T_n total execution time for n PEs

$P_n = \frac{T_1}{T_n}$ effective number of PEs when n are active

$\epsilon_n = 100 \frac{P_n}{n}$ % utilisation of n PEs

$M_n' = n \frac{S_1}{T_1}$ potential MIP rate for n PEs

$M_n = \frac{S_1}{T_n}$ actual MIP rate for n PEs

Benchmark Codes
(simple & no overflows generated)

name	source	parallelism	P_{BY}	comment
IPCPRO	low-level	 $\pi = 1 \rightarrow 16+$	1	DUP/BY test only
FFT	macro assem	 $\pi = 50/100$	.7	complex 32/64 point FFT
LAPLAC	macro assem	 $\pi = 50/130$	.7	real iterative Laplace relaxation
SUMPRO	mad	 $\pi = 2 \rightarrow 150+$	.61	recursive doubling integer sum
INTC	mad	 $\pi = 12$	.64	trapezoidal integration
PLUM1	macro assem	 $\pi = 50$	.61	plumbline algorithm
PLUM2	mad	 $\pi = 20$	.56	plumbline algorithm
HUGE	macro assem	 $\pi = 50/70$	.63	logic simulation
LEETC	mad	 $\pi = 20/40$	.61	lee router test

Program Measurement Table

Program Name: INTC
Source: Mad
Description: Integrates area under a curve using trapezoidal method.

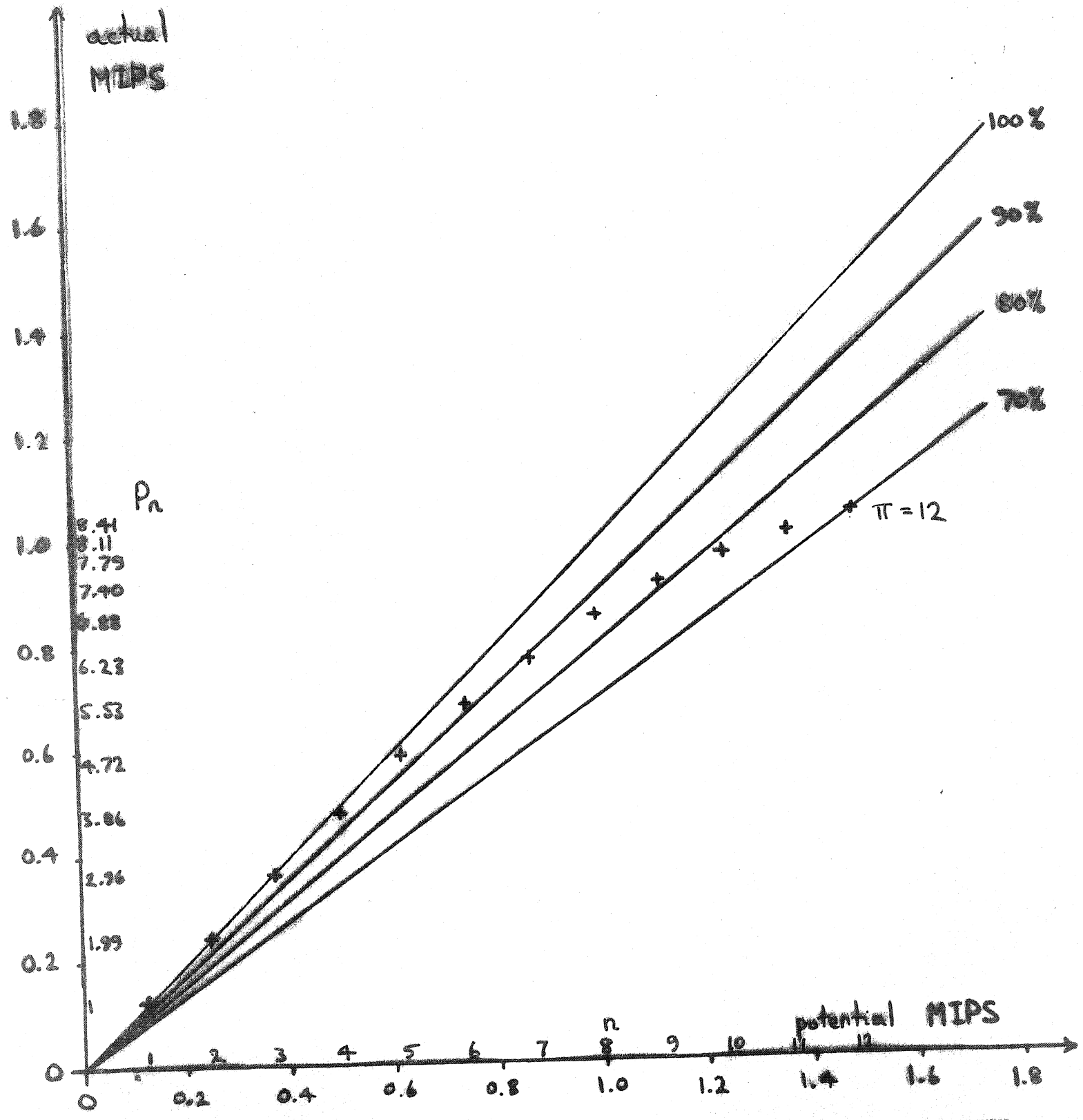
Data Name: INTD
Data Size:

S_1 1965
S_n 166
Parallelism 12
P_n 0.64
Non-Optimized Portions NONE

n	time for n PEs T_n	effective # PEs P_n T_1/T_n	%PE utilization ϵ_n $100P_n/n$	potential MIPS M_n' nS_1/T_1	actual MIPS M_n S_1/T_n
1	.01582	1	100	.124	.124
2	.00796	1.99	100	.248	.247
3	.00537	2.96	98	.373	.366
4	.00410	3.86	96	.497	.479
5	.00335	4.72	95	.621	.587
6	.00286	5.53	92	.745	.687
7	.00254	6.23	89	.869	.774
8	.00230	6.88	86	.994	.854
9	.00214	7.40	82	1.118	.918
10	.00203	7.79	78	1.242	.968
11	.00195	8.11	74	1.366	1.008
12	.00188	8.41	70	1.491	1.046

Speedup Curve

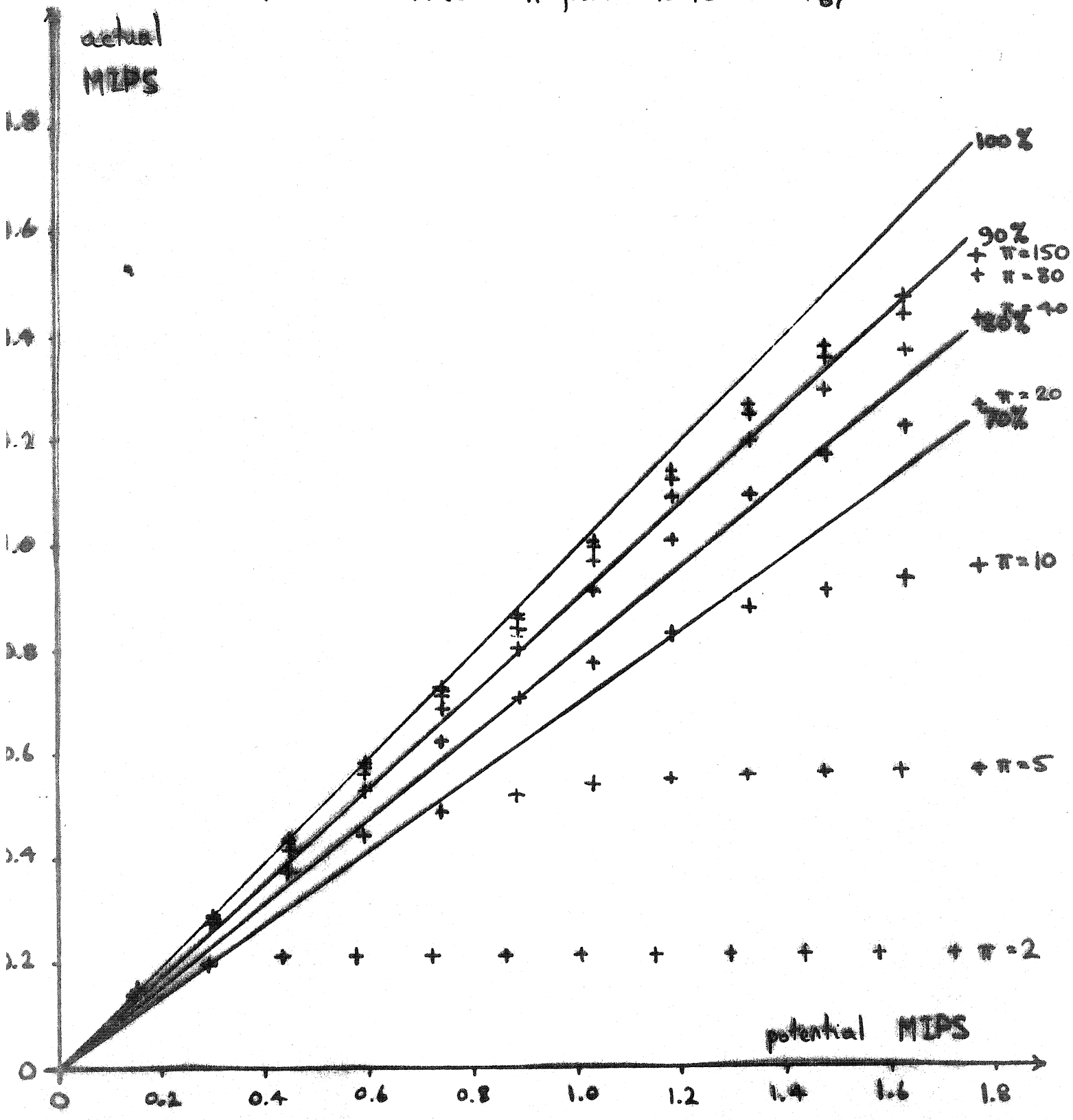
+ INTC $\pi = 12$ $P_{BY} = 0.64$



Speedup Curves

effect of variable π

+ SUMPRO π from 2 to 150 $P_{BY} = 0.61$

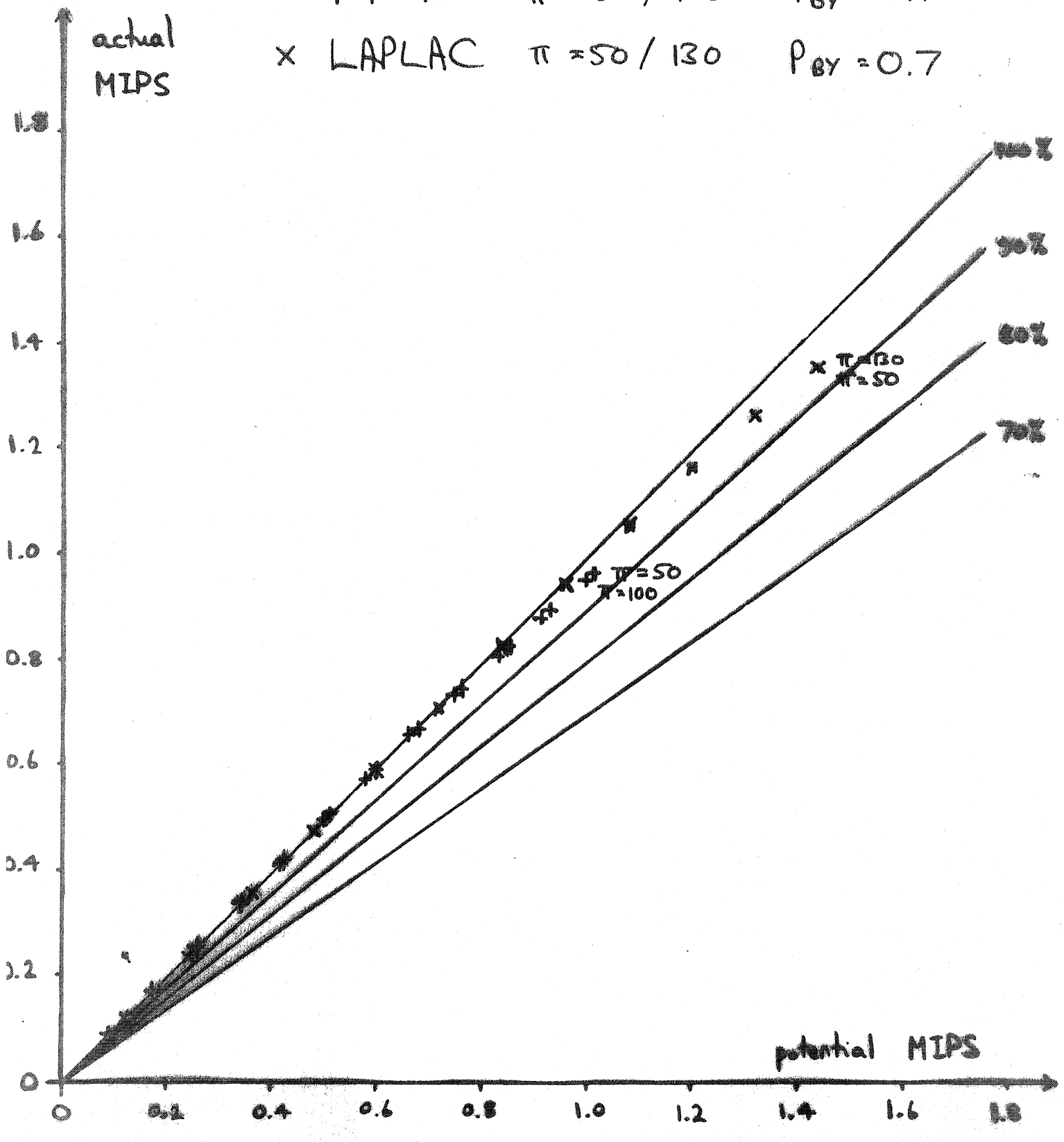


Speedup Curves

floating point array codes - variable instr. mix

+ FFT $\pi = 50 / 100$ $P_{BY} = 0.7$

× LAPLAC $\pi = 50 / 130$ $P_{BY} = 0.7$

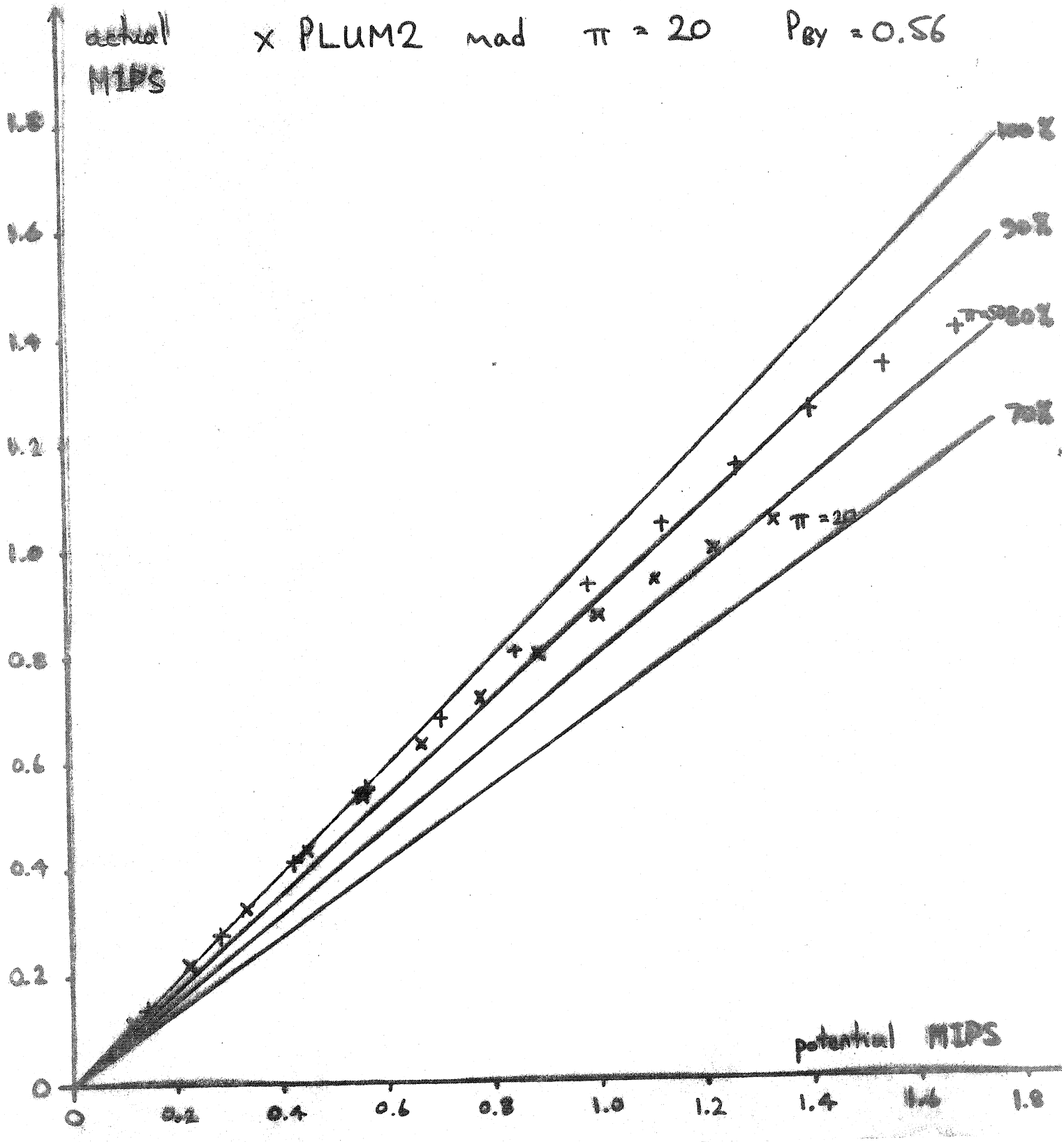


Speedup Curves

effect of source language

+ PLUM1 macro $\pi = 50$ $P_{BY} = 0.61$

× PLUM2 mad $\pi = 20$ $P_{BY} = 0.56$

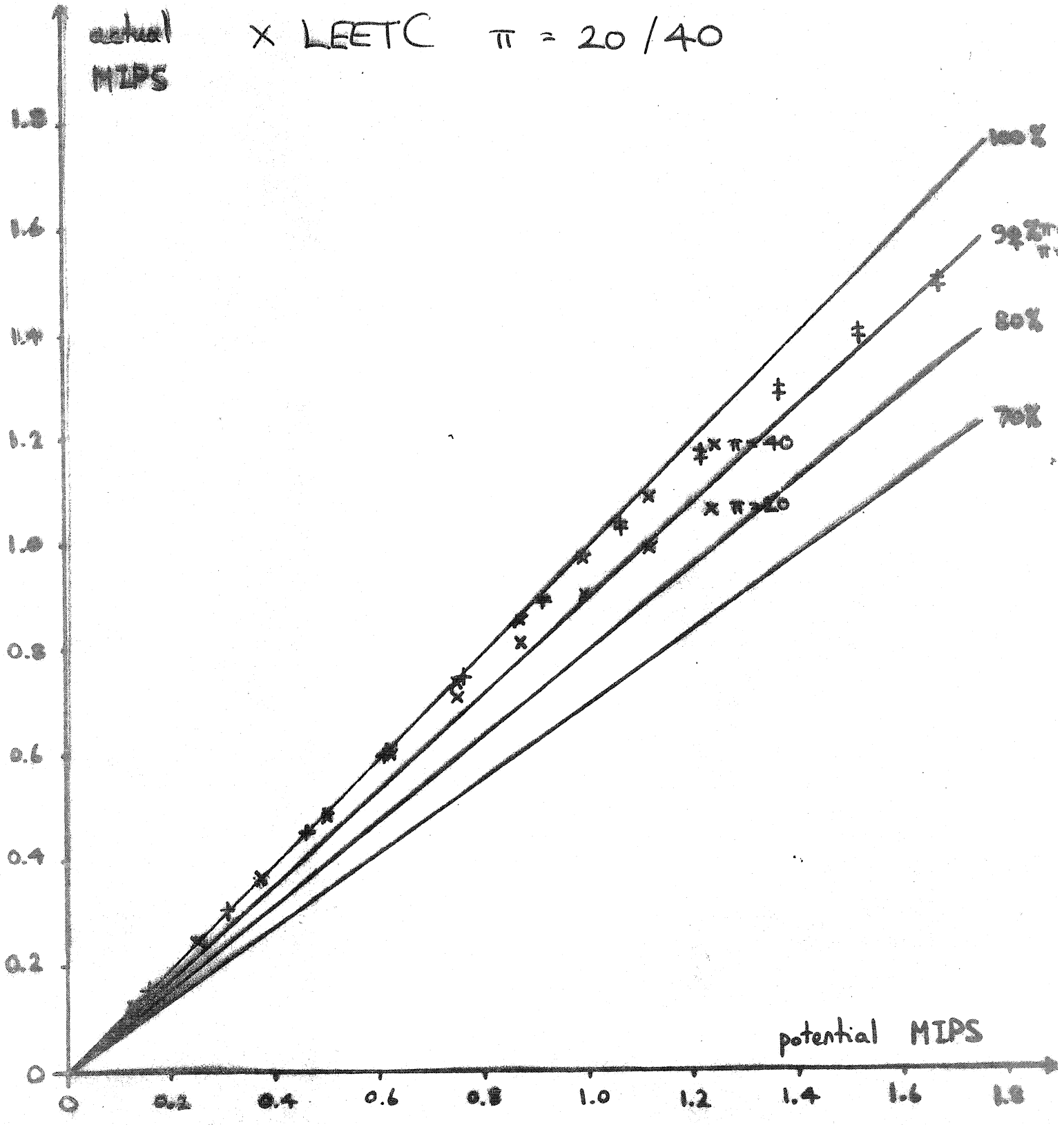


Speedup Curves

prototype CAD codes

\+ HUGE $\pi = 50/70$

× LEETC $\pi = 20/40$



Possible causes of fall-off in speedup curves for high values of π

1. Serial sections of code dominate timing

2. PE output arbitrator suffers contention

3. Pipeline buffering is inadequate *

major suspect

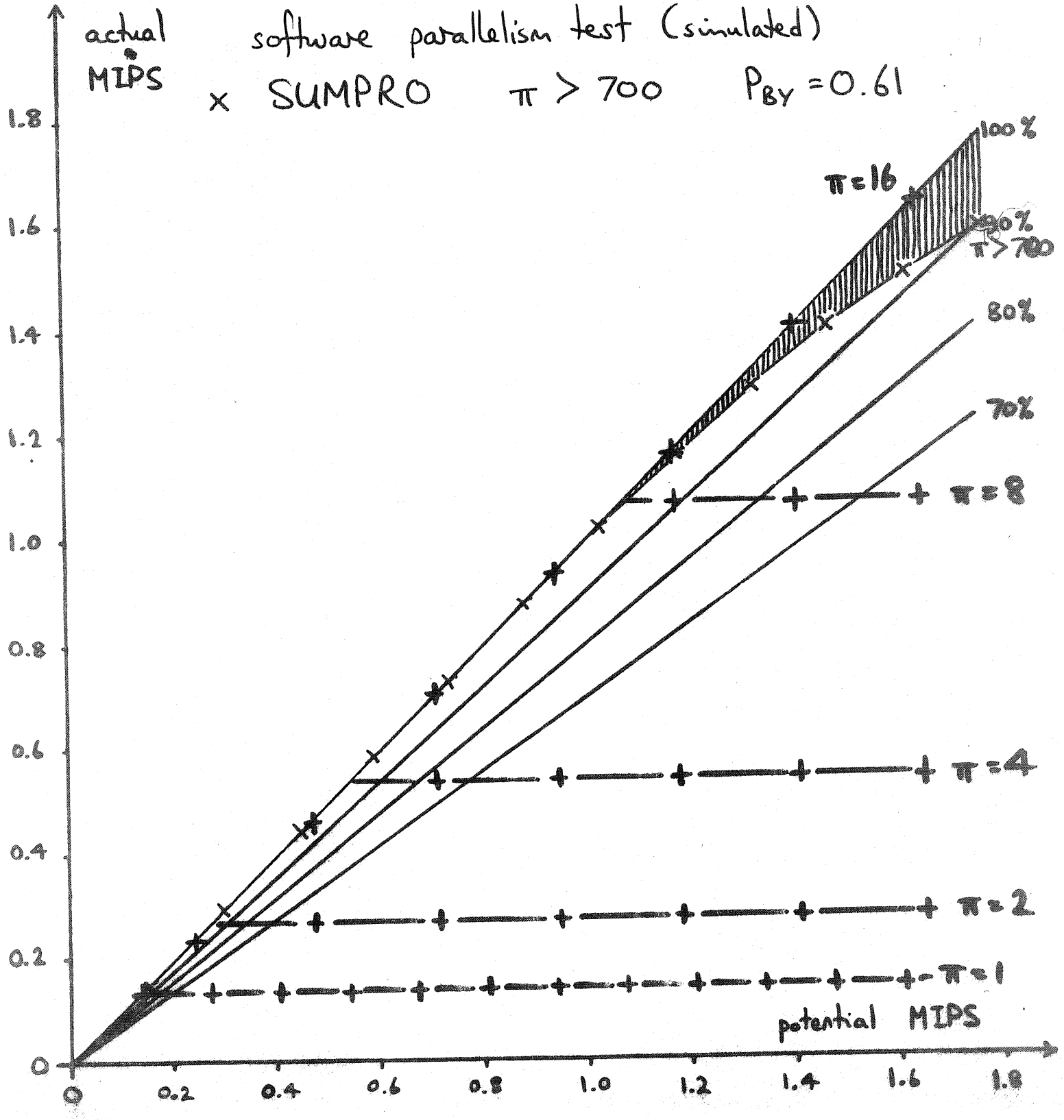
Speedup Curves

PE contention test

\+ IPCPRO $\pi = 1, 2, 4, 8, 16$ $P_{BY} = 1$

software parallelism test (simulated)

× SUMPRO $\pi > 700$ $P_{BY} = 0.61$



Utilisation of Matching Store
preliminary comments

1. Present hash algorithm not very effective.
(May not even be what we think it is !).

2. Overflow interface and processing rate far too slow and unsophisticated to obtain realistic results for programs with overflow.
Impossible to study memory hierarchy with existing equipment.

3. Larger matching store could defer overflow problem, but will also defer study of memory hierarchy.

SUMMARY

Preliminary Results of Evaluation Study

1. There is speedup (vs. number of PEs) for a wide variety of programs : i.e. programs have parallelism.
2. The crude measure $\pi = \frac{S_1}{S_\infty}$ is a good indicator of a program's suitability for the system, regardless of any time variance of the program parallelism.
3. The introduction of additional pipeline buffering should improve speedup curves considerably.
4. The intended clock speeds and number of PEs should give a reasonable match of processing rate to pipeline beat for floating point codes.
5. More work needed on the Matching Store and Overflow Unit.
6. No realistic assessment of the value of a dataflow MIP yet available.

A MULTILAYERED DATA FLOW COMPUTER ARCHITECTURE

JOHN GURD, IAN WATSON

AND JOHN GLAUERT

Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester, M13 9PL,
England

DRAFT : July 1978

A Multilayered Data Flow Computer Architecture

John Gurd, Ian Watson
and John Glauert

ABSTRACT

This paper introduces a multilayered data flow computer architecture based upon a high level language and underlying graphical notation for representing computations.

The language, LAPSE, employs a single assignment rule and several more familiar parallel programming constructs. It permits both iteration and recursion.

The graphical notation utilises the concept of labels for those tokens which use computational subgraphs reentrantly. The operation of labels in implementing iteration, recursion and data structures is illustrated.

The architecture is based on a ring-structure in which binary representations of tokens, known as results, circulate. A processing unit computes results as required by a stored program which represents the computational graph. Each result carries with it a name which is derived from the notational label. Names perform two functions : Firstly they separate different instantiations of reentrant code; secondly they can be associatively matched with other inputs to their destination node, to determine when the node may be executed.

The ring-structure can be pipelined to achieve an instruction execution rate which is limited by the amount of parallel activity permitted by the program and the technology used.

An extensible version of the architecture contains many such rings in a multilayered structure whose execution rate is bound solely by the inherent parallelism of the programs running on it. The architecture can also be organised so as to exhibit a high degree of tolerance to hardware component failures.

CONTENTS

Notes : References are given in square brackets (e.g. [TW75]).
Superscript numerals (e.g. [1])refer to the footnotes at the end of the paper.

1. INTRODUCTION

1.1. Parallel Processing Systems

Substantial interest in parallel arrangements of processing units has recently been expressed. The reasons for this are twofold. Firstly, an insatiable appetite for greater computing power is being less and less catered for by technological advance alone. Secondly, we are beginning to discover that many tasks are not performed efficiently by conventional uniprocessors.

Many parallel configurations of processors have been proposed for tackling specific problems such as pattern recognition, signal processing, and solving partial differential equations. The majority of these architectures use the fact that in such problems many similar operations are performed on independent sets of data. These operations can usefully be executed in parallel. The result of this line of thought has been the family of SIMD (also known as parallel or array) computers [TW75], and some kinds of associative computer [YF77]. These systems have tended to be rather specialised, creating many programming problems when used as general purpose machines.

Discounting pipelined (or overlapped) computers[1] [RL77], we see few proposals for parallel systems which provide a general increase in computing speed regardless of the application. Most proposals in this category can be classified as multiprocessors [En77] in which conventional processors closely share a common store via communications switches[2]. Three recent examples of experimental multiprocessors are C.mmp [WB72], IMP Pluribus [Or75], and Cm* [SF77]. Little is know about the performance of these systems : a few reports have been published on C.mmp [Fu76, WH78] and Cm* [FJ77]. It seems that there are many inherent difficulties on the software side of multiprocessors. In particular, awkward problems are posed by the initial partition-

ing of programs into suitable segments, and by the runtime intercommunication between and scheduling of segments via the operating system [Ba73, En77].

One solution to the partitioning problem is of interest to us. It has been adopted in the Algol 68 runtime system on Cm* [HK77]. Variables may be declared as eventual types whose value may be formed while other values are being computed. If the value is needed in the evaluation of an expression, then that evaluation is automatically suspended until the eventual value has been formed. The necessary synchronisation is performed using flag bits held with the eventual values. Note that this scheme is virtually identical to that used in the hardware of the CD 6600 to allow non-conflicting arithmetic operations to proceed in parallel [Th70]. In that case the necessary synchronisation is performed by the "scoreboard", using hardware flags. In both cases, the synchronisation mechanism ensures that the second of two conflicting demands is delayed until the first has been serviced.

1.2. Data Flow Computation

The two examples above are rather special cases of the use of a data flow computational schema. In such a schema, programs are usually expressed as flowgraphs which explicitly show the existence of potential parallel activity. This is in marked contrast to the strictly sequential formats of conventional programming languages and flowcharts, which hide parallelism. It is not surprising that data flow notation has been studied carefully in the context of parallel computation.

The first comprehensive theory for a graphical computational schema was published over a decade ago [KM66]. It was preceeded by several suggestions for graphical programming [YK58, Br62]. Since then various refinements and several other graphical schemata have been advanced [Ad70, Sl70, Ko73, Ba74, DF74,We75, Ru77, AG77] (see [Mi73, Ba73] for early surveys). Each

schema is a particular variation of the general theme of computational graphs. This paper contains a brief description of a similar schema.

Elsewhere there have been proposals for linear, single assignment programming languages whose properties reflect the attributes of flowgraphs [TE68, Ch71, Sy76, AW77, AG78]. The similarity of some of these languages to natural descriptions of flowgraphs is quite remarkable since they have been developed in complete isolation. In the case of the language Lucid [AW77], the criteria for development were concerned with the provable properties of programs written in the language. Other languages, such as Id [AG78] and DFL [Ac78] have been designed specifically with data flow in mind. In this paper we outline the constructs of a single assignment language, LAPSE [Gl78], alongside the description of our data flow schema. We emphasize the close relationship between the language constructs and their graphical counterparts.

Computer architectures for data flow schemata embody very different principles to those of the traditional von Neumann systems. Some of these are described by Glushkov et. al. in their paper on recursive machines [Gl74]. The term configurable computers has been coined by Miller and Cocke [MC74] as a general expression for flowgraph-based systems. Specific architectures have been proposed in some of the theoretical papers referenced above and elsewhere [DM75, Pl76, Co76, Ru77, Da77, DM77, AG77, SM77]. Two architectures are proposed and evaluated in the latter half of this paper. One is a simple ring structured architecture. The second is an extended version which is faster and more powerful. Both have been designed to execute computations based on LAPSE programs and they exhibit several desirable features which are discussed below.

The paper is intended to present an integrated view of three aspects of computation, namely a schema, a language-notation and an architecture. Several

features of the system can be found in the work referenced above. The whole scheme is a novel approach to designing a data flow computer system, and leads to a new architecture.

The graphical notation resembles both the "coloured" tokens of Dennis' procedural language [De74], and Arvind and Gostelow's unravelling interpreter [AG77, AG78]. The language LAPSE is similar to Id [AG78] and Lucid [AW77] (although, unlike Lucid, LAPSE allows recursion but has no equivalent to the as soon as operator). The architectures are superficially similar to those of Dennis and Misunas [DM75, DM77] and Arvind and Gostelow [AG77], but they exhibit fundamental differences to both of these pioneering proposals.

The proposed {schema, language, architecture} system has four main points in its favour. Firstly, the close relationship between the three components means that high level programs can be efficiently broken down into machine code. Secondly, the modular architectures are cheap and easy to maintain. Thirdly, the instruction execution rate of the extended architecture is not limited by the technology used : more throughput can be obtained by adding extra hardware[3]. Finally, the extended architecture can be made fault tolerant in respect of almost any detectable component failure : the faulty component can be isolated and service can be maintained with "gracefully" degraded performance.

2. GRAPHICAL NOTATION AND CONSTRUCTS IN 'LAPSE'

2.1. Basic Notation

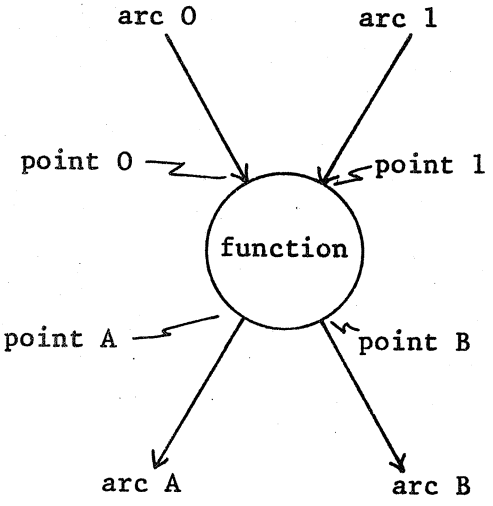
2.1.1. Computational Graphs

We noted in the introduction that data flow computations are usually represented by directed graphs. The nodes of a graph represent functions. They are connected to a number of incoming (input) and a number of outgoing (output) arcs. Each node has a fixed number of input and output points to which the arcs are attached. Arcs act as directed data paths which carry computed data values (tokens) between functions.

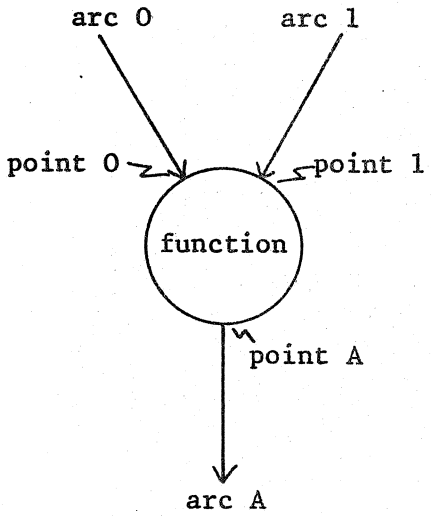
Normally an arc starts from an output point and is directed to an input point. Each input or output point is connected to exactly one arc. This latter convention is not restrictive since any graph can be converted into the appropriate form using functions such as DUPLICATE (one input, many outputs) and MERGE (many inputs, one output) whose actions will be described later.

There exist certain global input and global output arcs which are connected to an input or an output point only. These arcs carry data values to and from the graph as a whole and hence they represent input and output to and from the graphical program.

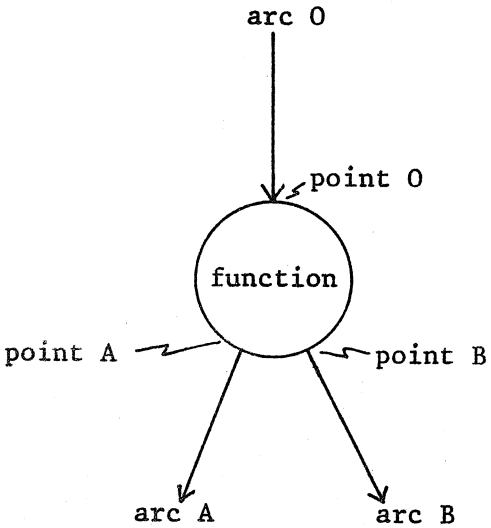
There are two kinds of nodal function known as primitive and compound functions. Primitive functions are basic machine operations such as "add two values" or "test whether value equals zero". Compound functions describe more complicated operations, but they can always be broken down into more detailed primitive graphs which contain only primitive nodes (primitives). In the following schema we consider primitives with at most two input and two output points and whose functions require at most one data value (token) from each input point[4]. The four basic kinds of primitive are called types A, B, C and D and are illustrated in figure 1.



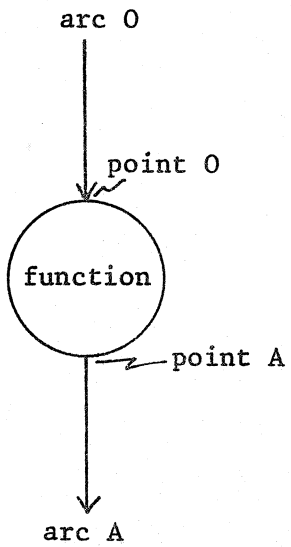
type A node



type B node



type C node



type D node

Figure 1 Primitive types of node.

The execution of all functions is expressed in terms of the execution of primitive graphs. This is governed by a set of firing rules which determine when a particular kind of primitive has sufficient input tokens available at its input points.

The firing rules vary for different functions. For example, "add two values" (type B) requires a token at both input points, while "test whether value equals zero" (type D) requires a token at input point 0 only.

Any primitive with sufficient input tokens available is eligible for execution (eligible). Its function can be executed any time that a suitable functional unit becomes available. Whenever the function is executed, the appropriate token at each input point is consumed. Output tokens will be produced later and placed on the appropriate output arcs (a maximum of one token per arc). All primitives that become eligible may be executed immediately and in parallel.

The state of execution (state) of an executing computation can be illustrated on a flowgraph, using black discs to represent tokens on arcs and shading of nodes to indicate that their function is being performed on some values. The value associated with a token is written alongside the relevant disc. For example, a possible history of the primitive node "add two values" (type B) is shown in figure 2.

An illustration of the state of a computation is known as a snapshot if it contains no shaded (i.e. executing) nodes. The progress of any computation can be described by a series of snapshots as we show later.

2.1.2. Simple Data Types

In LAPSE [G178], entities which can be represented in one computer word are treated as simple data types. At present only booleans, integers, and reals are considered. On a flowgraph it is possible to mark tokens representing simple data types with the type as well as the value, but this is

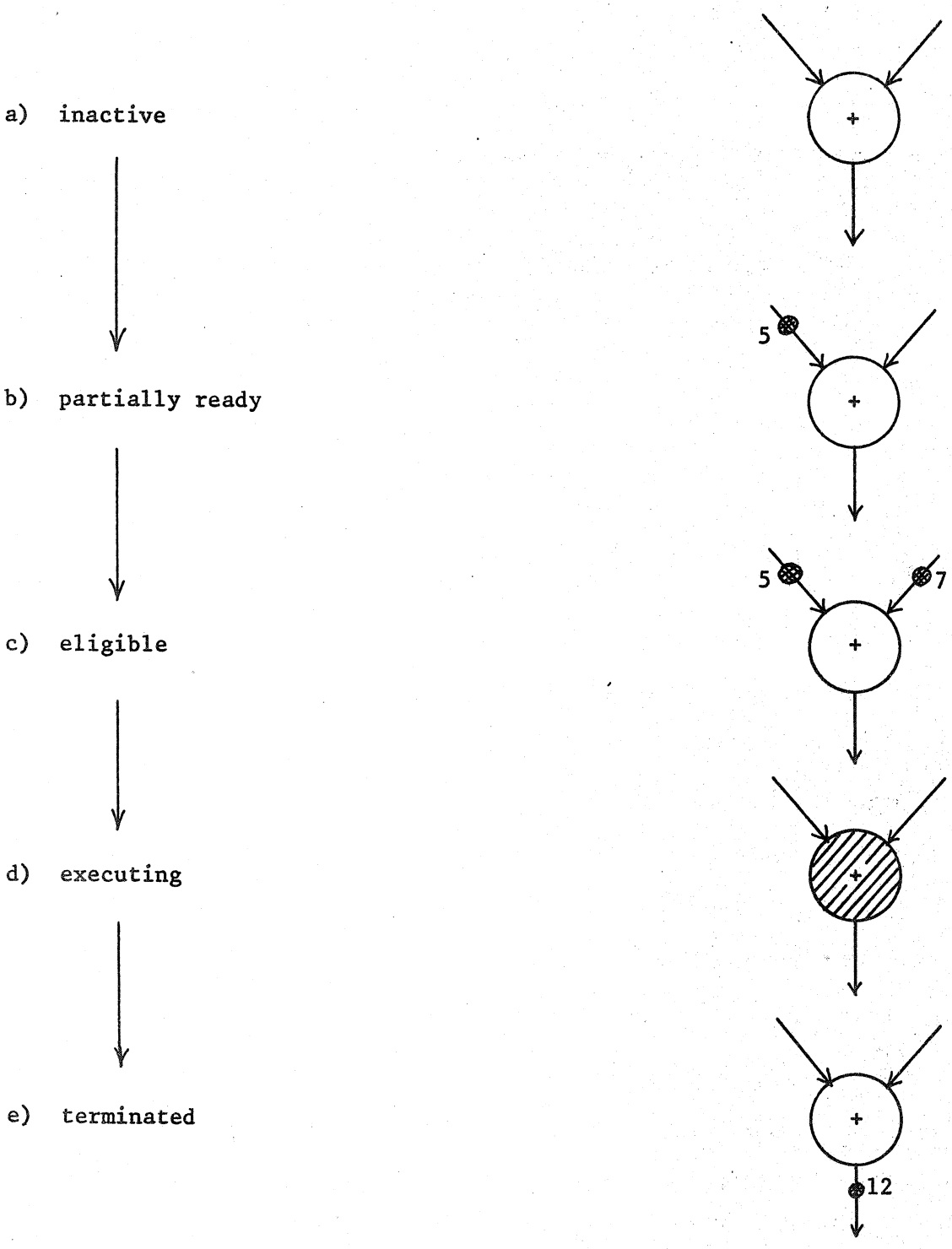


Figure 2 Execution states of a primitive type B node.

often unnecessary.

It is sometimes convenient to group values together, either because they are directly associated (e.g. two reals representing one complex) or because they are logically associated (e.g. the values returned from a multi-output function). To this end, LAPSE allows values to be grouped in record types which may be declared formally or used informally.

In a primitive graph, the record {A, B}, where A and B are of simple data types, represents two parallel arcs, named A and B, carrying tokens of the appropriate types. These ideas are illustrated further in the examples below.

In the remainder of this paper we shall use the LAPSE language to illustrate our discussion. It is appropriate to note two features of the language.

Firstly, the syntax of LAPSE is closely modelled on Pascal [JW74]. Secondly the unconventional nature of assignment should be noted. Identifiers correspond to arcs (or groups of arcs) on a flowgraph rather than storage locations containing variables. Thus a ← f(b,c) should read as "arc a will (eventually) receive the value obtained by performing function f on the values which will (eventually) arrive on arcs b and c" rather than "location a will (immediately) be assigned the value obtained by performing function f on the values which (currently) reside in locations b and c". The need for a single assignment rule (viz. assignment to an identifier can only be made at one point in a program) is apparent since an arc (with associated identifier) can be attached to just one output point. Programs written in LAPSE consequently exhibit a non-procedural (or assertional) flavour rather like Lucid [AW77] and Lisp [Mc65].

2.1.3. Complex Multiplication Example

As an example of the basic notation, consider a flowgraph program

which multiplies two complex values together. This can be expressed in LAPSE as follows :

```
type complex = record {re, im : real};
decl x,y,z : complex
function cmult (a, b : complex) : complex;
   begin
      cmult ← {(a.re * b.re) - (a.im * b.im),
               (a.re * b.im) + (a.im * b.re)}
   end;
begin
   z ← cmult (x,y)
end.
```

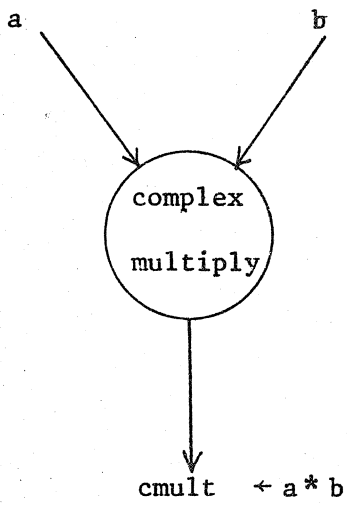
The function body (the assignment to cmult) can be drawn as a compound graph as shown in figure 3a. The corresponding primitive graph is shown in figure 3b[5]. Note the use of DUPLICATE primitives which reproduce tokens in order that a value can be used as an input to more than one node. Note also that each primitive in the graph has been assigned an identifying number (address).

When this function is processed, the precise order of execution of the primitives will be determined by the processing system. In a system with more than four processing units, the computation might proceed in three steps as shown by the snapshots of figure 4. Systems with less than four-way parallelism will take more than four steps. For example, the reader can readily verify that a single processor will take the equivalent of ten steps.

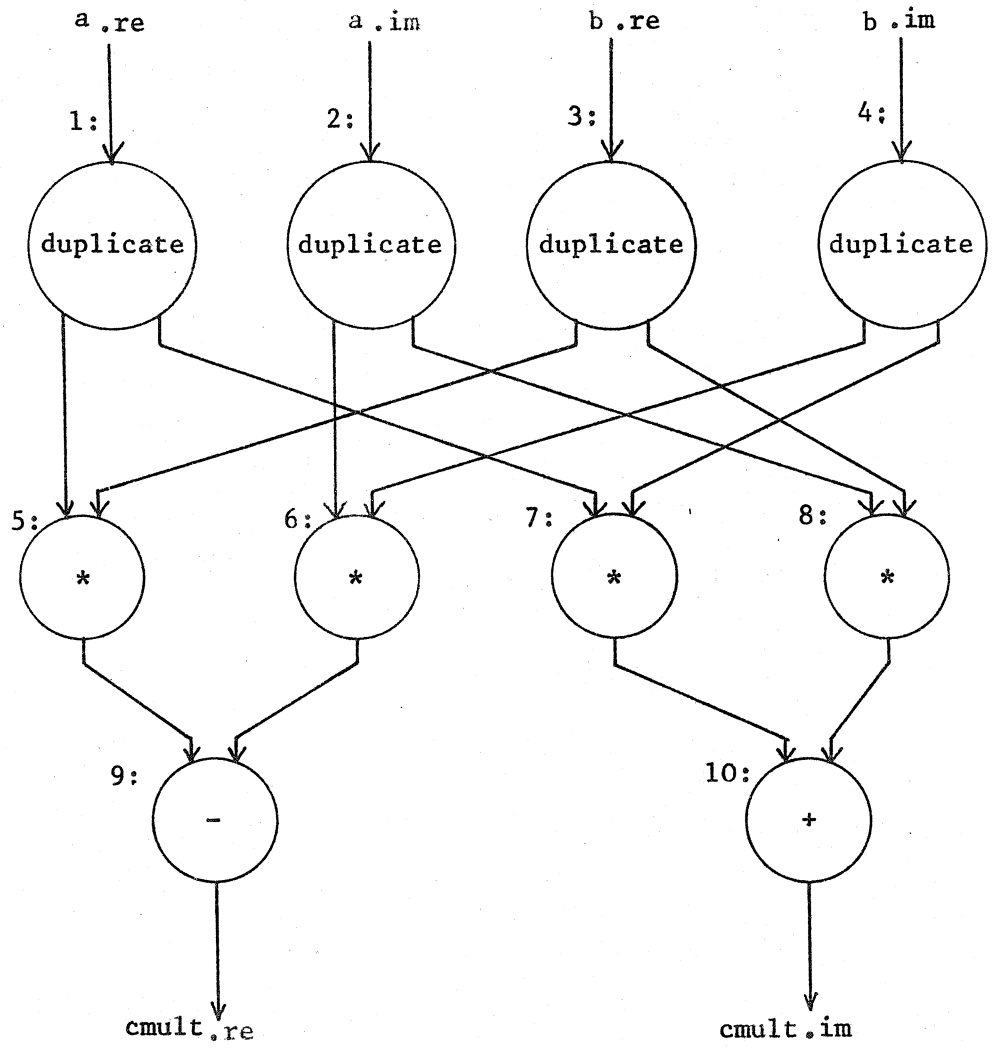
2.1.4. Parallelism in Computational Graphs

The discussion at the end of the last section naturally leads us to ask how much parallelism there is in any particular computational graph. This is a complex question to which we have devoted an appendix of this paper. A brief summary is appropriate at this point.

Following our earlier discussion of snapshots, we define the execu-

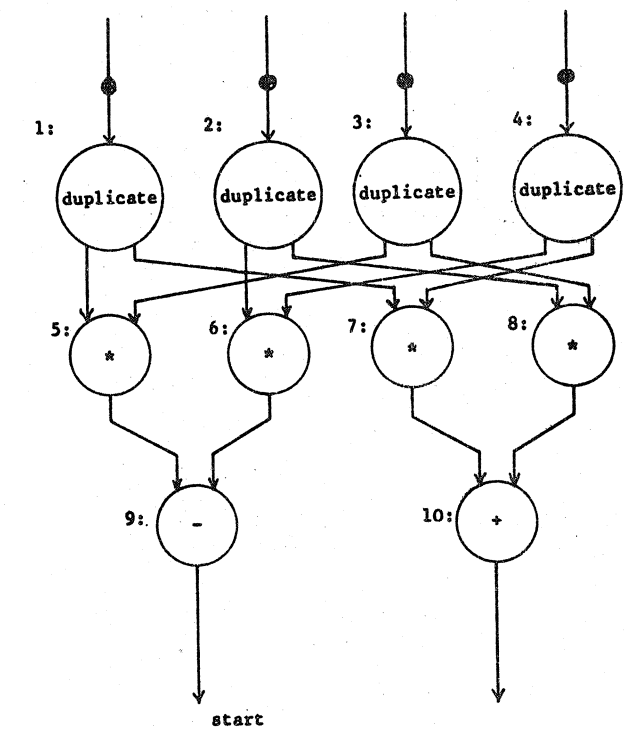


(a) compound function

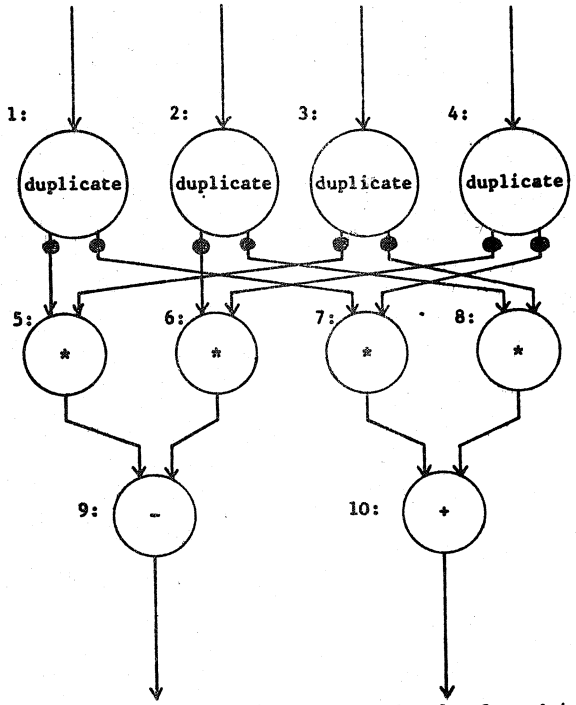


(b) primitive graph

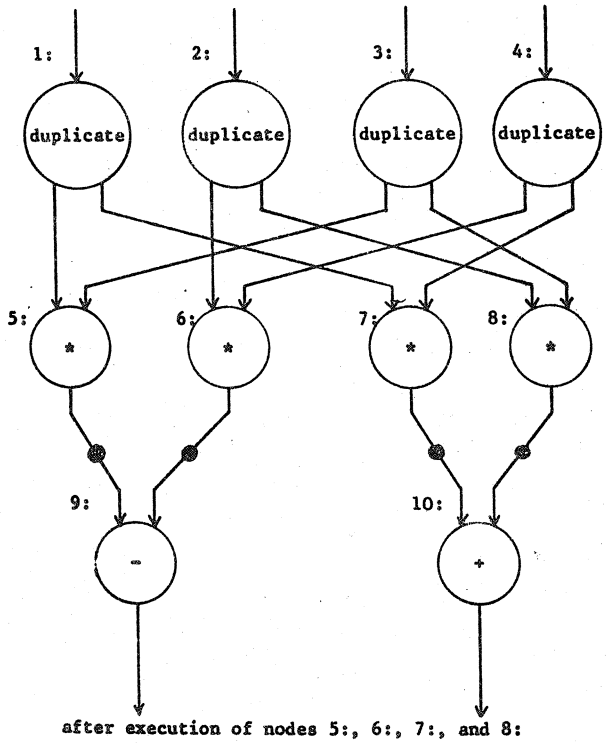
Figure 3 Two views of the complex multiplication function.



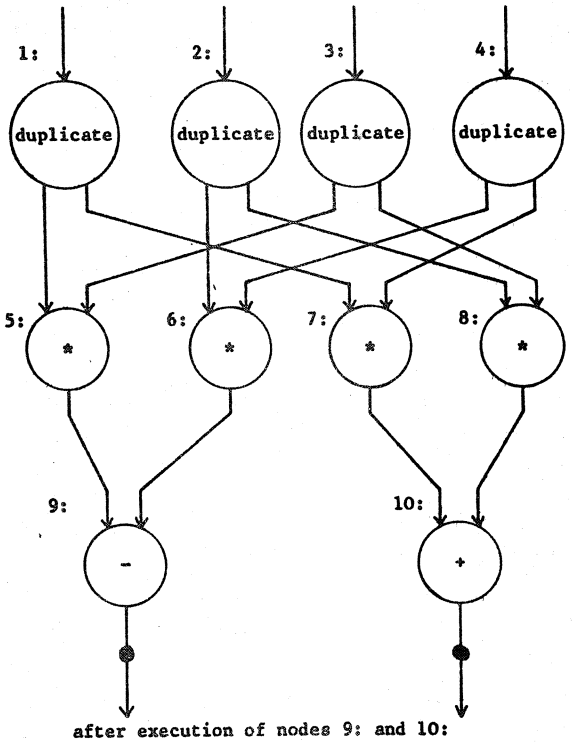
start



after execution of nodes 1:, 2:, 3: and 4:



after execution of nodes 5:, 6:, 7:, and 8:



after execution of nodes 9: and 10:

Figure 4 Snapshots illustrating an execution of the complex multiplication function.

tion time of a program S_k as an integer number of steps between snapshots which are derived by executing up to k eligible instructions from the previous snapshot. S_k can thus be regarded as the number of execution steps for the program if k "perfect" processors are used to perform the computation. By "perfect" we mean that :

(i) each processor takes a unit step time to perform an instruction;

(ii) all processors start and finish their instructions simultaneously; and

(iii) if e instructions are eligible at the finish of any step, then all e will be allocated immediately to a processor if $e \leq k$, or exactly k will be allocated if $e > k$.

Taking the complex multiplication program of figure 4 we have already seen that $S_4 = 3$ and $S_1 = 10$. We can quickly discover that $S_2 = 5$, $S_3 = 4$ and $S_k = 3$ for $k > 4$.

We next define the minimum and maximum values of S_k, as follows :

maximum : S_1 (the number of steps required by one processor : also equal to the total number of executed instructions).

minimum : S_∞ (the number of steps required if all eligible instructions are executed at each step : this may require an arbitrarily large number of processors during any step).

These values refer to the executed instructions and steps. In the following sections we describe some cases where the number of instructions executed is different from the number of nodes on the flowgraph. We use the maximum and minimum values of S_k to define the average parallelism Ψ of the (executed) flowgraph :

$$\Psi = \frac{S_1}{S_\infty}$$

The reader may feel intuitively that this measure of flowgraph parallelism is merely an abstract definition that does not distinguish between many different kinds of program with the same Ψ. However, we show in Appendix A that if we execute any program using $\Psi' = \lceil \Psi \rceil$ processors[6], then the speedup efficiency $\varepsilon_{\Psi'} = \frac{S_\infty}{S_{\Psi'}}$ is always greater than 0.5. We shall see later, when we study the performance of data flow computer architectures, that this is an important result.

2.2. Conditional Computation

Conditional computation in a data flowgraph requires conditional activation of one or the other (but not both) of a pair of subgraphs. If the program containing the conditional computation is to be well behaved then the two subgraphs must produce the same set of output tokens which can be merged together safely at the end. The natural language construct for this is the conditional expression in which either one value (or set of values) or another (but not both) is assigned to a simple data type (or informal record).

For example, the MAXIMUM function can be expressed in LAPSE as follows :

```
decl x,y,z : integer;
function max(a,b : integer) : integer;
   begin
      max ← if a > b then a else b fi
   end;
begin
   z ← max(x,y)
end.
```

The executed primitive graph for the function body is shown in figure 5. This illustrates the form of primitives used to implement conditional flow. The COMPARE function is used to derive a boolean value representing the required condition. The input values are directed by this boolean into either the true or the false subgraph by means of PASS-ON- <boolean state> functions. Finally the appropriate result value is obtained on one arc by the MERGE function[7].

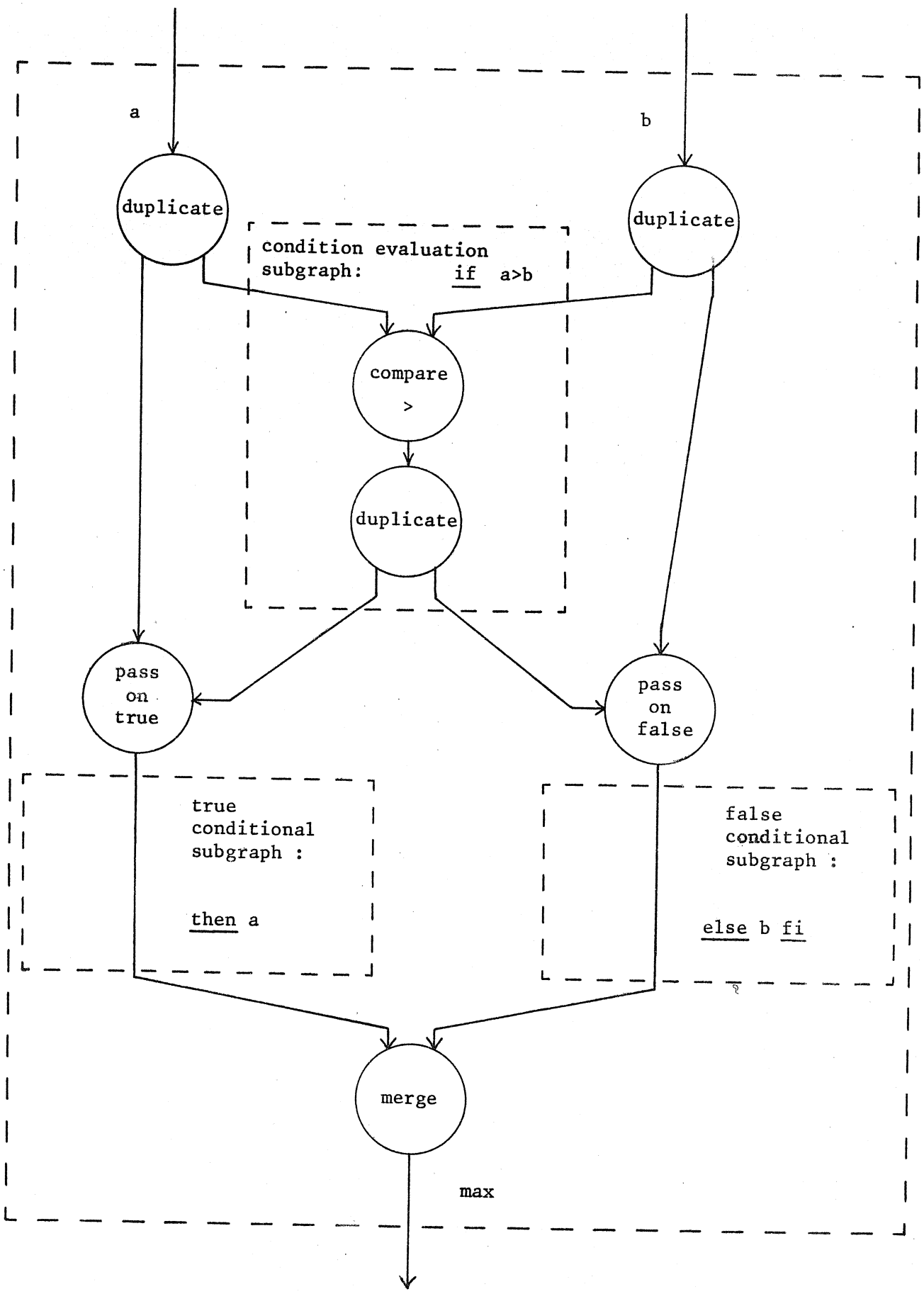


Figure 5 Primitive graph for the maximum function.

Note that a statement of the form a ← if b then c fi is illegal because some value must be assigned to the arc a after execution.

2.3. Iterative Computation

In order to implement iteration in data flowgraphs we need to use conditional flow around cyclic subgraphs. The cyclic subgraphs are reentrant; i.e. more than one token may pass along each arc in them. This is a potential source of confusion as we show below.

As an example, consider the evaluation of n! using an iterative alogorithm. This can be expressed in LAPSE as shown below[8].

```
decl n,facn,dummy : integer;
iteration factorial (i,f : integer);
   while
      old i > 1
   do
      {i,f} ← {old i-1, old i * old f}
   od;
begin
   {dummy,facn} ← factorial (n,1)
end.
```

The corresponding (executed) primitive graph is shown in figure 6. The special nodal functions employed are identical to those used for conditional flow. The MERGE nodes select either the initial input tokens or new values returned from the iteration body. The COMPARE function derives a boolean terminating condition, and the BRANCH function combines the function of a {PASS-ON-TRUE, PASS-ON-FALSE} pair, controlling the flow of tokens around or out-from the loop[9]. Note the use of the reserved identifier old to distinguish between the current and previously formed values of the parameters.

In this example, the inner loop which generates values for i may run faster than the parallel outer loop which generates values for f. This could lead to a situation in which successive values of i build up on arc A

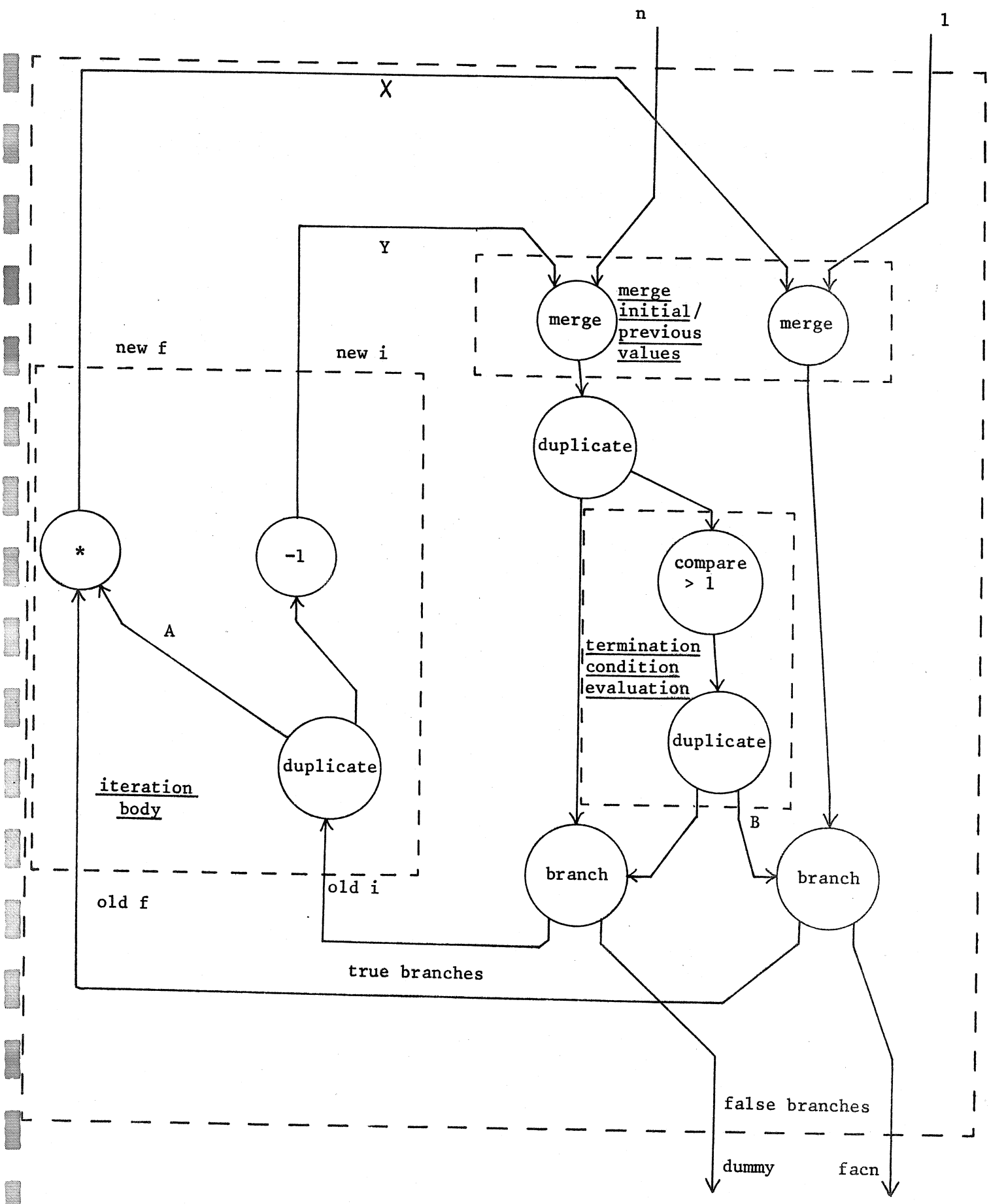


Figure 6 Primitive graph for the simple iterative factorial function.

(figure 6) and successive values of the boolean terminating condition build up on arc B, both waiting for the corresponding values of f to be formed and passed around the outer loop. Unless the values building up on an arc can be queued in first-in-first-out order, unpredictable actions may occur. For example, if the last boolean value (which must be false) "jumps" the queue on arc B, the iteration will terminate prematurely and in an incorrect state.

This situation is common to all iterations, but unfortunately it is difficult to design hardware which can guarantee correct queueing of tokens. Two solutions to the ensuing problem may be adopted.

The first solution requires more stringent firing rules for each node, designed to ensure that at most one token may ever lie on one arc[10]. This is the approach adopted by Dennis and Misunas [DM77], and it resembles the approach to a similar problem in the design of logical control networks which can be seen in Petri nets [Mi73]. The extended firing rules require that a node can only become eligible when it has no tokens on any of its output arcs. Implementing these extended rules requires that the receipt of an input token by an executing node should be acknowledged to the preceding node. This requires "backward" communication between nodes, and means that there is increased traffic along the hardware equivalent of arcs. There are also circumstances (such as in recursion) where it is essential to be able to execute an eligible (under the old rules) node even if it does have a token on an output arc. Since a mechanism is needed to allow this, it seems wasteful not to use a similar mechanism to allow any eligible (under the old rules) node to fire.

The second solution allows free and unambiguous firing of eligible nodes without specifically queueing tokens. This is the solution adopted by Arvind and Gostelow [AG77], and in the system described in this paper. Each token is

associated with a label, part of which is changed every time the iteration body has been performed. The altered part of the label is known as the iteration level of the token. It is an integer quantity. Any strategy can be used to change it, but the simplest method is to increment it once every iteration[11]. In order to execute instructions in the intended order we arrange to match pairs of tokens arriving at a node according to their labels. Thus we extend the firing rules to ensure that only those nodes which have available the required number of input tokens with matching labels become eligible.

2.4. Compound Functions

We have already discovered that the function construct in LAPSE is convenient for representing compound functions. However, so far we have glossed over the graphical representation of functions. We have drawn primitive graphs which perform the compound functions required but do not show how to call or exit from the function.

Of course, it is possible to make each call of a function to a separate copy of the code for the function body. Such a scheme is used by Miranker [Mi77] to implement procedures in the system proposed by Dennis and Misunas [DM75]. Each procedure call creates a new copy of the code for the procedure body and then transfers all incoming tokens to the new graph. When the outgoing tokens arrive at the end of the procedure body they are transferred back to the appropriate part of the original graph, and the "dead" copy of the procedure body is destroyed.

This implementation becomes inefficient for large functions because of the duplication of functional code. In LAPSE we do not copy any code but use labels to distinguish calls to a function. Hence the implementation of functions in LAPSE is analogous to the implementation of procedures in Algol-like languages using a stack : the label mimics the role of a conventional

namebase register.

By convention, a LAPSE function consumes some number of input arguments and returns just one output argument. Any argument may be a record so that, for example, more than one output value may be returned. The general form of a function call is illustrated in figure 7.

The code comprising the function body and output interface is written just once for all calls. The input interface is specific to each call.

The function body may be activated simultaneously by more than one call, either from the same point (recursion) or from different points (parallel use of a compound function) of a flowgraph. Tokens belonging to different activations of the function are distinguished by changing a part of their label known as the identifier. The change is performed for each incoming token at the input interface and has to be reversed at the output interface before the output argument can be returned[12]. The output argument must also be returned to the intended part of the flowgraph. To this end, a link consisting of the original identifier and the address to which the output argument is to be sent is transmitted from the input interface to the output interface. The output interface restores the original identifier to the output argument and forms a dynamic arc (dotted in figure 7) which directs the output appropriately. Note that dynamic arcs break a rule of the basic notation (each arc should be attached to only one input point and one output point). This is unavoidable in the circumstances : all data flow schemata which support compound functions employ an equivalent mechanism. As long as the return address is fixed for each call at the appropriate input interface there will be no problems. The mechanism mimics the generation of a separate copy of the function code inserted at the appropriate point in the graph.

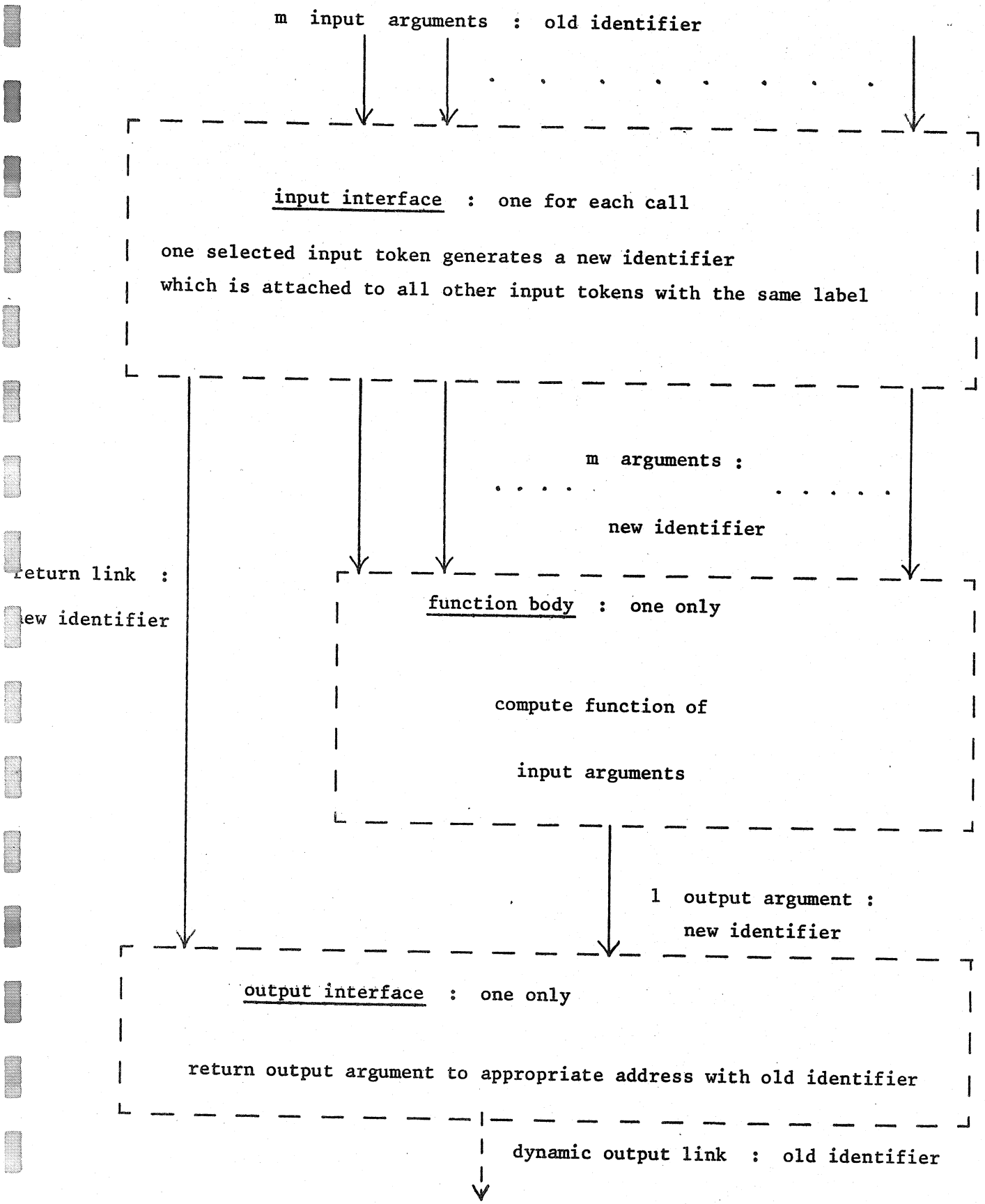


Figure 7 General form of a function call.

As an example of the actual code produced, the complex multiply program of 2.1.3 and figure 3 is redrawn in figure 8. The input interface (program body) has been separated from the function body and the output interface. Each token has its identifier (id) written alongside. Note the use of a trigger token which is sent to all activations of a function.

The trigger is firstly used to generate a new identifier for all input tokens via a GENERATE-IDENTIFIER node. This node cannot merely increment the incoming identifier because that would fail to distinguish parallel activations of the function. Hence it generates a brand new value of identifier.

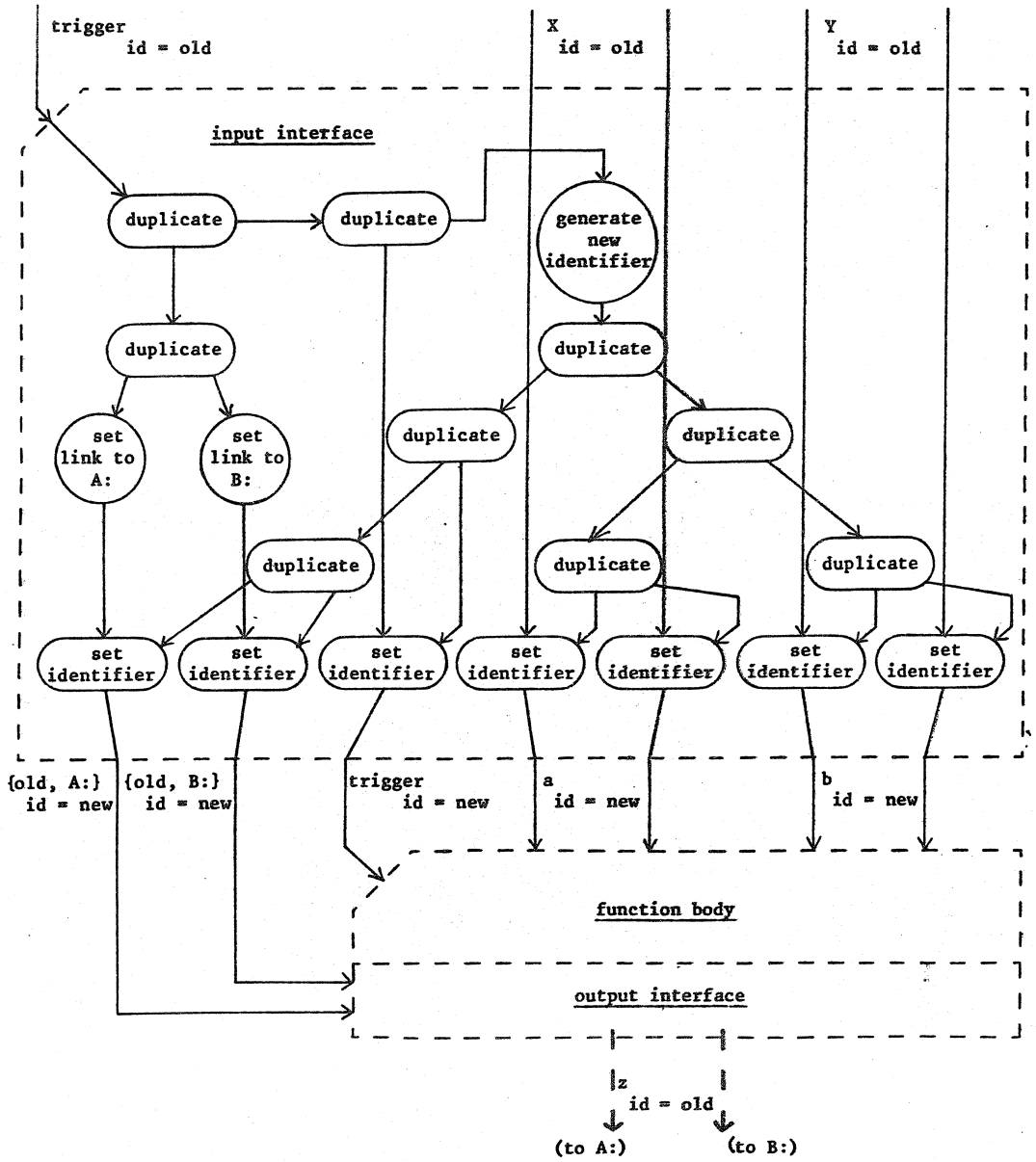
The trigger also generates the return links via SET-LINK nodes[13]. The output of these nodes are {address, identifier} pairs for which each address is the return point for an output value (specified as a literal) and the identifier is the original input identifier (same for each node).

The new identifier is attached to all the input arguments and the return links by SET-IDENTIFIER nodes. The function body can now be entered safely. Note that the trigger is passed into the function body to trigger any internal functions. It can also be used to trigger constant values used with the body.

The old identifier and return addresses wait in the output interface until output values have been formed. The RETURN nodes then restore the external identifier and direct the values to the destination nodes A: and B: .

2.5. Recursive Computation

Given the means for activating separately executable functions, we can implement recursion. For example, using the mechanism described in the previous section, we implement the following program which evalues n! as shown in figure 9:



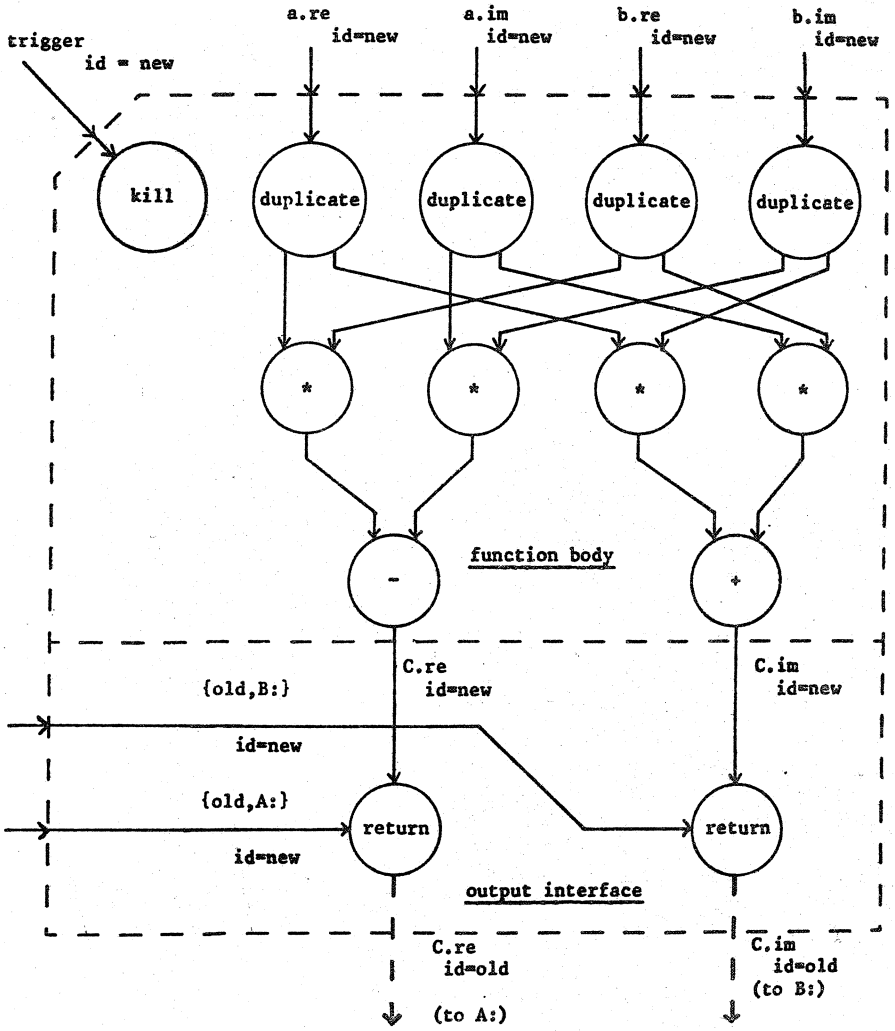


Figure 8 Expanded code for the complex multiplication function.

```
decl n,facn : integer
function fac(j : integer) : integer;
   begin
      fac←if j>1 then j*fac(j-1) else 1 fi
   end;
begin
   facn←fac(n)
end.
```

Note that the trigger passed into the function body in this case is used to trigger the recursive call.

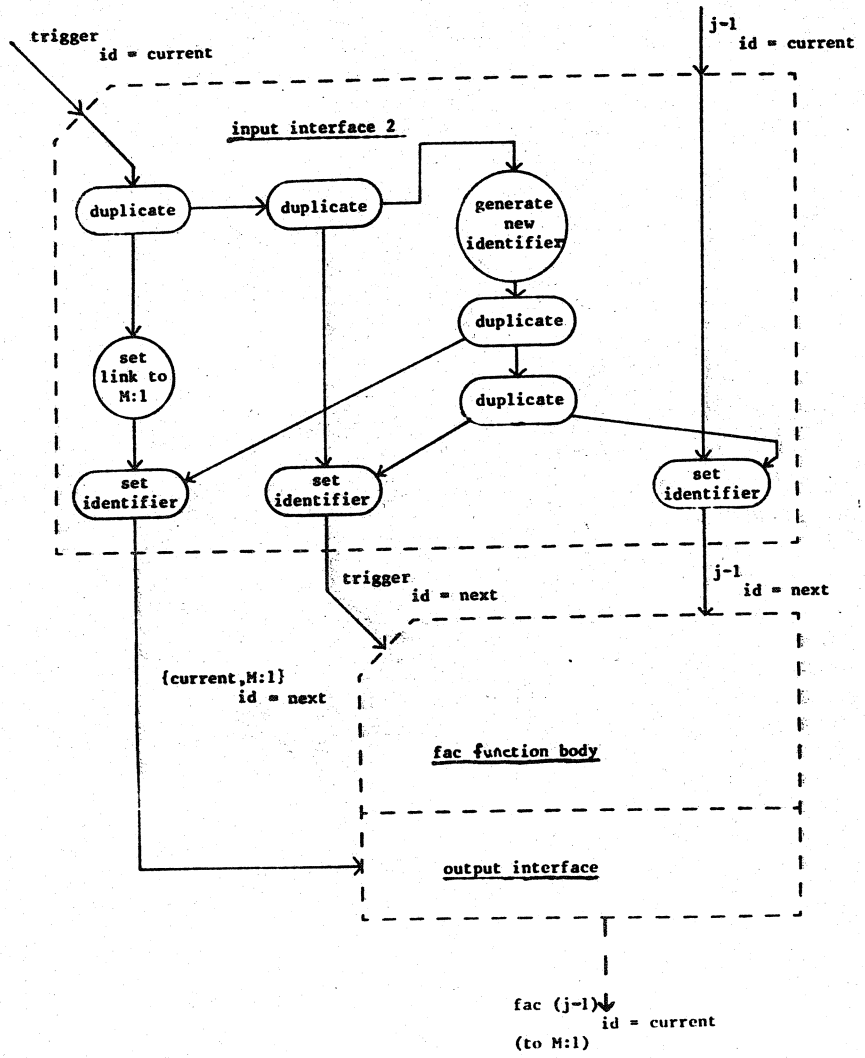
It is also possible use double recursion [St73]. For example, the following program is illustrated in figure 10 :

```
decl n, facn : integer;
function dblefac (x,y : integer) : integer;
   begin
      dblefac←if x≤y then y else
           dblefac (x, (x+y) div 2 + 1) *
              dblefac ((x+y) div 2,  y) fi
   end;
begin
   facn←dblefac (n,1)
end.
```

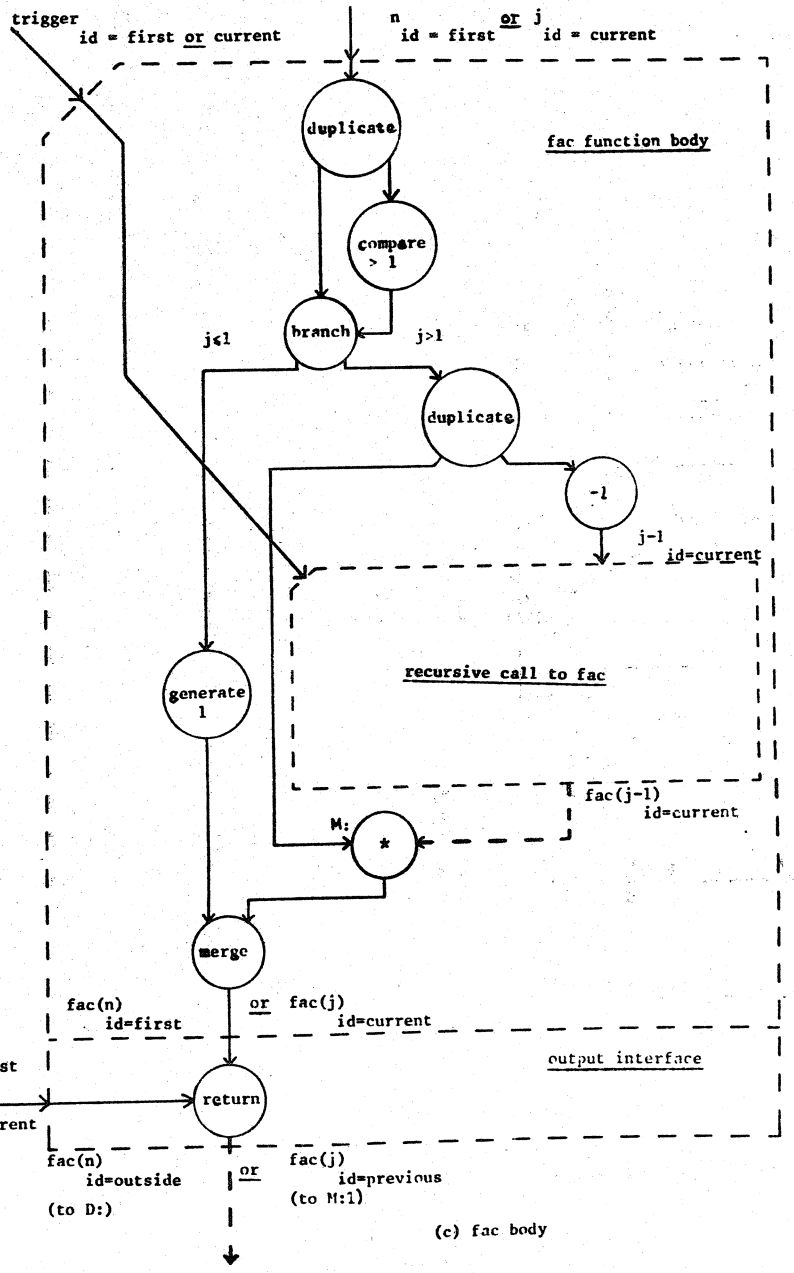
Because of the pair of recursions repeatedly applied during each activation of the function, this algorithm can be used by a highly parallel computer to evaluate n! in $O(\log_2 n)$ time[14].

2.6. Structured Data Types

Apart from records, we have not discussed methods for handling data structures. It is characteristic of many such structures that the programmer wishes to perform a common operation throughout the structure. This suggests that labels might be useful for distinguishing elements of a structure which can all be sent down the same arcs in order to have a common function performed on them. In LAPSE the array structure is implemented in

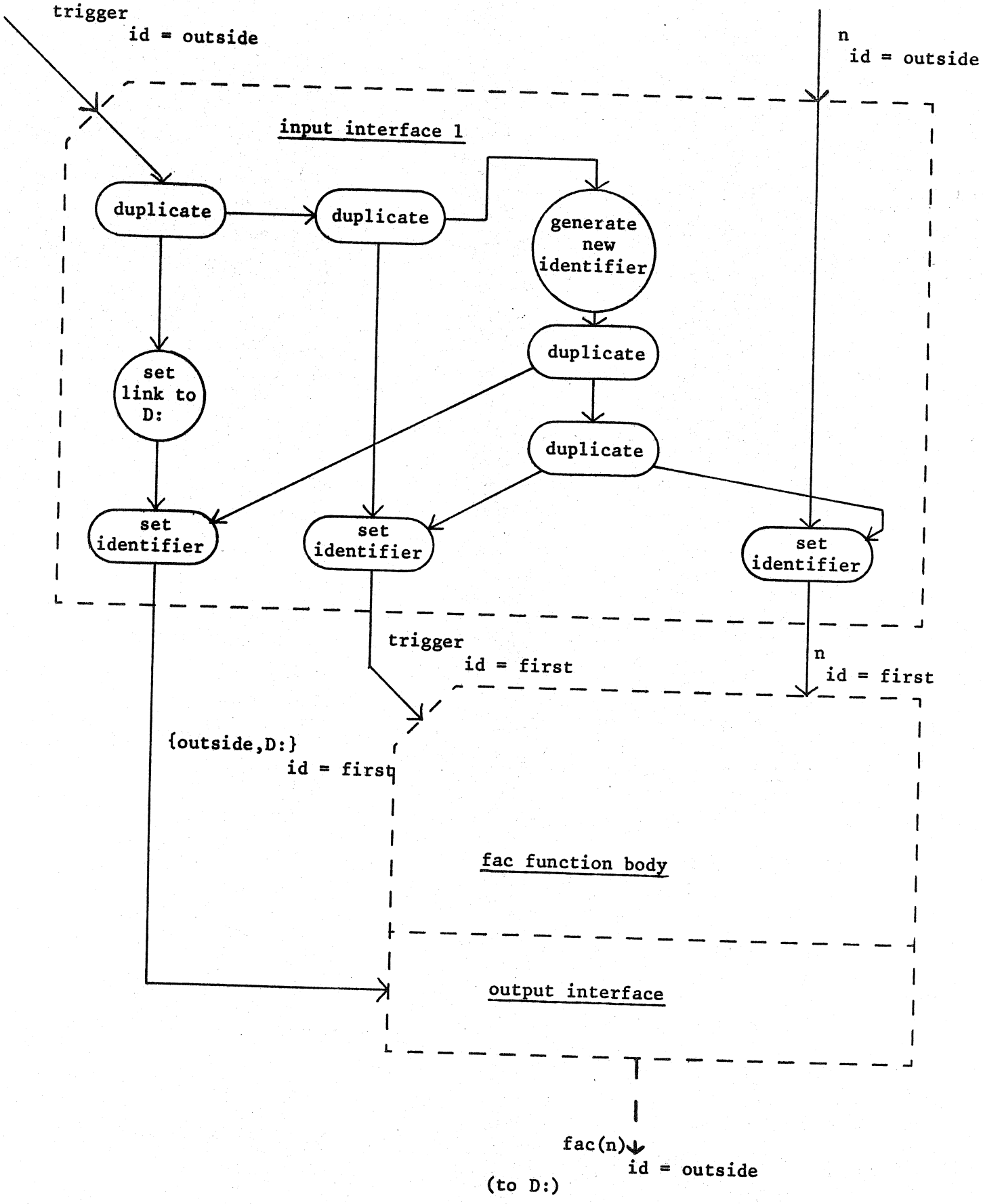


(b) internal (recursive) call to fac.



(c) fac body

Figure 9 continued



(a) external call to fac.

Figure 9 Primitive graph for the single recursive factorial function.

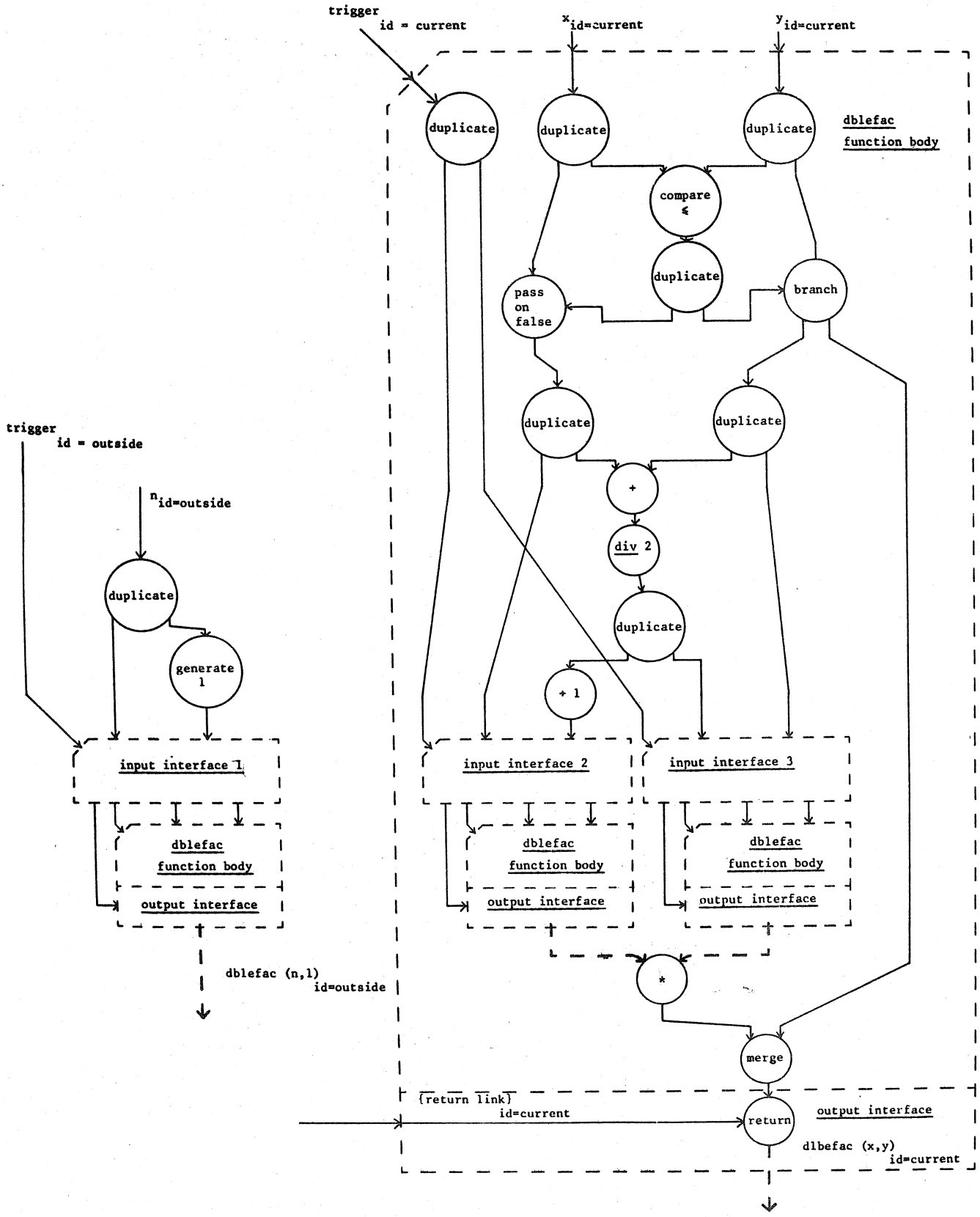


Figure 10 Neo-primitive graph for the double recursive factorial function.

this fashion.

Part of each label (the index) is used to hold the index of each element in an array. The index must lie between specified bounds and is defined in a Pascal-like manner in LAPSE. Assignment to an array is specified by the for all .... use .... construct. Operations may be parallel or serial in nature.

An example of a fully parallel operation is the element-wise addition of two arrays. This can be specified in LAPSE as follows :

```
decl  a,b,c : array [1 .. n] of integer;
begin
   c[i] ← for all i use a[i] + b[i]
end.
```

The whole program can be implemented as a single primitive, as shown in figure 11. Note that each token in this diagram has its index (ix) value written alongside its data value. Although the tokens of the array are drawn in order on the arcs, they cannot be regarded as in any sense being in order. Arcs do not act as first-in-first-out queues, nor does the ADD node take a determinate time to execute each time it is fired. There is no constraint on the order in which the additions take place[15].

As an example of an array operation in which sequence is important, consider the program below which generates an array of the first n Fibonacci numbers :

```
decl  f : array [1 .. n] of integer;
begin
   f[i] ← for all i use if i≤2 then i else f[i-2] + f[i-1] fi
end.
```

The code for this is illustrated in figure 12. Because action is conditional upon the index values, COMPARE-INDEX-with-integer primitives are used to evaluate boolean conditions. The sequence follows the production of the required data, not an arbitrarily programmed path.

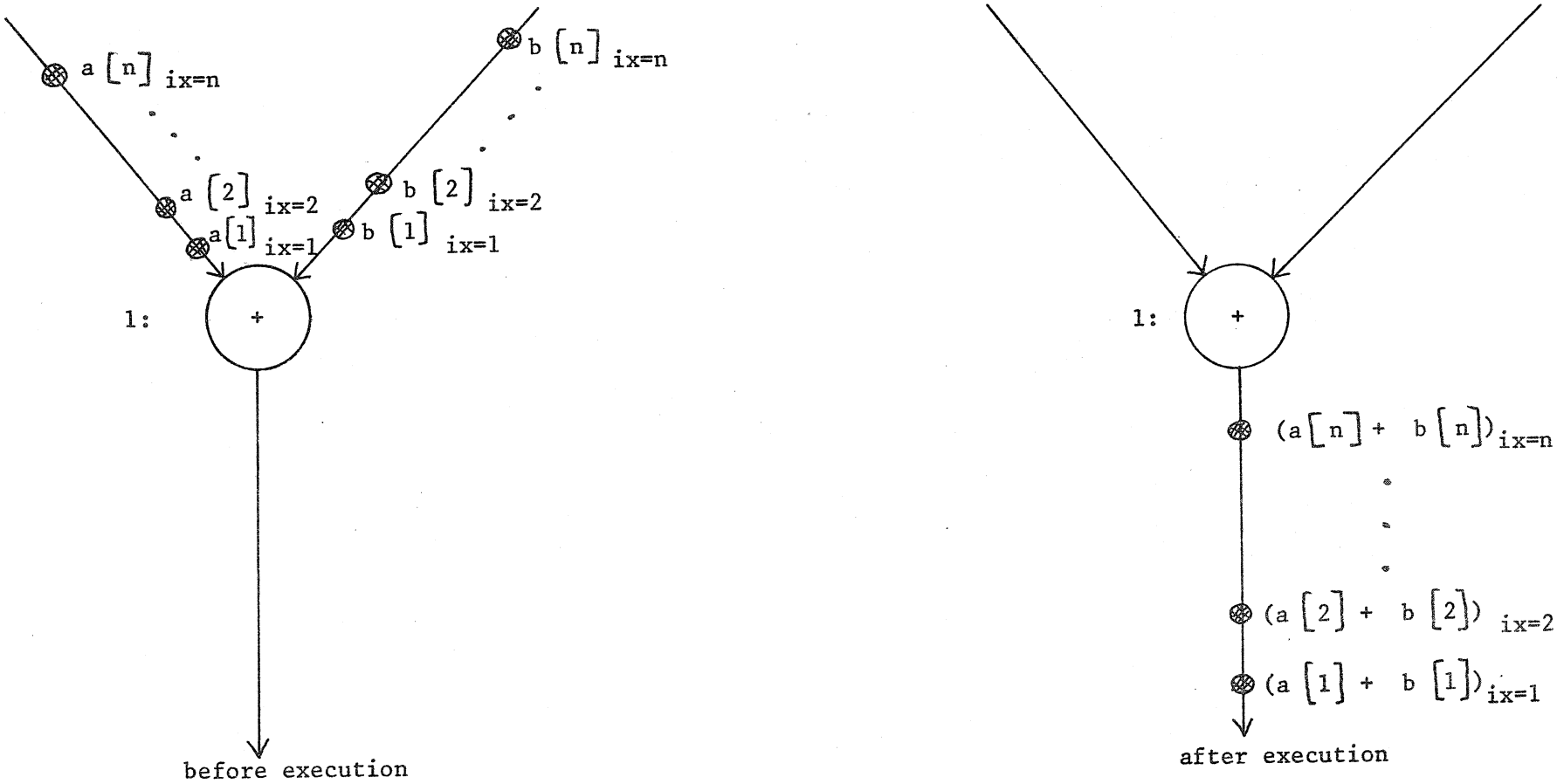


Figure 11 Addition of two arrays, a [1..n] + b [1..n] .

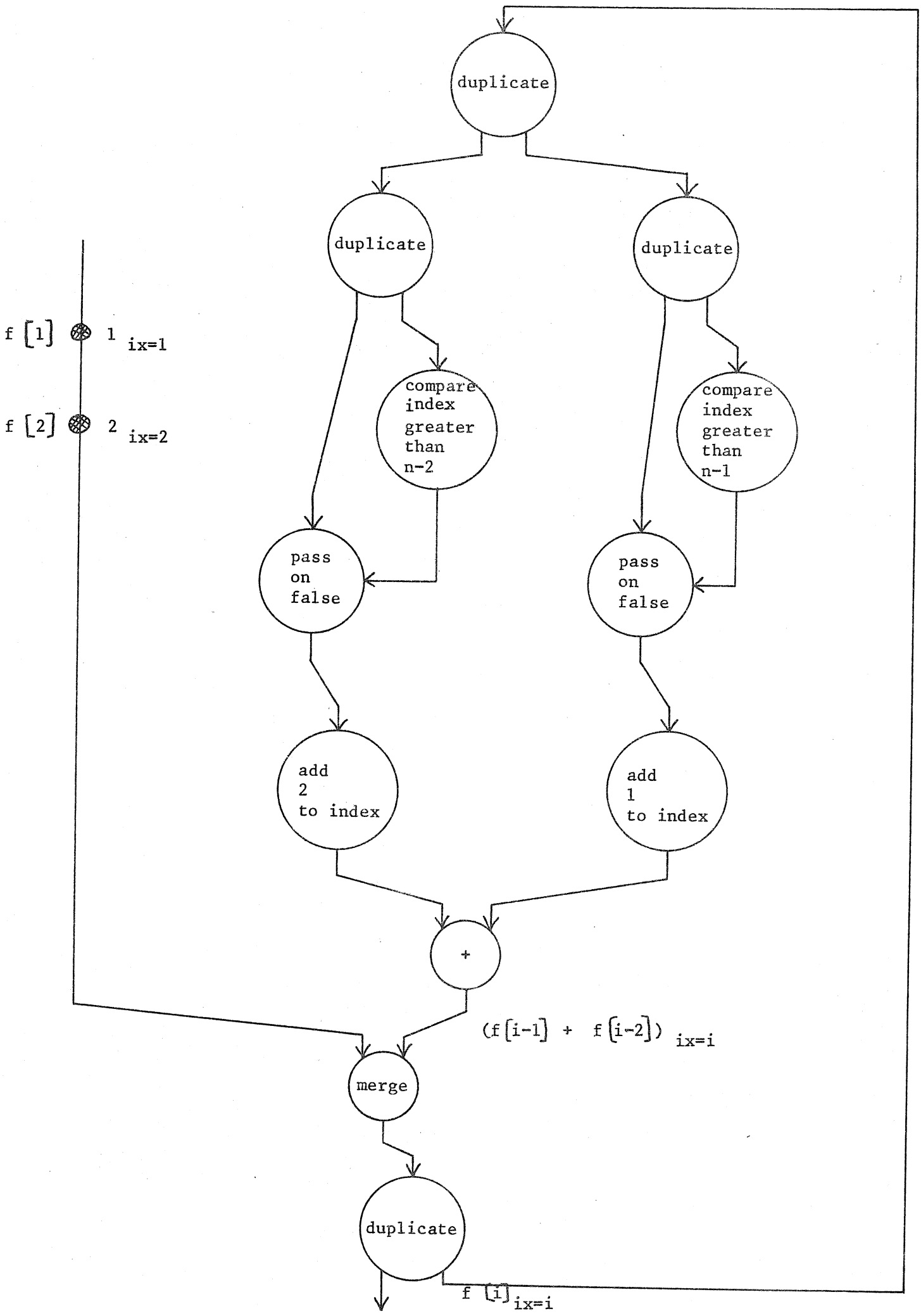


Figure 12 Primitive graph for Fibonacci program.

Finally consider a program in which some sequence must be specified, but where overspecification of sequence is costly in time. One such example is the summation of the elements of an array, which should be possible in time O(log(number of elements)). A program for this function which uses several of the constructs described above follows[16] :

```
decl a,b : array [1 .. n] of integer;
   result, dummy : integer;
iteration sum (k : integer ; s : array [1 .. n] of integer);
   while k > 1
   do
      k ← (old + 1) div 2;
      s ← for i
          use
             if i < k then old s [2*i-1] + old s [2*i]
             elif (i=k) and (odd (old k)) then old s [old k]
             else ∅
             fi
   od;
begin
   {dummy, b} ← sum (n,a);
    result ← b [1]
end.
```

We are making further investigation of the implementation of data structures other than the array. Other research in this area includes that of Weng, who describes recursive computation on streams [We75], and Dennis and Misunas, who describe structure processing [De73, Mi 75].

2.6. Summary

We have developed a basic notation for graphical computation and demonstrated that this can be used to implement the constructs of a high level, single assignment language known as LAPSE. The notation uses labels which distinguish tokens in reentrant subgraphs. The labels are segmented in a fashion which recognises three sources of reentrancy, namely iteration,

recursion and data structures. The power of the language and notation has been demonstrated by examples of commonly encountered programs.

Several unnecessary constraints were imposed on the notation. For example, a maximum of two inputs and two outputs was specified. In the next sections, we demonstrate how these constraints make it possible to design efficient, high speed architectures for executing data flow computations.

3. A RING STRUCTURED DATA FLOW COMPUTER

3.1. Ring Structured Architecture

A simple data flow computer comprises five units arranged in a ring-structured pipeline as shown in figure 13.

The computer representation of a primitive flowgraph is a set of instructions held in the instruction store. An instruction contains the description of a node : i.e. its function and the destination of its output arcs. The format of an instruction is shown in figure 14a. The function field contains a description of the operation the node performs, and optionally, a literal value for use in certain type C and D functions. Each destination field specifies an instruction address (i.e. the address of the next node), an input point (0 or 1) on which the arc is incident at that address, and an indication of the number of operands (1 or 2) expected at the destination.

Tokens are represented by results which comprise an operand (data value) together with its label and its destination. Since tokens always reside on arcs, it is convenient to associate the destination of an arc with its token rather than vice versa. The label and destination of a result will later be considered as a single field known as the name. The format of a result is shown in figure 14c. Note that the data value in the operand is tagged with its type. The system performs some type-checking using this field.

Results are either generated in the processing unit or they are transmitted into the system from the outside world. The input/output switch accepts both kinds of result, and will permit some results to be transmitted out from the system. By convention, results with negative instruction address fields are assumed to contain output values.

Results can be stored in either the result queue or the matching

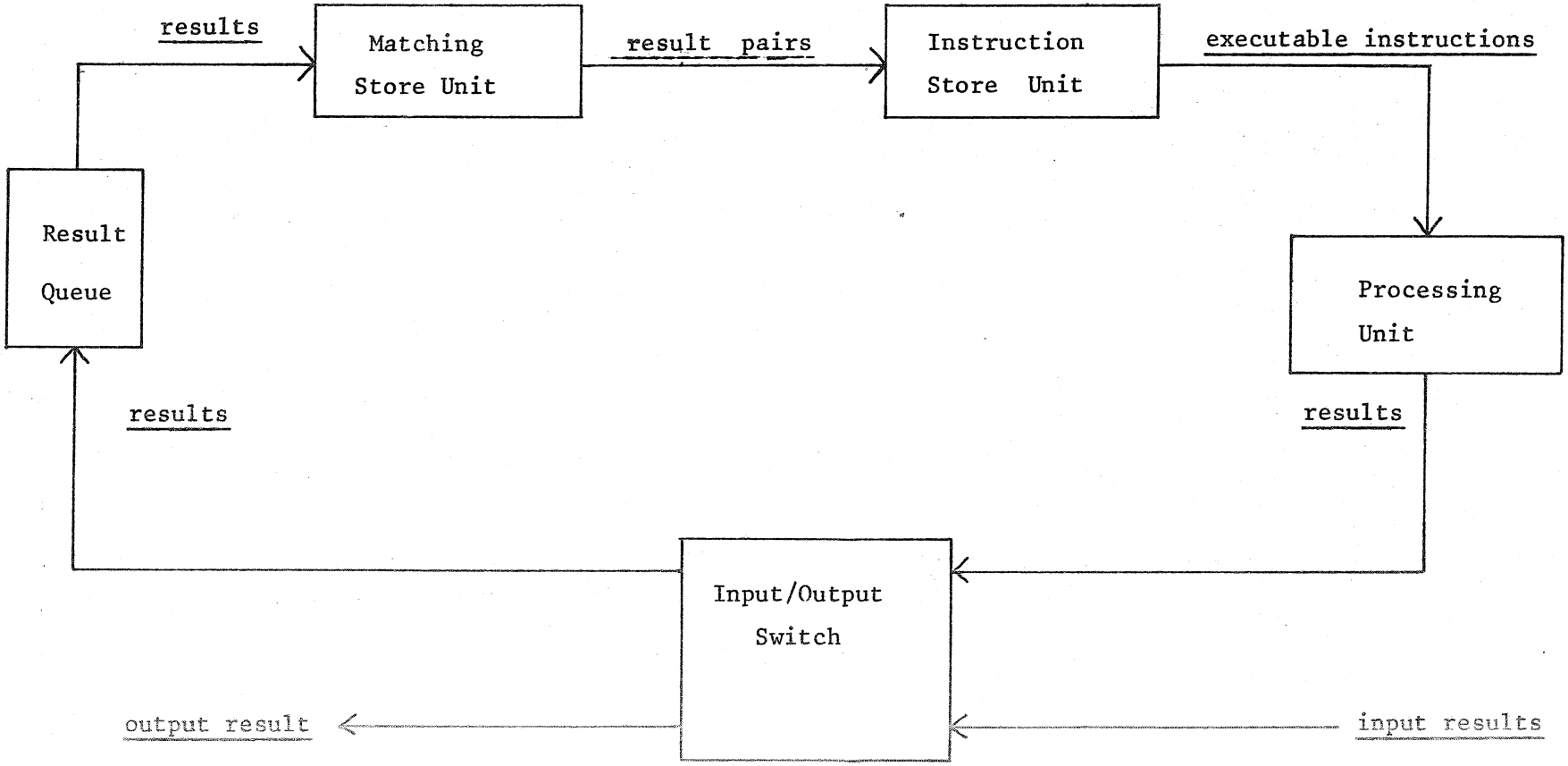


Figure 13 A simple ring structured architecture for a data flow computer.

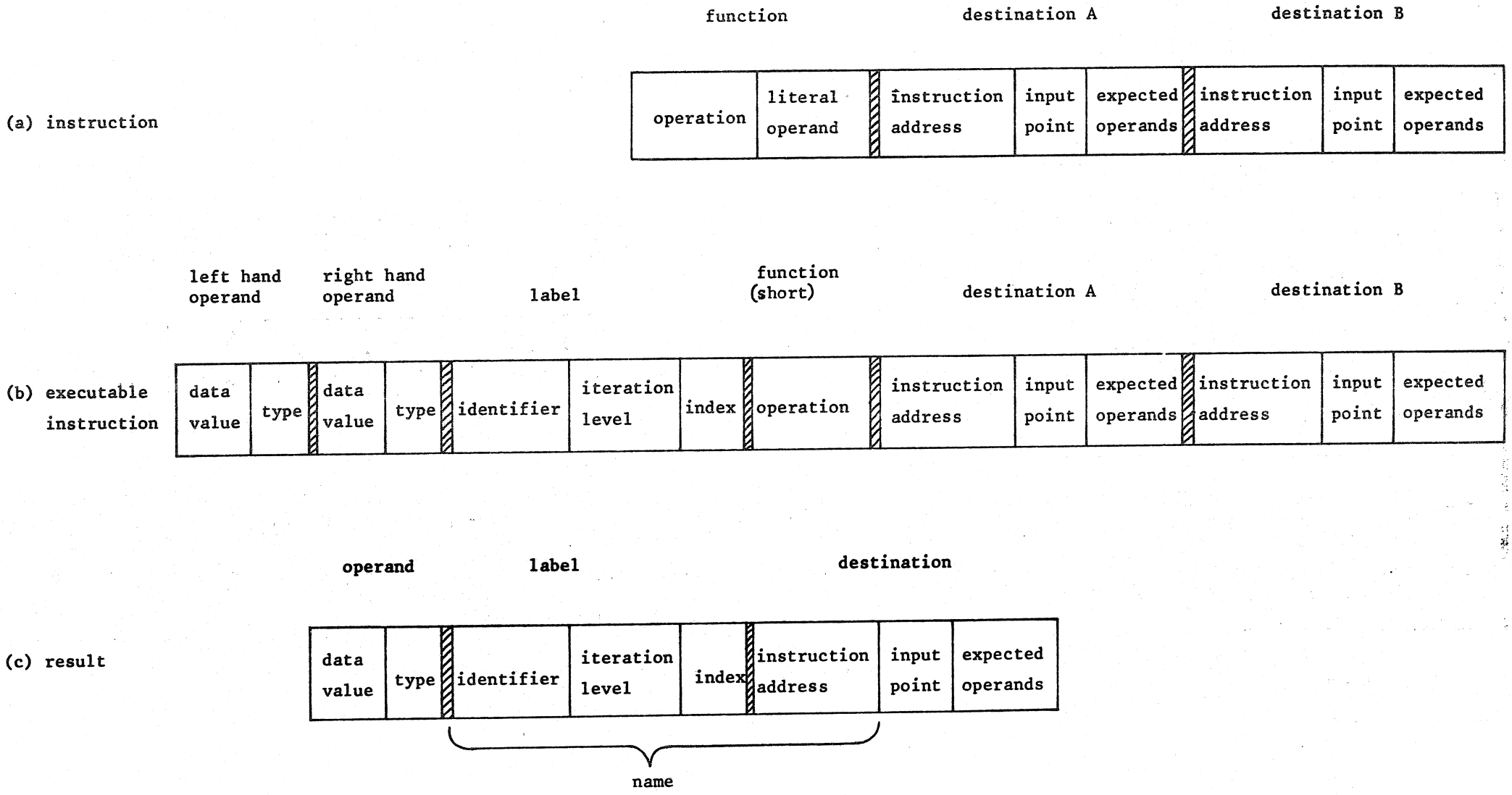


Figure 14 Formats of information items in the data flow computer.

store. The result queue is a temporary buffer whose purpose is to synchronise disparate rates of production and consumption of results. It need not behave as a first-in-first-out queue. The matching store is associatively accessed by the name of stored results. Its operation is central to the working of the system and will be described later.

The matching store and the instruction store together generate executable instructions. The details of this operation are given in the next section. The format of an executable instruction is shown in figure 14b. It comprises two input operands with a common label, an operation to be performed, and up to two destinations for the result(s) of performing the operation on the operands.

The processing unit consumes each executable instruction, then performs the specified operation, yielding zero, one or two results at the input/output switch.

3.2. Operation

The ring structured architecture executes programs in two phases known as the initialisation phase and the execution phase.

During initialisation, three tasks are performed. Firstly, the result queue and matching store are emptied. Secondly, the instructions representing the computational flowgraph are stored in appropriate locations of the instruction store. Thirdly, a set of global input results (corresponding to tokens on the global input arcs of the flowgraph) are made available at the input/output switch.

The initialisation for the complex multiplication function of section 2.1.3 and figure 3 is illustrated by tables 1 and 2. Table 1 shows the instruction store layout and table 2 lists the global input results. Since this program does not alter the labels of any tokens, the fields of each label are set to undefined but common values.

	function		destination A			destination B		
instruction store address	operation	literal operand	instruction address	input point	expected operands	instruction address	input point	expected operands
1	duplicate		5	0	2	7	0	2
2	duplicate		6	0	2	8	0	2
3	duplicate		5	1	2	8	1	2
4	duplicate		6	1	2	7	1	2
5	multiply		9	0	2			
6	multiply		9	1	2			
7	multiply		10	0	2			
8	multiply		10	1	2			
9	subtract		-1	(output)				
10	add		-2	(output)				

Table 1 Instructions for the complex multiplication program of figure 3(b).

operand		label			destination		
data value	type	identifier	iteration level	index	instruction address	input point	expected operands
x.re	real	id	i1	ix	1	0	1
x.im	real	id	i1	ix	2	0	1
y.re	real	id	i1	ix	3	0	1
y.im	real	id	i1	ix	4	0	1

Table 2 Global input results for the complex multiplication program of figure 3(b).

At the start of the execution phase, the global input results enter the system through the input/output switch and begin to build up in the result queue. The operation of the system is thereafter determined by the matching store/instruction store stages. The cycle of figure 15 is performed continuously until the processing unit is idle and there are no results in the result queue. At this stage all the required output values will have been directed to negative instruction addresses (i.e. out of the system). It is thus possible to commence initialisation of another program.

The operation of the matching store deserves some attention. It can be seen from figure15 that type C and D instructions (i.e. those expecting only one token) bypass the matching store. For type A and B instructions there is at most one entry in the matching store for each name. This follows from the three conditions governing matching store behaviour :

(i) the store is empty at the start of the execution phase;

(ii) each unmatched access results in an entry being written; and

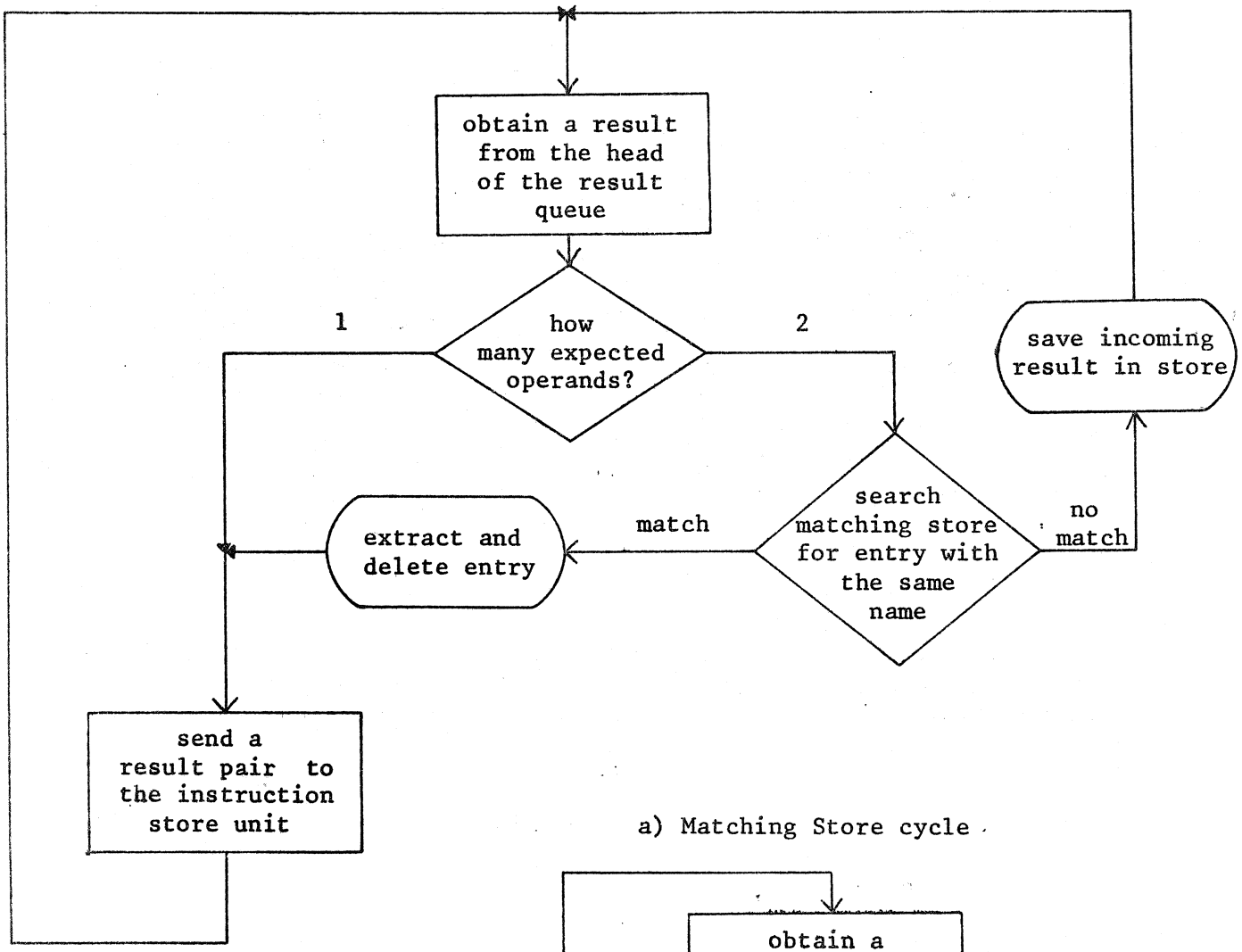
(iii) each matched access results in the matched entry being deleted.

An important class of programs, known as well-formed flowgraphs, leave no entries in the matching store at the end of the execution phase. Code generated from LAPSE programs is well-formed[17].

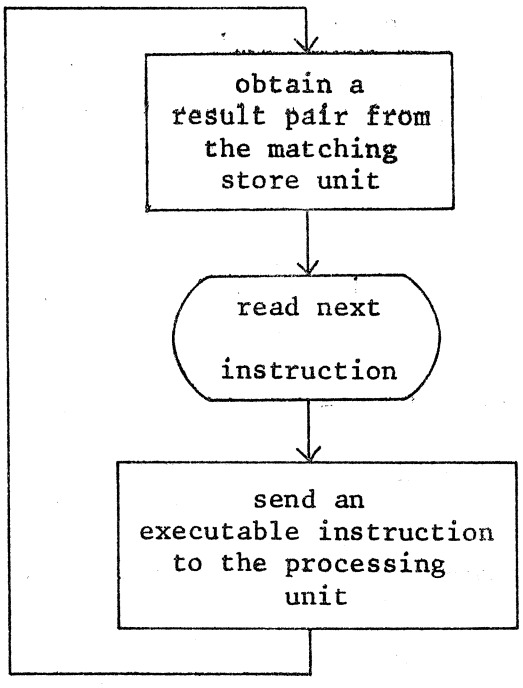
For well-formed programs it is possible to implement multiprogramming by simply adding a {process number} field to each label. There will be no need for context switching since each flowgraph is isolated.

3.3. Implementation

In this section we will consider a possible implementation of the ring structured system in some detail. In order to achieve this we introduce a notation for describing modular hardware components and their interconnection. This will facilitate the evaluation of the system which is pre-



a) Matching Store cycle



b) Instruction Store cycle

Figure 15 **Flowcharts showing the operation of the matching and instruction store units during the execution phase.**

sented in the next section.

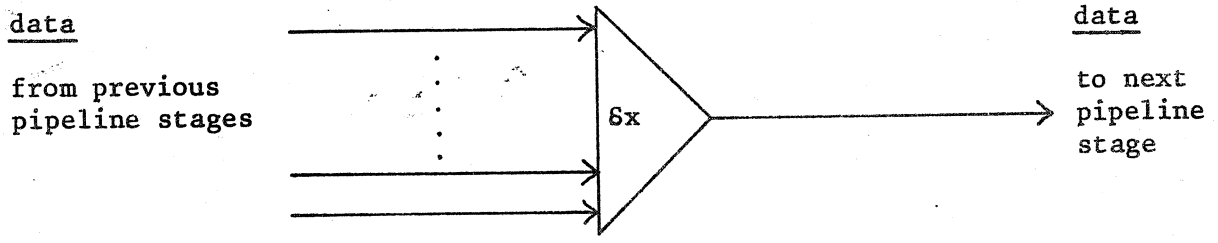
The ring structure is based on a circular pipeline. The notation thus recognises pipeline control units as well as resource modules. The five basic module types are illustrated in figure 16.

The arbitrator and distributor stages generate and terminate parallel actions in the pipeline. A resource accessor stage is tied to a particular resource at the appropriate point of the pipeline. Note that a minimum delay time is associated with each pipeline stage and the type of data unit is associated with input and output lines.

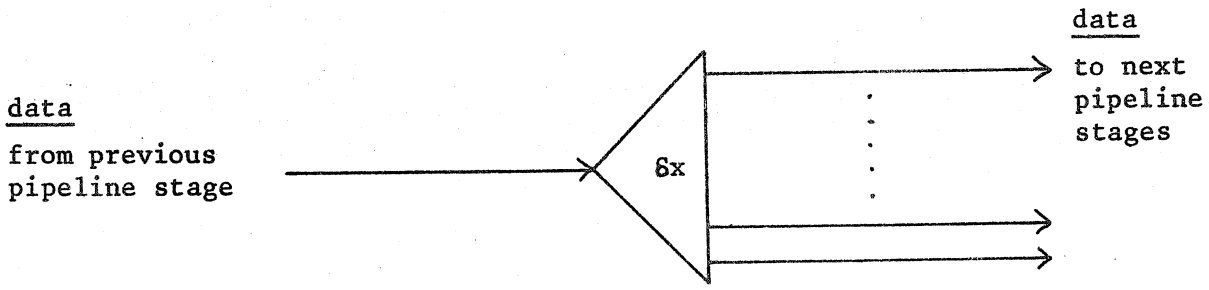
Each resource is associated with a cycle time. Resources are usually stores or processors. Since access may be made to a resource from different points in the pipeline, multiplexing resource controllers are provided. Each resource accessor is constrained to make one kind of demand of the resource. The resource controller associates the appropriate demand type with the accessor.

A simple example of a resource-accessing pipeline stage is the instruction store unit. Using the above notation this can be drawn as shown in figure 17. This shows that paired results from the matching store unit enter the instruction store accessor which arranges a read access to the instruction store. The store responds after time t_i. The accessor then transmits an executable instruction to the processing unit after a minimum delay (from initial receipt of the result) of δ_i ($\geq t_i$). The store is written from the external environment during initialisation. Hence reading and writing demands do not overlap and are not multiplexed at high speed.

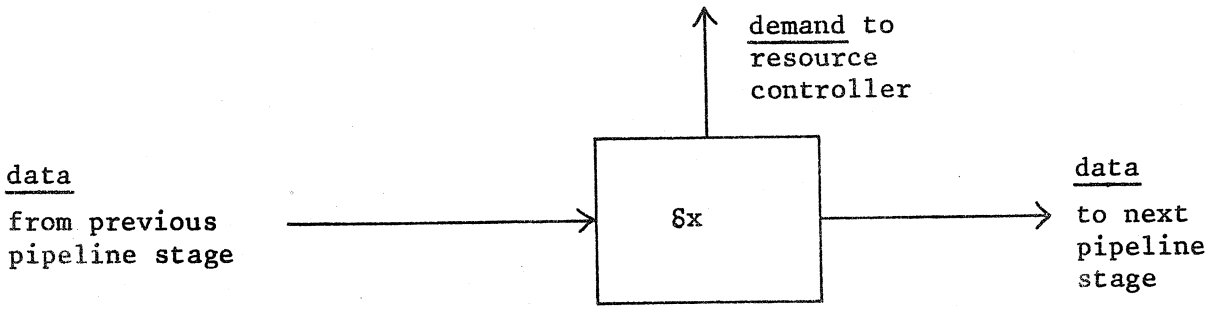
The situation is more complicated in the result queue unit which is drawn in figure 18. Two accessors use the linear result queue store as a circular buffer. The minimum delay through the store is $\delta_r + \delta_r'$: this must be longer than the time for two store accesses ($\geq 2t_r$).



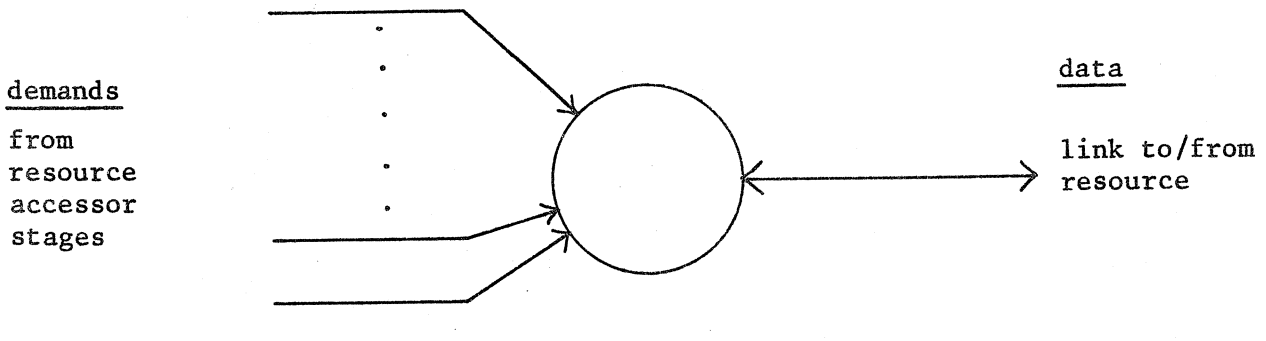
(a) arbitrator stage



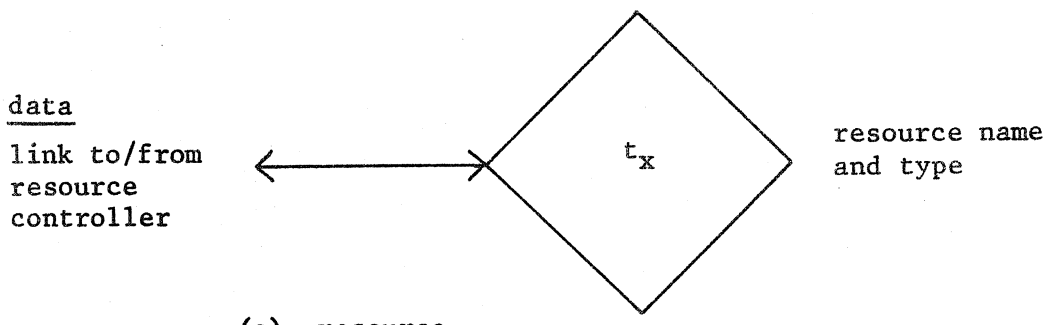
(b) distributor stage



(c) resource accessor stage



(d) resource controller



(e) resource

Figure 16 Schematic diagrams of basic hardware modules.

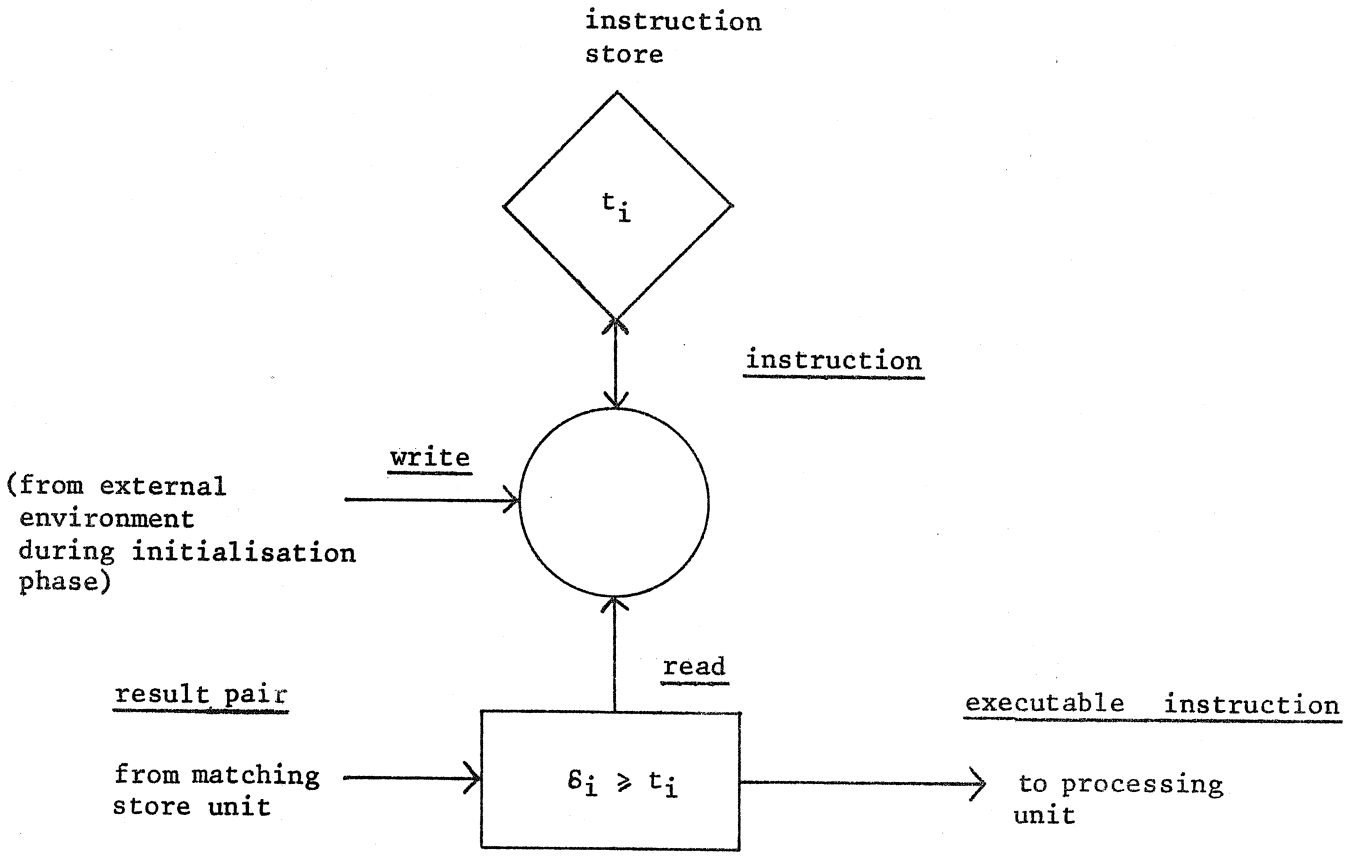


Figure 17 The instruction store unit.

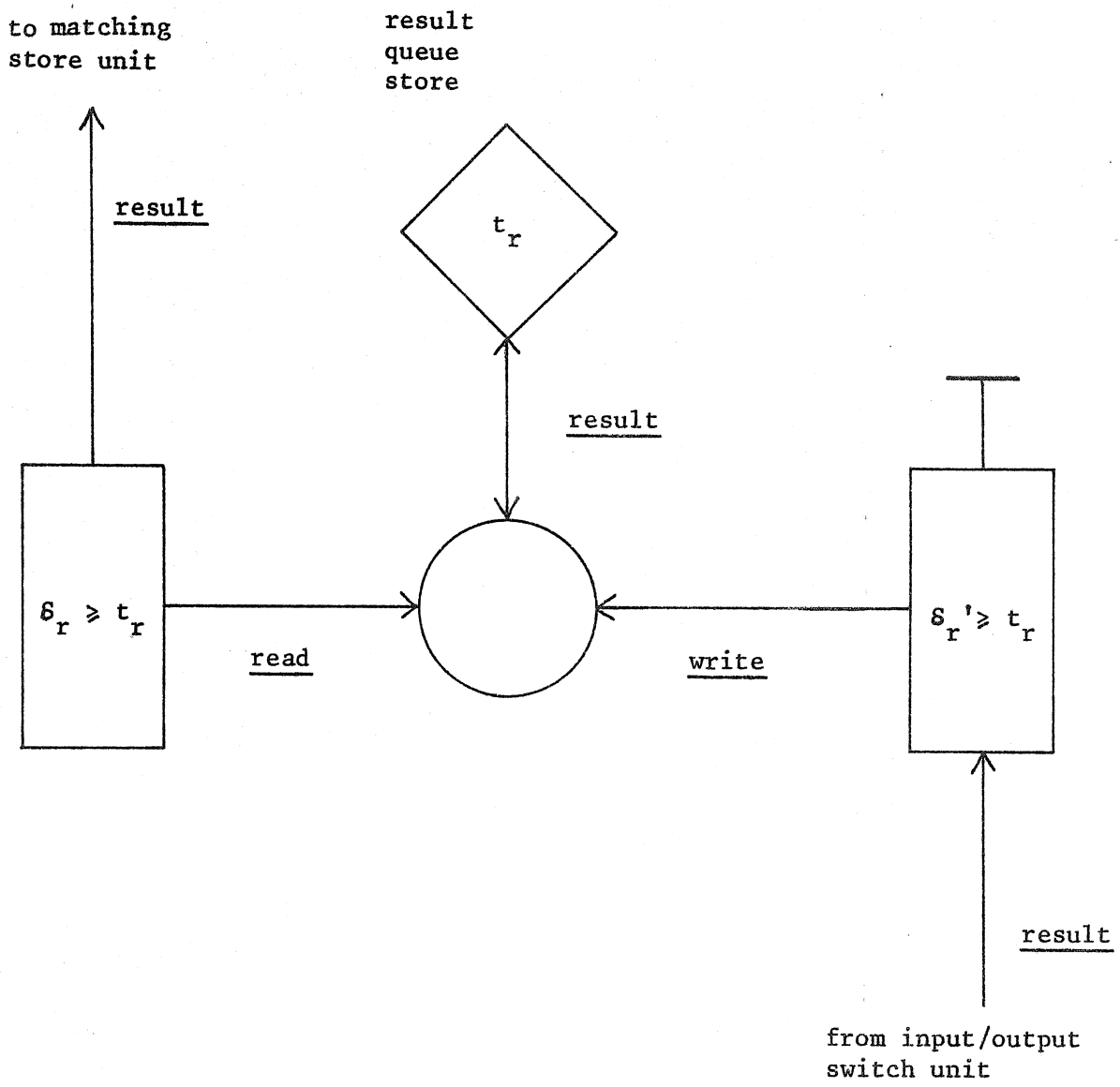


Figure 18 The result queue unit.

The input/output switch comprises an arbitrator and distributor as shown in figure 19. We have assumed for simplicity that there is no buffering within the switch although we would expect some to be present in a practical system. The minimum delay through the switch as shown is $\delta_{as} + \delta_{ds}$.

The processing unit is a parallel array of homogenous microprogrammable function units which are accessed via a distributor and an arbitrator as shown in figure 20. The processing resources are given a cycle time t_p which is the average time to execute an instruction[18]. We assume an even flow of information through the unit, and quote a minimum average delay through the unit of $\delta_{dp} + \delta_p + \delta_{ap}$, where $\delta_p \geq t_p$.

The matching store unit provides two separate paths for different types of instruction as shown in figure 21. Type C and D instructions (expected operands = 1) bypass the store and are immediately transformed into executable instructions (delay = $\delta_{am} + \delta_{dm}$). Type A and B instructions (expected operands = 2) are passed to the matching store accessors. The first accessor attempts to read-and-destroy the appropriate matching store entry. If this is successful, an executable instruction is sent to the processing unit (delay = $\delta_{am} + \delta_{mr} + \delta_{dm}$), otherwise the incoming result must be written associatively into the store. Although the last action does not have a delay time associated with producing an executable instruction, the multiplexed write access to the store will affect the minimum average delay time for reading the store (δ_{mr}^*). For well-formed programs the store will spend exactly half its time being read. Hence $\delta_{mr}^* \geq 2t_m$. The minimum average delay time for the whole unit depends on the relative frequencies of execution of types A, B, C and D instructions. (See Appendix C).

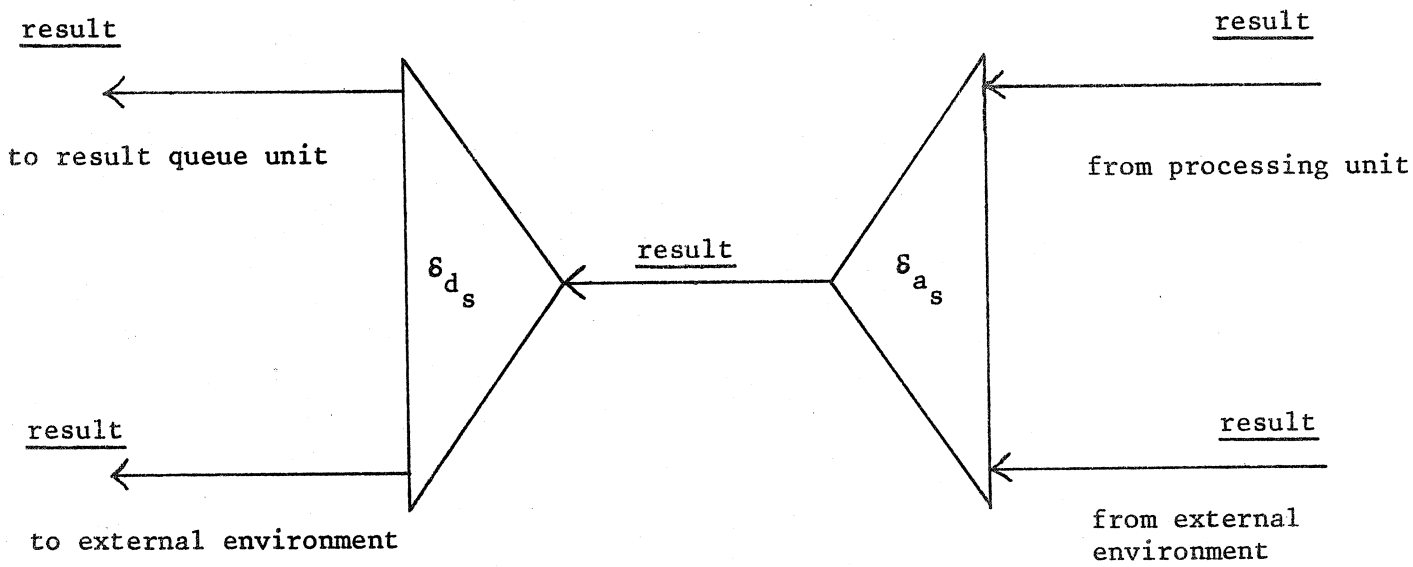


Figure 19 The input/output switch unit.

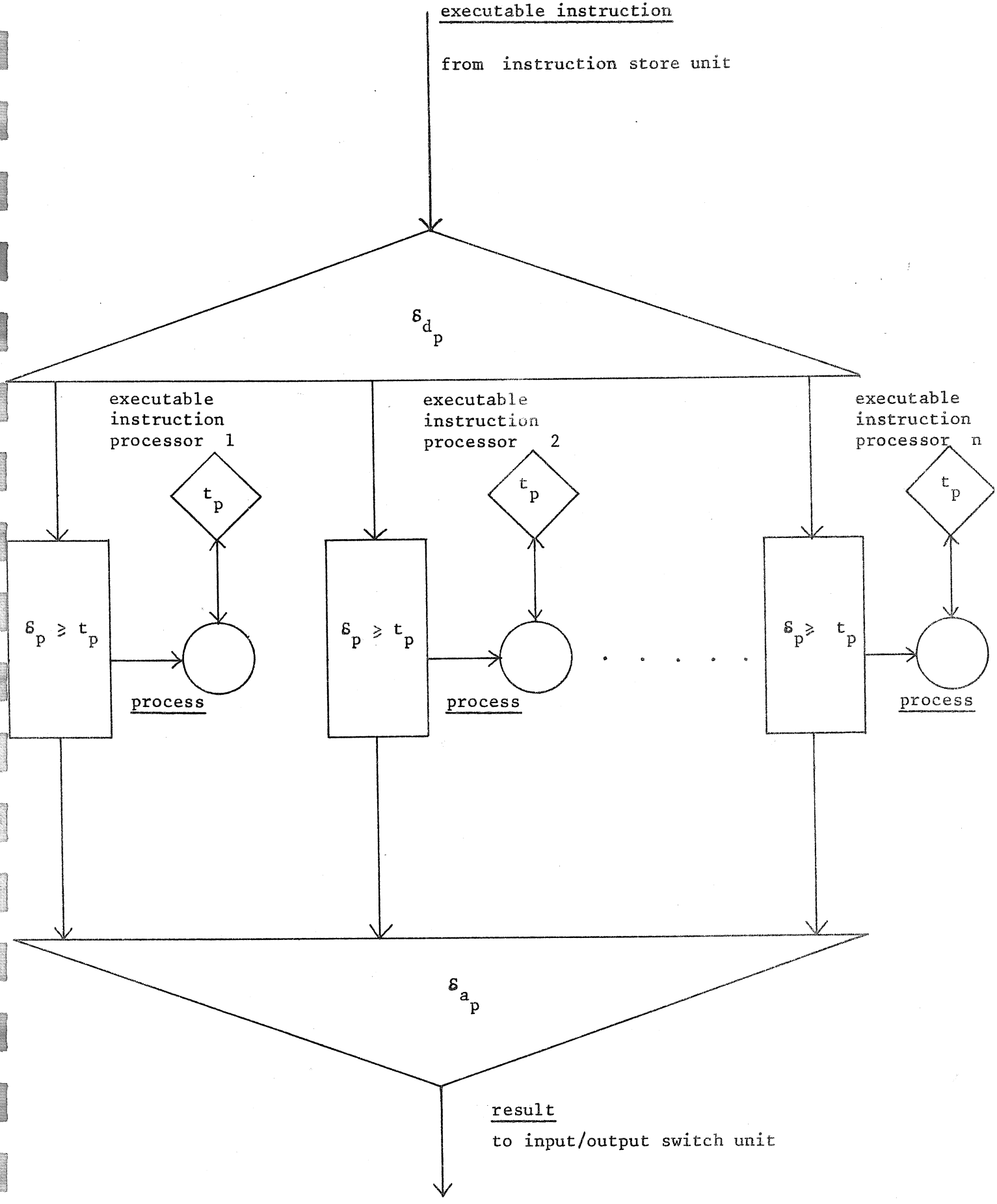


Figure 20 The processing unit.

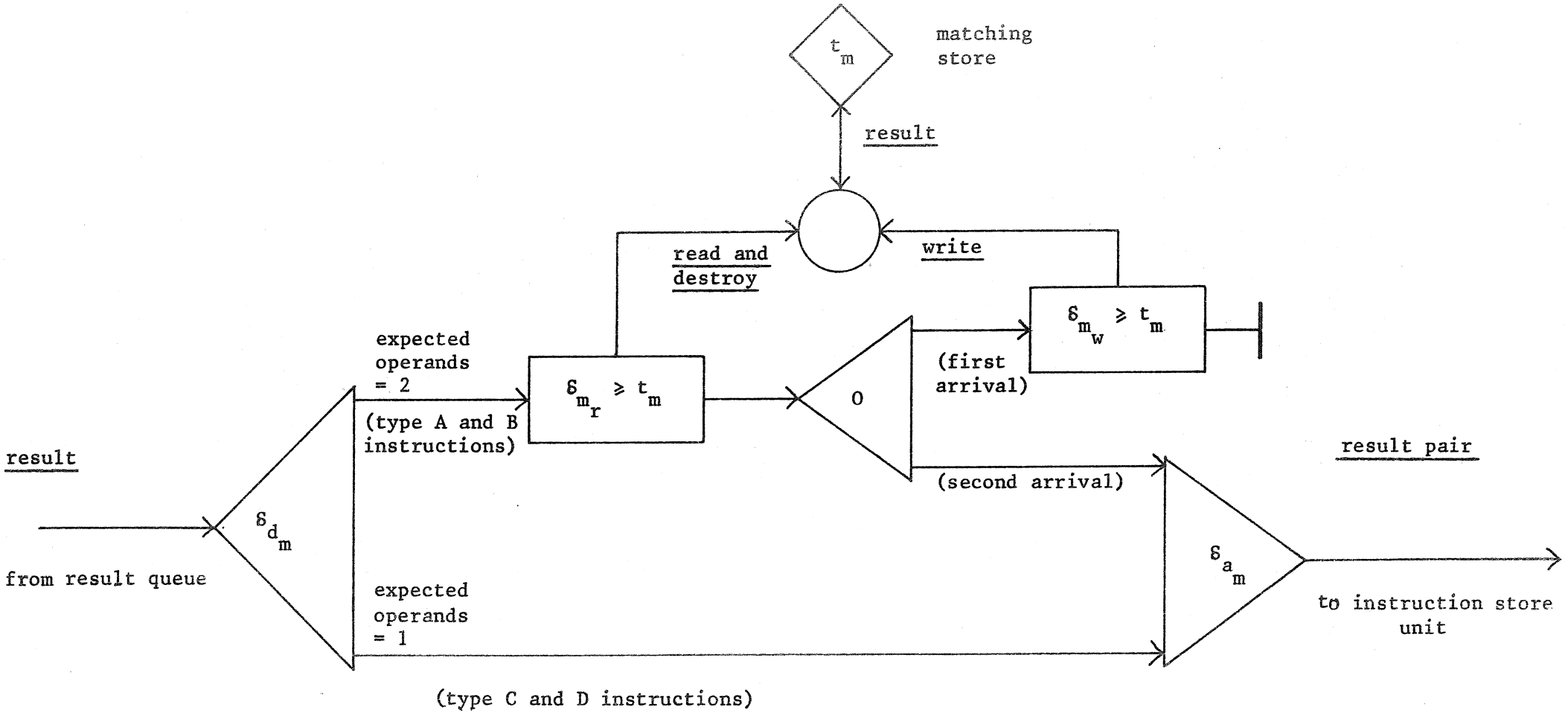


Figure 21 The matching store unit.

3.4. Performance

3.4.1. General

In this section we are interested in two aspects of the simple ring structured data flow computer. Firstly, we are concerned with how fast a particular program can be executed (and hence with the average instruction execution rate). Secondly, we wish to know the extent to which the architecture can tolerate the presence of hardware faults in various components.

Fault tolerance is relatively simple to analyse and is dealt with in section 3.4.3. Processing speed is difficult to assess, and the following analysis needs a brief introduction.

The two major factors influencing the instruction execution rate are (i) the technology used to construct the system; and (ii) the characteristics of the executing program. In Appendix A we derive some important characteristics of flowgraph programs. In particular, we demonstrate that the ratio $\psi = \frac{S_1}{S_\infty}$ as defined in section 2.1.4 is a useful measure of the inherent parallelism of a program. In Appendix C we show that a ring structured system constructed in any particular technology has an effective parallelism $k = \frac{\delta_s}{\delta_b}$ where δ_s is the delay round the ring and δ_b is the pipeline beat period. Formulae for k in terms of the characteristics of the pipeline introduced in section 3.3 are also presented. In the next subsection we show how ψ and k are related in order to assess the performance of the system for various programs. It is important to remember that the analysis is approximate since the ring structured system is not a "perfect" system as defined in section 2.1.4.

3.4.2. Efficiency of Execution

In Appendix C we suggest that a close approximation to a "perfect" computational step is the average time taken for one operation to pass right

around the ring. We call this time δ_s.

The easiest way to visualise this is to consider a completely serial graph in which only one node may be executed at a time. Each execution step must wait for the output result of the previous step to flow through the system before it can commence processing.

We define an average delay δ_b between instructions in the pipeline of the ring structured architecture and note that the "length" or "parallelism" of the pipeline, $k = \delta_s/\delta_b$. We now note that the total execution time of a computation using k processors is $T_k = S_k\delta_s = kS_k\delta_b$. The average instruction execution rate, Ω_k is given by :

$$\Omega_k = \frac{S_1}{T_k} = \mathcal{E}_k\left(\frac{1}{\delta_b}\right)$$

where $\mathcal{E}_k = \frac{S_1}{kS_k}$, and the S_p are defined as in Appendix A.

Since $(\frac{1}{\delta_b})$ is the maximum possible execution rate, $\mathcal{E}_k$ is a measure of the average occupancy of the pipeline. Its value lies between 0 and 1.

At this point we recall from Appendix A that for the execution of any program for which $\Psi = \frac{S_1}{S_\infty}$:

$$\varepsilon_{\Psi'} = \frac{S_1}{\Psi S_{\Psi'}} > \frac{1}{2}.$$

Since it must be impossible to speed up a computation by a factor greater than the extra proportion of processors provided, we must have, for all $i,j \geq 1$, $S_i \geq S_{i+j} \geq (\frac{i}{i+j})\, S_i$. Hence, any program for which $\Psi \geq k$ will execute in such a way that :

$$\mathcal{E}_k = \frac{S_1}{kS_k} \geq \frac{S_1}{\Psi S_\Psi} > \frac{1}{2}$$

Thus if Ψ (the program parallelism) $\geq k$ (the system parallelism), the average

occupancy of the pipeline is always more than 50%. Also, as Ψ gets larger so ϱ_k, and hence Ω_k, increases. For $\Psi<k$ the occupancy of the pipeline cannot be guaranteed to be any particular value, although larger ratios of Ψ/k will tend to give better values for ϱ_k.

It should be emphasised that we can increase Ψ by executing more than one program at a time. For m simultaneously active programs with individual values of ψ equal to Ψ_1, Ψ_2, ... Ψ_m, the effective value of Ψ is $\sum_{i=1}^{m} \Psi_i$.

Of course this technique can only be used to increase the average Ω_k : it cannot achieve a decrease in the execution time for any one program.

In Appendix C we give some typical values to the parameters of our performance analysis, and discuss the probably performance of an actual system implementation. This indicates that a system for which δ_b < 300nS and t_p = 3μs (i.e. moderate TTL technology) would have a system parallelism $k \simeq 14$. For programs with $\Psi \geq 14$, this system will execute over 1.6 million nodes per second[19].

3.4.3. Fault Tolerance

Unfortunately, the pipelined nature of the simple ring renders the tolerance of most component failures impracticable. It is only possible to tolerate faults in identical components which lie in parallel with one another. Thus the failure of any function unit in the processing unit of figure 20 can be handled by extracting that unit and then operating more slowly with a reduced complement. All other components in the ring are, however, critical in that no externally visible fault in them can be accommodated.

In the next section we describe a more tolerant architecture which can withstand the presence of faults in most of its hardware components.

4. A MULTILAYERED DATA FLOW COMPUTER

4.1. Multiple Ring Architecture

The performance analysis of the ring structured architecture indicates that while the system can be efficiently pipelined, the serial task of assembling executable instructions limits the execution rate (to $1/\delta_b$) and makes it impossible to tolerate certain hardware component failures. In this section we demonstrate that both of these problems can be overcome by interconnecting many data flow rings in a multilayered architecture as shown in figure 22.

The distinguishing feature of the multilayered architecture is an extended input/output switch which we call the exchange switch. Apart from the activity in this switch, all rings are autonomous in that they operate asynchronously and in parallel with one another. Increased throughput (compared to a single ring) is possible because none of the stores is shared by more than one ring and because the exchange switch is a serial, pipelined structure in which "clashes" are relatively unimportant (see below). The "cost"of increasing throughput is that the programs using the system need higher values of Ψ in order to use the system efficiently.

Note that more than one input or output channels may be accommodated in this system. Thus it is possible to tailor the resources in a system to the requirements of its programs. Compute-bound applications could use many rings with one or two input/output channels. Input/output bound applications could choose the reverse emphasis.

The components in an implementation of this architecture are identical to those introduced in section 3.3, except for the exchange switch. The latter module is organised as an arbitration network [JM77] as shown in figure 23.

The switch consists of alternate layers of buffer, arbitration, and

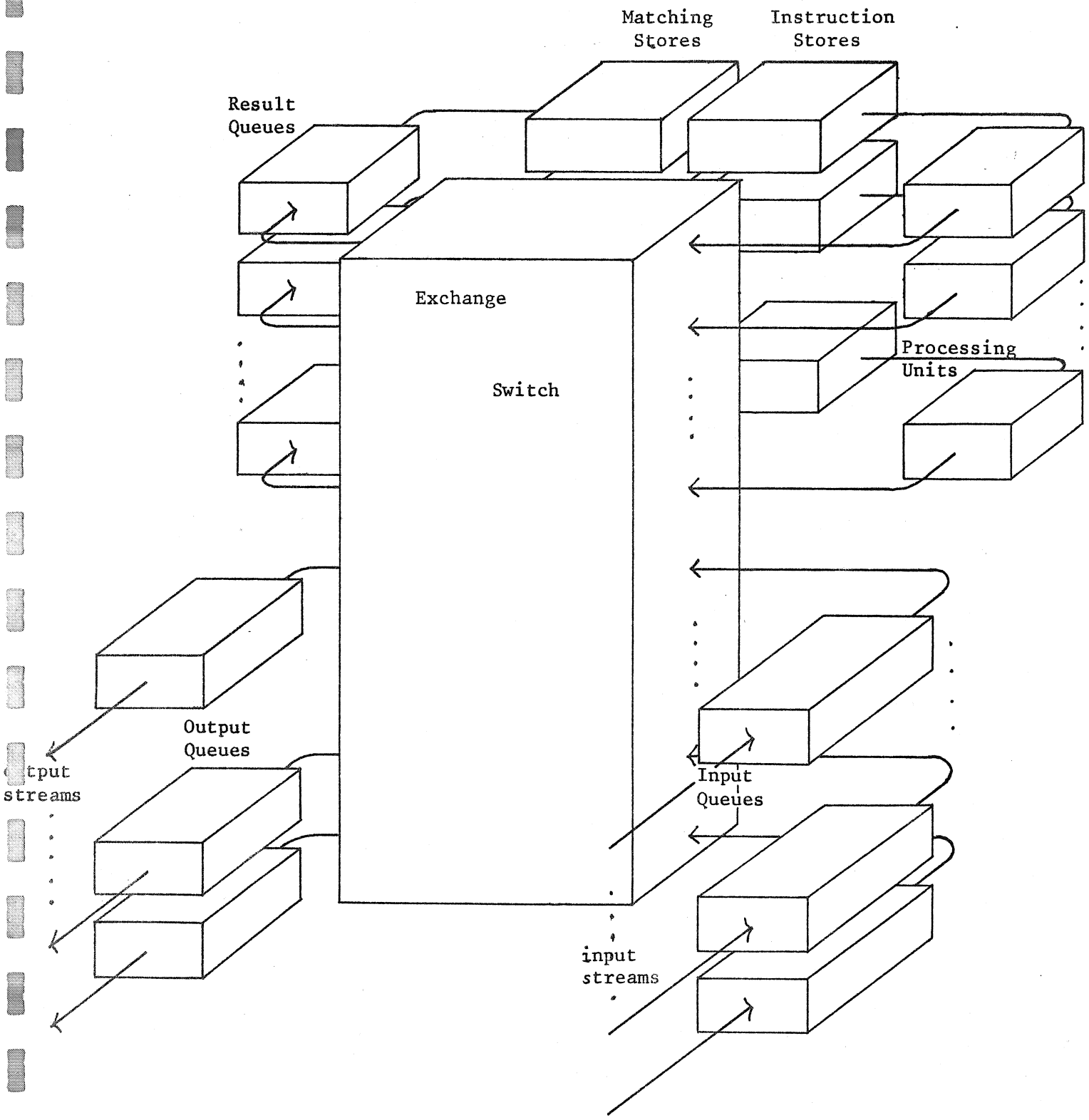


Figure 22 Multilayered data flow architecture.

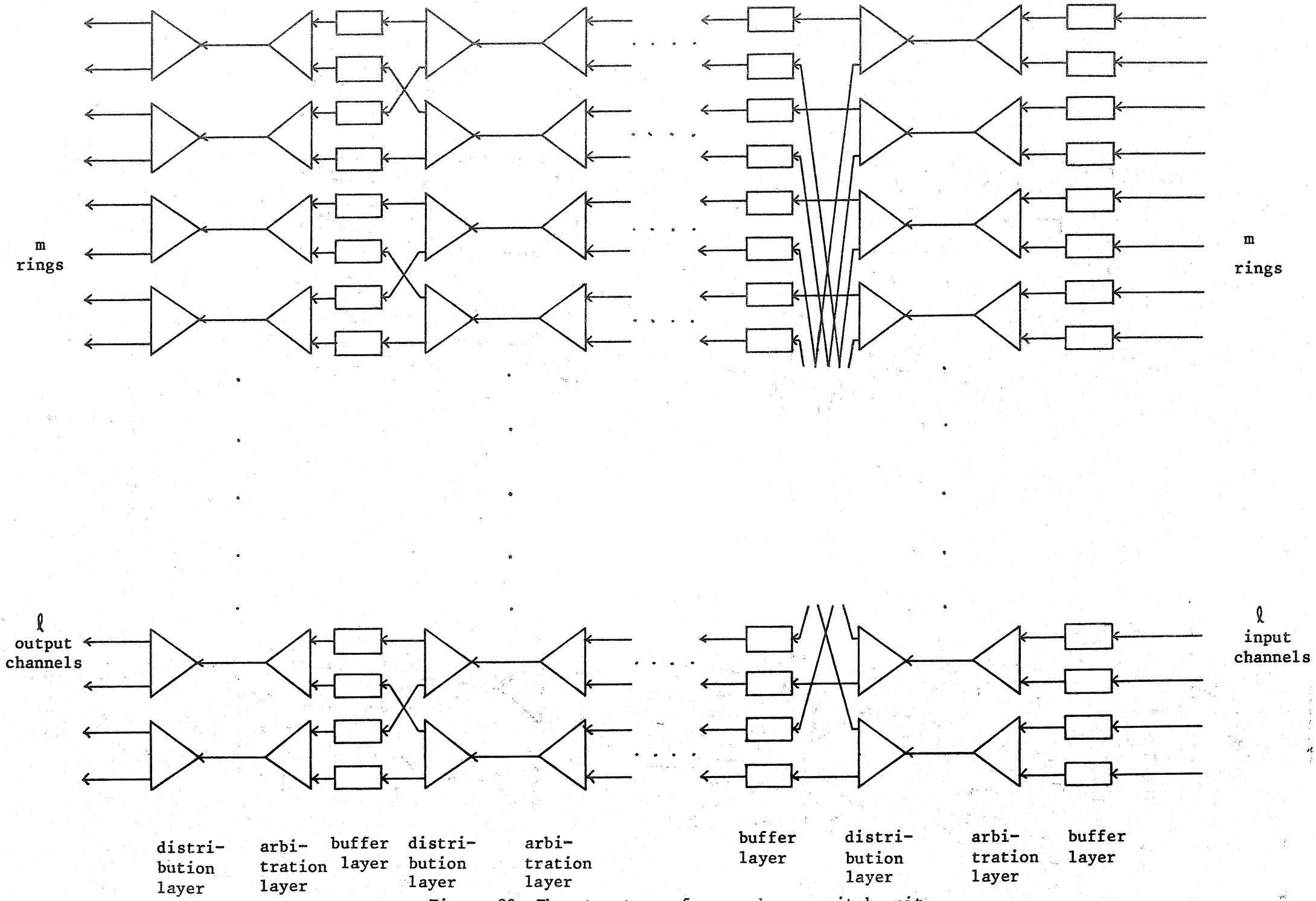


Figure 23 The structure of an exchange switch unit.

distribution units. At each distribution stage, results are directed to an output according to one bit of their name. A different bit of the name is used at each stage (the same bit being used for all distribution units in that stage). It is possible to alter the bit that is used at each stage and also to direct all results to just one output or to both outputs. Thus it is possible to isolate areas of the system in order to provide correct operation when components are faulty.

The buffer stages are optional and can be used to average out the effects of address "clashes" at each distribution stage of the switch. They will merely increase the delay through the switch and hence the effective parallelism, k, of the system (see Appendix D).

The component-complexity of the exchange switch is $O(m+\ell) \log_2 (m+\ell))$, where m is the number of data flow rings, and ℓ is the number of input/output channels. This makes the switch structure attractive for systems with very large computational power.

4.2. Performance

4.2.1. Parallelism in the System

We can extend the notation of Appendix C and section 3.4.2 to analyse the multiple ring architecture. The main extension concerns the value of k, the effective parallelism of the architecture. It is no longer sufficient to equate this to the "length" of the pipeline since there are now m such pipelines which must all be kept full if the maximum execution rate is to be achieved.

If we note that the average minimum delay around the system, δ_s, is the delay around any ring plus the delay through the exchange switch, then the obvious definition for k is :

$$k = m * \frac{\delta_s}{\delta_b}$$

where δ_b is again the average delay between instructions in a single pipeline.

We can now complete a similar analysis to that of section 3.4.2 by noting that the execution rate is given by :

$$\Omega_k = \varrho_k \left(\frac{m}{\delta_b} \right)$$

where

$$\varrho_k = \frac{S_1}{kS_k}$$

In this case the maximum possible execution rate is $(\frac{m}{\delta_b})$, whilst ϱ_k measures the average occupancy of all m pipelines.

We now need to develop an expression for $\varepsilon_{\Psi'}$ when a program is executed on a multiple ring system. This is not the same as that developed in Appendix A (for the single ring) because we must now distribute the Ψ' processors over m rings and we cannot guarantee that all of these can be used in any one step of C_∞ using Ψ' processors. The analysis of Appendix B establishes that :

$$\varepsilon_{\Psi_{m'}} = \frac{S_{11}}{\Psi_m S_{\Psi_{m'}}} > \frac{1}{m+1}$$

This is a generalisation of the previous result $(\varepsilon_{\Psi_{2'}} > \frac{1}{2})$ and illustrates the problem of keeping many rings occupied simultaneously. The switching strategy employed in the distribution stages of the exchange switch can have a dramatic effect on the actual value of $\varepsilon_{\Psi m'}$ since the best case performance must be the same as for a single ring $(\varepsilon_{\Psi_{1'}})$.

Note again the similarity of the forms for ϱ_k and $\varepsilon_{\Psi m'}$. This leads us to conclude that for $\Psi_m \geq k$, then $\varrho_k \geq \varepsilon_{\Psi m'} > (\frac{1}{m+1})$.

In Appendix D we look at a 10 ring system of construction similar

to the single ring system described in Appendix C. We see that $k \simeq 160$. Although the theoretical maximum exeuction rate is some 33 million nodes per second, we can only guarantee an average rate of 3.1 million nodes per second for programs with $\Psi \geq 160$.

Jinks [Ji77] describes a simulator for the multilayered architecture in his Master's dissertation. We are currently using this simulator to evaluate various multilayer systems. In particular, we intend to study the effect of different switching strategies in the exchange switch. Using the primitive strategy of distributing instructions equally among the rings so that the instruction at address A is located in ring i = (i+A mod m) for i = 1 to m, we have established for several small programs that $\frac{\varepsilon_{\Psi_m'}}{\varepsilon_{\Psi_1'}} > \frac{3}{4}$ if $m \leq 10$.

4.2.2. Fault Tolerance

That the exchange switch can isolate faulty data flow rings should be clear from the discussion of the switch structure given earlier. What is not quite so obvious is that individual {buffer, arbitrator, distributor} segments of thw switch itself may be avoided if the set of rings to which the faulty segment is indispensible are sacrificed. The only type of fault that cannot be handled is one which causes different distribution stages of the exchange switch to see a different switching "key". This is necessarily a weak point of the system which might perhaps be strengthened by hardware fault tolerance techniques in an extremely fault tolerant configuration.

The above discussion merely demonstrates that fault tolerant running is feasible. It says nothing about detecting and recovering from faults. That remains an area for further research.

5. CONCLUSIONS

5.1. Summary

We have described a data flow graphical notation and a set of high level language constructs which have been developed together with two computer architectures.

The data flow notation is similar to those of Dennis [De74] and Arvind and Gostelow [AG77].

The language constructs have much in common with those in Lucid [AW77], although recursion is incorporated, making LAPSE similar to Id [AG78]. High level programs can be efficiently translated into graphical machine code.

One of the architectures is a simple pipelined ring structure. The other comprises many simple rings connected in a multilayered, parallel configuration. The multilayered architecture offers the following advantages :

(i) the modular structure is cheap and easy to maintain;

(ii) for sufficiently parallel programs, throughput can be increased substantially by adding more modules to the system;

(iii) many hardware faults can be tolerated, albeit with degraded performance.

The performance of the architectures has been analysed, and limits to effectively expoitable parallelism in programs have been established. These are naturally influenced by the technology in which the system is constructed. Examples indicate that high instruction execution rates can be obtained using moderate technology.

5.2. Current and Future Research

We have written emulator and simulator programs for the proposed architectures [Ji77]. We also have a compiler for the LAPSE language which compiles directly into data flow code [G178]. These programs are currently being used to refine the structure and ordercode of the architecture. We are aiming to generate efficient code from compilers in the system. We are

also studying the performance of various proposed hardware system configurations.

For the longer term future we envisage applying concerted effort in the following areas :

(i) investigation of high level and low level parallel algorithms;

(ii) investigation of alternative high level languages (for example, a LISP-based functional language similar to that proposed by Weng [We75]);

(iii) study of "operating system" techniques for supporting multiprogramming and yielding a reliable (fault tolerant) system; and

(iv) design and construction of a prototype extensible (multilayered system).

ACKNOWLEDGEMENTS

The authors wish to thank all their colleagues who have worked on the data flow project at Manchester, all those who have commented on the many versions of this paper, and all those who have listened patiently to our enthusiastic brainstorms. In particular, we acknowledge the efforts of Phil Treleaven, Viv Woods and Rob Witty.

We also acknowledge the support of the Science Research Council of Great Britain which will enable us to implement a prototype single ring system in the near future.

REFERENCES

[Ac78] Ackerman W.B. : "Preliminary Data Flow Language" - CSG note 36, Laboratory for Computer Science, M.I.T., March 1978.

[Ad70] Adams D.A. : "A Model for Parallel Computations" - in Hobbs (ed) "Parallel Processor Systems, Technologies and Applications" - Spartan Books, 1970, pp.311-333.

[AG77] Arvind & Gostelow K.P. : "A Computer Capable of Exchanging Processors for Time" - Information Processing 77, North Holland, 1977, pp.849-853.

[AG78] Arvind, Gostelow K.P. & Plouffe W. : "The (Preliminary) Id Report : An Asynchronous Programming Language and Computing Machine" - Technical Report 114, Department of Information and Computer Science, UC Irvine, May 1978.

[AW77] Ashcroft E.A. & Wadge W.W. : "Lucid, a Nonprocedural Language with Iteration" - CACM, Vol.20, No.7, July 1977, pp.519-526.

[Ba73] Baer J.L. : "A Survey of Some Theoretical Aspects of Multiprocessing" - ACM Computing Surveys, Vol.5, No.1, March 1973, pp.31-80.

[Ba74] Bährs A. : "Operation Patterns" - Lecture Notes in Computer Science, Vol.5, Springer-Verlag, 1974, pp.217-246.

[Br62] Brown G.W. : "A New Concept in Programming" - in "Computers and the World of the Future" (ed. M. Greenberger), MIT Press, Cambridge, Mass. 1962.

[Ch71] Chamberlin D.D. : "The 'Single-Assignment' Approach to Parallel Processing" - AFIPS FJCC, Vol.39, 1971, pp.263-269.

[Co76] Comte D, et al : "TEAU 9/7 : SYSTEME LAU - Summary in English" - Report # 11/3059, ONERA CERT-DERI, Toulouse, October 1976.

[Da77] Davis A.L. : "Architecture of DDM1 : A Recursively Structured Data Driven Machine" - Technical Report, Department of Computer Science, University of Utah, 1977.

[DF74] Dennis J.B., Fosseen J.B. & Linderman J.P. : "Data Flow Schemas" - Lecture Notes in Computer Science, Vol.5, Springer-Verlag, 1974, pp.187-216.

[De74] Dennis J.B. : "First Version of a Data Flow Procedure Language" - Lecture Notes in Computer Science, Vol.19, Springer-Verlag, 1974, pp.362-376.

[DM75] Dennis J.B. & Misunas D.P. : "A Preliminary Architecture for a Basic Data Flow Processor" - Proceedings of the Second Annual IEEE Symposium on Computer Architecture, January 1975, pp.126-132.

[DM77] Dennis J.B., Misunas D.P. & Leung C.K. : "A Highly Parallel Processor Using a Data Flow Machine Language" - CSG Memo # 134, Laboratory for Computer Science, Massachussetts Institute of Technology, January 1977.

[En77] Enslow P.H. : " Multiprocessor Organisation - A Survey" - ACM Computing Surveys, Vol.9, No.1, March 1977, pp.103-129.

[Fl72] Flynn M.J.: "Some Computer Organisations and their Effectiveness" - IEEE Transactions on Computers, Vol.C-21, No.9, September 1972, pp.948-960.

[Fu76] Fuller S.H. : "Price/Performance Comparison of C.mmp and the PDP-10" - Proceedings of the Third Annual IEEE Symposium on Computer Architecture, January 1976, pp.195-202.

[FJ77] Fuller S.H., Jones A.K. & Durham I. (eds.) : "Cm* Review" - Report, Department of Computer Science, Carnegie-Mellon University, June 1977.

[Gl78] Glauert J. : "A Single Assignment Language for Data Flow Computing" - M.Sc. Dissertation, Department of Computer Science, University of Manchester, January 1978.

[Gl74] Glushkov V.M. et al : "Recursive Machines and Computing Technology" - Information Processing 74, North Holland, 1974, pp.65-70.

[HK77] Hibbard P.G. Knueven P. & Leverett B.W. : "Issues in the Efficient Implementation and Use of Multiprocessing in Algol 68" - Report, Department of Computer Science, Carnegie-Mellon University, September 1977.

[JM77] Jacobsen R.G. & Mişunas D.P. : "Analysis of Structures for Packet Communication" - Proceedings of the IEEE International Conference on Parallel Processing, August 1977, pp.38-43.

[JW74] Jensen K. & Wirth N. : "PASCAL User Manual and Report" - Lecture Notes in Computer Science, Vol.18, 1974.

[Ji77] Jinks P.J. : "A Data Flow System Simulator" - M.Sc. Dissertation, Department of Computer Science, University of Manchester, October 1977.

[KM66] Karp R.M. & Miller R.E. : "Properties of a Model for Parallel Computations : Determinacy, Termination and Queueing" - SIAM J. Applied Mathematics, Vol.11, No.6, November 1966, pp.1390-1411.

[Ko73] Kosinski P.R. : "A Data Flow Language for Operating Systems Programming" - ACM SIGPLAN Notices, Vol.8, No.9, September 1973, pp.89-94.

[Mc65] McCarthy J. et al : "LISP 1.5 Programmer's Manual" - 2nd ed., MIT Press, 1965.

[Mi73] Miller R.E, : "A Comparison of Some Theoretical Models of Parallel Computation" - IEEE Transactions on Computers, Vol.C-22, No.8, August 1973, pp.710-717.

[MC74] Miller R.E. & Cocke J. : "Configurable Computers : A New Class of General Purpose Machines" - Lecture Notes in Computer Science, Vol.5, Springer-Verlag, 1974, pp.285-298.

[Mi77] Miranker G.S. : "Implementation of Procedures on a Class of Data Flow Processors" - Proceedings of the IEEE International Conference on Parallel Processing, August 1977, pp.77-86.

[Mi75] Misunas D.P. : "Structure Processing in a Data Flow Computer" - Proceedings 1975 Sagamore Conference on Parallel Computation, (MIT CSG Memo # 129), August 1975.

[Or75] Ornstein S.M. et al : "Pluribus - A Reliable Multiprocessor" - AFIPS NCC, Vol.44, 1975, pp.551-559.

[Pl76] Plas A. et al : "LAU System Architecture : A Parallel Data Driven Processor Based on Single Assignment" - Proceedings of the IEEE International Conference on Parallel Processing, August 1976, pp.293-302.

[RL77] Ramamoorthy C.V. & Li H.F. : "Pipeline Architecture" - ACM Computing Surveys, Vol.9, No.1, March 1977, pp.61-102.

[Ru77] Rumaugh J.E. : "A Data Flow Multiprocessor" - IEEE Transactions on Computers,Vol.C-26, No.2, February 1977, pp.138-146.

[SM77] Schroeder M.A. & Meyer R.A. : "A Distributed Computer Using a Data Flow Approach" - Proceedings of the IEEE International Conference on Parallel Processing, August 1977, p.93.

[Sl70] Slutz D.R. : "Flow Graph Schemata" ; Proceedings of the Project MAC Conference on Concurrent Systems and Parallel Computation, 1970, pp.129-142.

[St73] Stone H.S. : "Problems of Parallel Computation" - in Traub J.F. (ed.) "Complexity of Sequential and Parallel Numerical Algorithms" - Academic Press 1973, pp.1-16.

[SF77] Swan R.J., Fuller S.H. & Siewiorek D.P. : "Cm* - A Modular Multi-microprocessor" - AFIPS NCC, Vol.46, 1977, pp.637-663.

[Sy76] Syre J.C. et al : "Parallelism, Control and Synchronisation Expression in a Single Assignment Language" - Proceedings of the Fourth Annual ACM Computer Science Conference, February 1976, pp.

[Te68] Tesler L.G. & Enea H.J. : "A Language Design for Concurrent Processes" - AFIPS SJCC, Vol.32, 1968, pp.403-408.

[Th70] Thornton J.E. : "Design of a Computer - The CDC6600" - Scott Foresman, 1970.

[TW75] Thurber K.J. & Wald L.D. : "Associative and Parallel Processors" - ACM Computing Surveys, Vol.7, No.4, December 1975, pp.215-255.

[We75] Weng K.S. : "Stream-Oriented Computation in Recursive Data Flow Schemas" - Technical Memo # 68, Laboratory for Computer Science, Massachussetts Institute of Technology, October 1975.

[WB72] Wulf W.A. & Bell C.G. : "C.mmp - A Multiminiprocessor" - AFIPS FJCC, Vol.41, 1972, pp.765-777.

[WH78] Wulf W.A. & Harbison S.P. : "Reflections in a Pool of Processors : An Experience Report on C.mmp/Hydra" - Report, Department of Computer Science, Carnegie-Mellon University, February 1978.

[YF77] Yau S.S. & Fung H.S. :"Associative Processor Architecture - A Survey" - ACM Computing Surveys, Vol.9, No.1, March 1977, pp.3-28.

[YK58] Young J.W. & Kent H.K. : "Abstract Formulation of Data Processing Problems" - NCR Internal Report, 1958.

[Zu77] Zurawski J. : "The Design of a Processor Array for a Data Flow Architecture" - M.Sc. Dissertation, Department of Computer Science, University of Manchester, October 1977.

APPENDIX A

Parallelism in Flowgraphs Executed by a "Perfect" System

For a given flowgraph computation, C, a perfect execution, C_p, by a perfect system of p processors, is defined as follows, :

(i) C_p progresses via a series of S_p distinct steps which are generated by means of the p processors executing as many elegible instructions as possible (up to p) at the start of C_p, and thereafter at the end of each step of C_p until no further instructions are eligible;

(ii) each processor takes a unit step time to perform any instruction;

(iii) all processors start and finish executing their instructions simultaneously so that C_p can be monitored via snapshots (see section 2.1) of the flowgraph between each step.

It is apparent that C_1, representing a completely serial execution of the flowgraph, is the longest possible perfect execution. Hence :

$$S_1 = \text{total number of executed instructions}$$
$$= \max (S_i) \qquad i \geq 1$$

The shortest possible perfect execution is usually called C_∞ although the same number of steps is usually attainable with a finite number of processors :

$$S_\infty = \min (S_i) \qquad i \geq 1$$

C_∞ is obtained by executing all eligible instructions of the initial snapshot in each step. Note that only C_1 and C_∞ will always have the same numbers of steps S_1 and S_∞. In C_∞ we have no choice about which instructions to execute in each step, and in C_1 we only choose one eligible instruction at each step which means that there is always an instruction available unless we have executed them all. In an intermediate perfect execution C_p (where $S_1 > S_p > S_\infty$), the random choice of up to p eligible instructions before each step can affect S_p.

For example, considering the flowgraph of figure 24, we can see that $S_1=6$ and $S_3=S_4= \ldots = S_\infty=2$. However, S_2 may be 3 or 4 according as we choose to execute the pairs of eligible instructions in the order {(1:,2:); (3:,4:); (5:,6:)} or {(2:,3:); (5:,6:); (1:,idle); 4:,idle)}.

Figure 25 is a diagrammatic representation of the perfect execution C_∞ of a computation. At the start of the ith step, W_{i-1} input tokens are able to activate all n_i eligible instructions. At the end of the ith step, W_i output tokens are produced and transmitted to the (i+1)th step. Included in the W_i tokens are those tokens from W_{i-1} which do not cause an instruction to become eligible in the ith step: these are passed directly through to the next step. Every step except the last must produce at least one eligible instruction. Hence :

$$n_i \geqslant 1 \qquad 1 \geqslant i \geqslant S$$

We define the <u>parallelism</u> of a flowgraph C as :

$$\Psi = \frac{S_1}{S_\infty} .$$

Since S_1 equals the total number of instructions to be executed, we see that :

$$\Psi = \frac{\sum_{i=1}^{S_\infty} n_i}{S_\infty} .$$

We require an integer measure of Ψ which we define as :

$$\Psi' = \lceil \Psi \rceil .$$

Finally, we define the (speedup) efficiency of a perfect system of p processors as follows :

$$\varepsilon_p = \frac{S_\infty}{S_p} .$$

In general, $S_1 \geqslant S_p \geqslant S_\infty$, and S_p may not be constant. It is thus customary to consider ε_p to be the worst-case efficiency (i.e. that corresponding to the

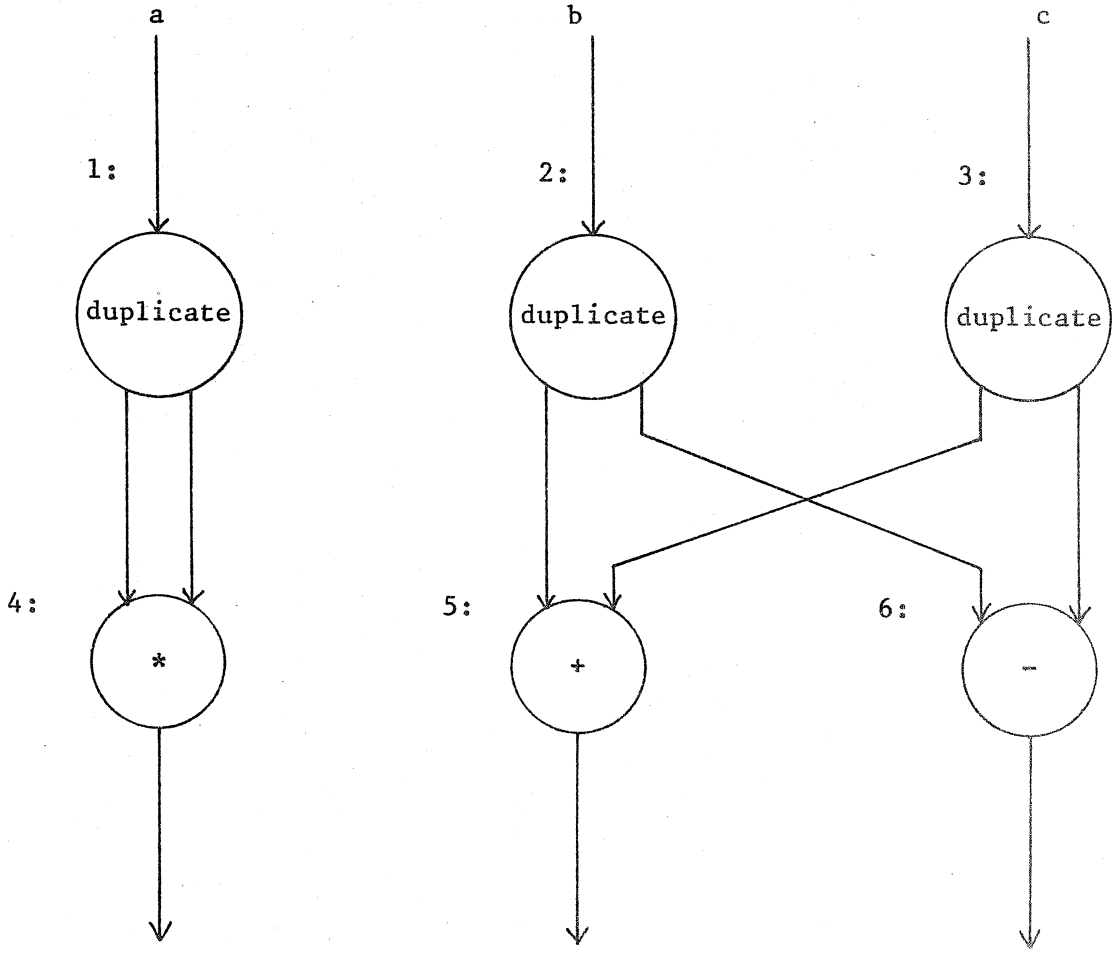


Figure 24 Flowgraph illustrating non-determinacy in S_2 .

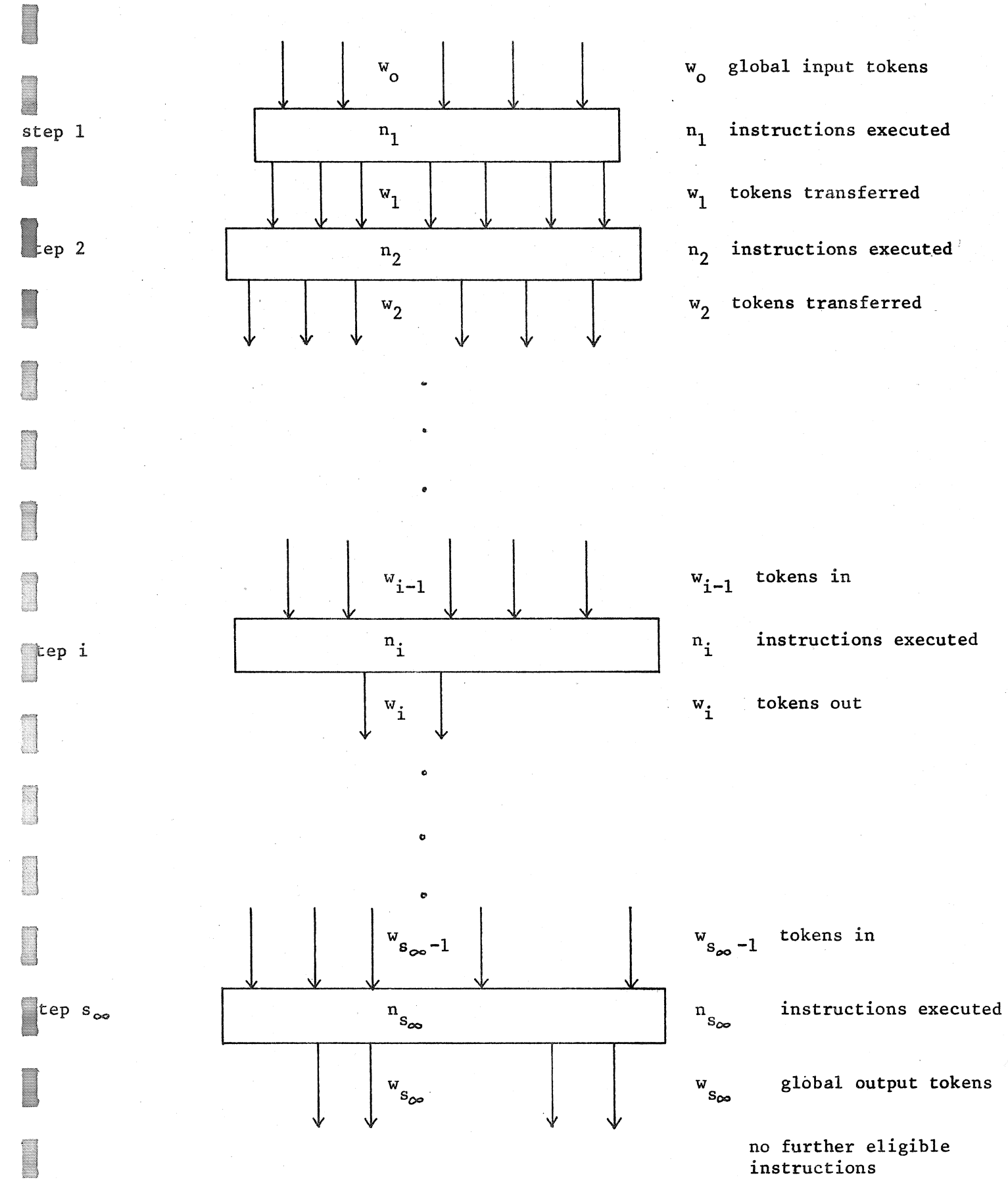


Figure 25 Schematic diagram of a perfect execution, C_∞ .

maximum value of S_p).

Considering a system of $\Psi' = \left\lceil \frac{S_1}{S_\infty} \right\rceil$ perfect processors, we see that :

$$\varepsilon_{\Psi'} = \frac{S_1}{\Psi S_{\Psi'}} .$$

We can now state the major result of this appendix in the following theorem :

Theorem A :

For any flowgraph program C run on a perfect system of Ψ' processors.

$$\frac{1}{2} < \varepsilon_{\Psi'} \leqslant 1.$$

Proof :

For finite Ψ', by definition, $S_{\Psi'} \geqslant S_\infty$. Thus :

$$\varepsilon_{\Psi'} \leqslant 1.$$

The number of steps in $C_{\Psi'}$ can be determined by considering the steps of C_∞. For ith step of C_∞, the maximum number of steps in $C_{\Psi'}$ is $\left\lceil \frac{n_i}{\Psi'} \right\rceil$. Hence

$$S_{\Psi'} \leqslant \sum_{i=1}^{S_\infty} \left\lceil \frac{n_i}{\Psi'} \right\rceil$$

$$\leqslant \sum_{i=1}^{S_\infty} \left\lceil \frac{S_\infty n_i}{S_1} \right\rceil$$

$$< \sum_{i=1}^{S_\infty} \left(\frac{S_\infty n_i}{S_1} \right) + \sum_{i=1}^{S_\infty} (1)$$

$$< \frac{S_\infty}{S_1} \left(\sum_{i=1}^{S_\infty} (n_i) \right) + S_\infty$$

$$< 2S_\infty .$$

Hence :

$$\varepsilon_{\Psi'} = \frac{S_\infty}{S_{\Psi'}} > \frac{1}{2} .$$

APPENDIX B

Parallelism in Flowgraphs Executed by an "Ideal" Multilayered System

The analysis of Appendix A can be extended to describe the behaviour of a multilayered architecture executing a flowgraph computation.

The ideal multilayered system comprises m layers each containing q perfect processors so that the total number of processors p=mq. For a given flowgraph, C, the ideal execution, C_{mq}, by a system of p processors is defined as follows :

(i) C_{mq} progresses via a series of S_{mq} distinct steps which are generated by means of the q processors on each ring executing as many eligible instructions as possible (up to q per ring) at the start of C_{mq}, and thereafter at the end of each step of C_{mq} until no further instructions become eligible;

(ii) each processor takes a unit step time to perform any instruction;

(iii) all processors start and finish executing their instructions simultaneously.

The longest possible (i.e. serial) ideal execution is C_{11}, so that :

$$S_{11} = \text{total number of executed instructions}$$
$$= \max (S_{ij}) \qquad i,j \geq 1$$

The shortest possible ideal execution is $C_{\infty\infty}$. However, we are more often interested in using a finite number of layers. Hence we define the shortest practical ideal execution for m layers as $C_{m\infty}$.

$$S_{\infty\infty} = \min (S_{ij}) \qquad i,j \geq 1$$
$$S_{m\infty} = \min (S_{mj}) \qquad j \geq 1$$

Also :

$$S_{m1} = \max (S_{mj}) \qquad j \geq 1$$

The distribution of instructions across the layers is critical to the behaviour of the mutlilayered system. We may adopt any strategy for

placing instructions in a particular layer and so the individual layer executions, C_q^{ℓ} $(1 \leqslant \ell \leqslant m)$, can be different from one another. The inter-layer switches achieve the distribution of instructions to layers.

Note that $S_{11}(\equiv S_1)$ and $S_{\infty\infty}(\equiv S_\infty)$ are constant, corresponding to the execution of one or all eligible instruction(s) per step. For any fixed strategy for distributing instructions to layers, S_{m1} and $S_{m\infty}$ are also determinate. In terms of figure 26 :

$$S_{m1} = \sum_{i=1}^{S_{m\infty}} \left(\max \left(n_i^{\ell}\right)\right) \qquad 1 < \ell < m$$

$$S_{11} = \sum_{i=1}^{S_{m\infty}} \sum_{\ell=1}^{m} \left(n_i^{\ell}\right)$$

$$\geqslant S_{m1}$$

We define the multilayer parallelism of a flowgraph C as :

$$\Psi_m = \frac{S_{11}}{S_{m\infty}} .$$

We also require an integer, derived from Ψ_m, which we define as :

$$\Psi'_m = \left\lceil \frac{\Psi_m}{m} \right\rceil .$$

Finally, we define the multilayer efficiency of an ideal system as :

$$\varepsilon_{mq} = \frac{S_{m\infty}}{S_{mq}} .$$

Again it is customary to consider the worst-case of S_{mq}. In the case that $q = \Psi'_m$, we see that :

$$\varepsilon_{\Psi'_m} = \frac{S_{11}}{\Psi_m S_{\Psi'_m}} .$$

The major result of this appendix is a generalisation of Theorem A, stated as follows :

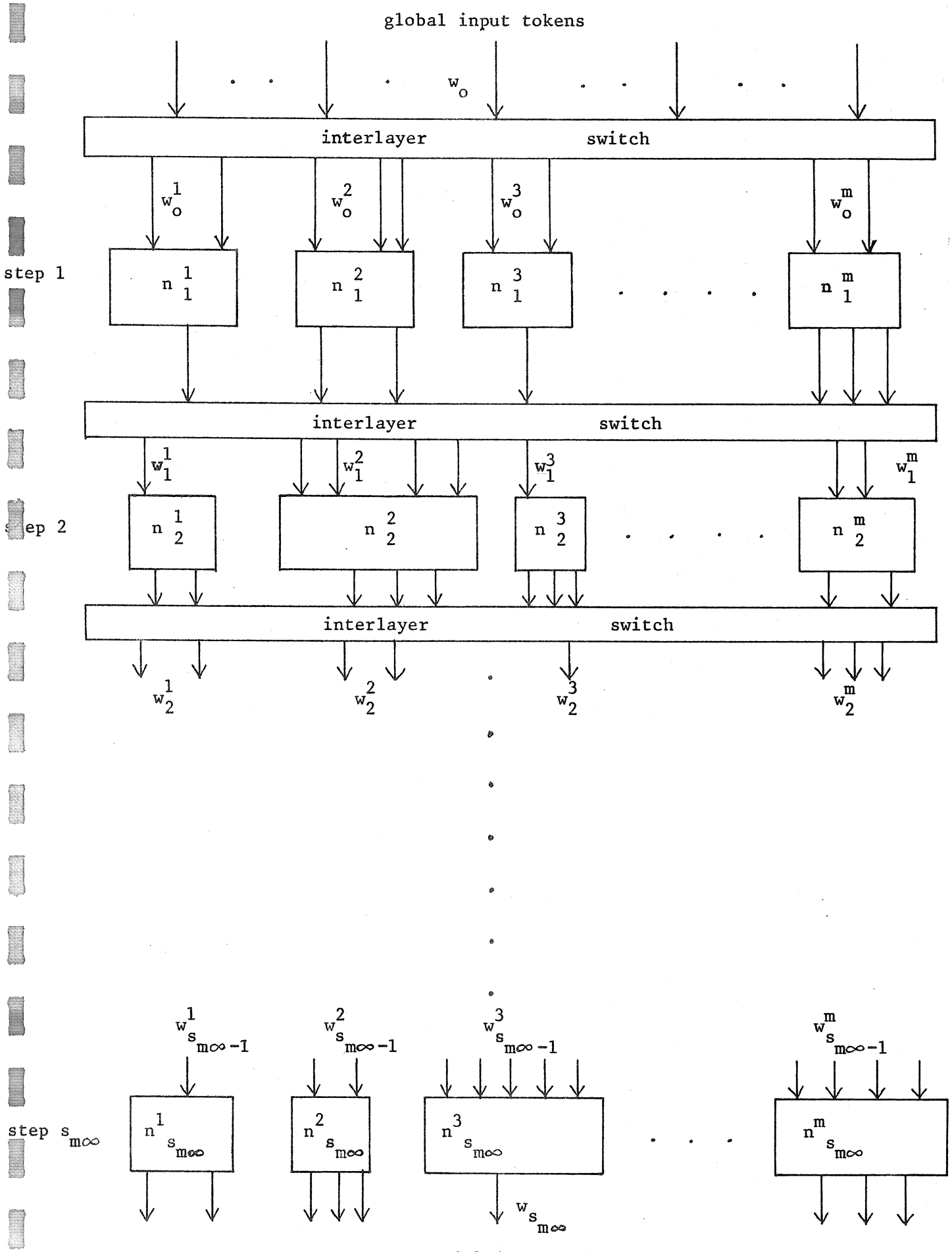


Figure 26 Schematic diagram of an ideal execution, $C_{m\infty}$.

Theorem B :

For any flowgraph program C run on an ideal multilayered system of m layers each containing Ψ'_m processors, $\frac{1}{m+1} < \varepsilon_{\Psi'_m} \leq 1$.

Proof

For finite Ψ'_m, by definition, $S_{m_{\Psi'_m}} \geq S_{m\infty}$. Thus :

$$\varepsilon_{\Psi'_m} \leq 1 .$$

The number of steps in $C_{m\Psi'_m}$ can be determined by considering the steps of $C_{m\infty}$. For the ith step of $C_{m\infty}$, the maximum number of steps in $C_{m\Psi'_m}$ is :

$$\left\lceil \frac{\max(n_i^{\ell})}{\Psi'_m} \right\rceil , \qquad 1 < \ell < m.$$

Since $\max(n_i) \leq \sum_{\ell=1}^{m} (n_i^{\ell})$, we have :

$$S_{\Psi'_m} \leq \sum_{i=1}^{S_{m\infty}} \left\lceil \frac{\sum_{\ell=1}^{m} (n_i^{\ell})}{\Psi'_m} \right\rceil$$

$$< \sum_{i=1}^{S_{m\infty}} \left(\frac{mS_{m_\infty} \sum_{\ell=1}^{m} (n_i^{\ell})}{S_{11}} \right) + \sum_{i=1}^{S_{m\infty}} \Sigma(1)$$

$$< (m+1) S_{m\infty}$$

From which :

$$\varepsilon_{\Psi'_m} = \frac{S_{m\infty}}{S_{\Psi'_m}} > \frac{1}{m+1} .$$

APPENDIX C

Effective Parallelism in the Ring Structured Architecture

The pipelined ring structured architecture described in section 3 is by no means a perfect system in the sense of Appendix A. Hence in this appendix we attempt to describe an approximation to both these systems in order to apply our program analysis and determine the likely performance of the architecture.

The two important characteristics of the perfect system of p processors are :

(i) each instruction takes the same (unit) step time to be executed; and

(ii) all p processors start and finish executing simultaneously.

The reason for insisting on this behaviour in Appendix A was to ensure that the execution C_∞ was well defined. The average behaviour of the ring structured system can approximate the perfect system if we have the following :

(i) an even distribution of different kinds of instruction; and

(ii) a fixed execution time for each instruction (and hence a determinate average execution time for all instructions).

With a well-defined average execution time we can determine the potential step time, δ_s, of the architecture by assessing the total delay, δ_t, around the pipelined ring and adding this to the average execution time, δ_p, to form $\delta_s = \delta_t + \delta_p$. With an even distribution of instructions we can determine the minimum delay, δ_b, between instructions in the pipeline and hence estimate the length of the pipeline as an effective number of stages, $k = \frac{\delta_s}{\delta_b}$. In the following analysis we usually assume that k is an integer. It remains to convince the reader that pipelined access and execution of up to k eligible instructions in ring delay time $(\delta_t + \delta_p)$ is equivalent to completely parallel access and execution of k eligible in step time δ_s. We hope that this equivalence on the average is intuitively acceptable since we cannot offer a con-

vincing proof of its existence. We can, however, suggest two examples which illustrate our argument.

Consider a totally serial computation (i.e. one for which $S_1 = S_2 = S_3 = \ldots = S_\infty$). Each instruction produces tokens which cause the next instruction to become eligible. Only one such instruction becomes eligible in each step. In the "averaged" ring structured system the results corresponding to these tokens must travel from the output of the processing unit, right round the pipelined ring, until they form the next input to the processing unit. There is then an (average) execution delay, δ_p, before the next step commences. Clearly the step time in this case is $(\delta_t + \delta_p)$ and the execution time for the complete program is $S_1(\delta_t + \delta_p)$. The average time to execute an instruction is $(\delta_t + \delta_p)$. Note that the processing unit need contain just one processor.

Consider k identical serial computations (as above) being executed simultaneously. The execution of these computations can be overlapped in such a way as the ith computation starts executing its current instruction, so the result produced by the previous instruction of the ((i+1) mod k)th computation completes its journey around the pipeline ready to commence the next step. The step time is again $(\delta_t + \delta_p)$ and the total execution time is still $S_1 (\delta_t + \delta_p)$. The average time to execute an instruction is now $(\delta_t + \delta_p)/k = \delta_b$, and at least $\frac{\delta_p}{\delta_b}$ processors must be provided in the processing unit.

In both of the above cases, the behaviour of the "average" ring structured architecture is identical to that of the perfect system. We claim that for any program, the behaviour of these two systems is very similar. Consequently, we can determine the general performance of the ring structured architecture by evaluating δ_b, δ_t, δ_p and k, and relating these to the measure of program parallelism, Ψ (see Appendix A and section 3.4.2).

In order to determine δ_b, we consider the execution of a complete program which consumes x gloabal input results, executes S_1 instructions, and produces y global output results. We can determine relationships between the total numbers of data items passing through each part of the system during the time of execution of the program, T. The complete ring structured system is shown in figure 27 with symbols representing these total numbers of data items written alongside the data links between modules. By the obvious action of distributors and arbitrators we see that :

$$r = r_1 + r_2 + r_w$$

$$m = r_1 + r_2$$

$$S_1 = m$$

$$z = o + x$$

$$= r' + y$$

If the program is well-formed (see section 3.2), then, by definition, the matching store finishes empty. Thus :

$$r_w = r_2$$

$$= S_1 - r_1$$

We must now consider the statistical form of the program (i.e. the distribution of various kinds of instruction). The important parameters are :

(i) the number of input tokens consumed per executed instruction (restricted to be either 1 or 2); and

(ii) the number of output results produced per executed instruction (either 0 or 1 or 2).

We can characterise these parameters by three probability constants :

P_{2_i} the proportion of executed instructions requiring two input results;

P_{0_o} the proportion of executed instructions producing no output results;

P_{2_o} the proportion of executed instructions producing two output results.

Naturally the proportion requiring one input result is $(1-P_{2_i})$ and the prop-

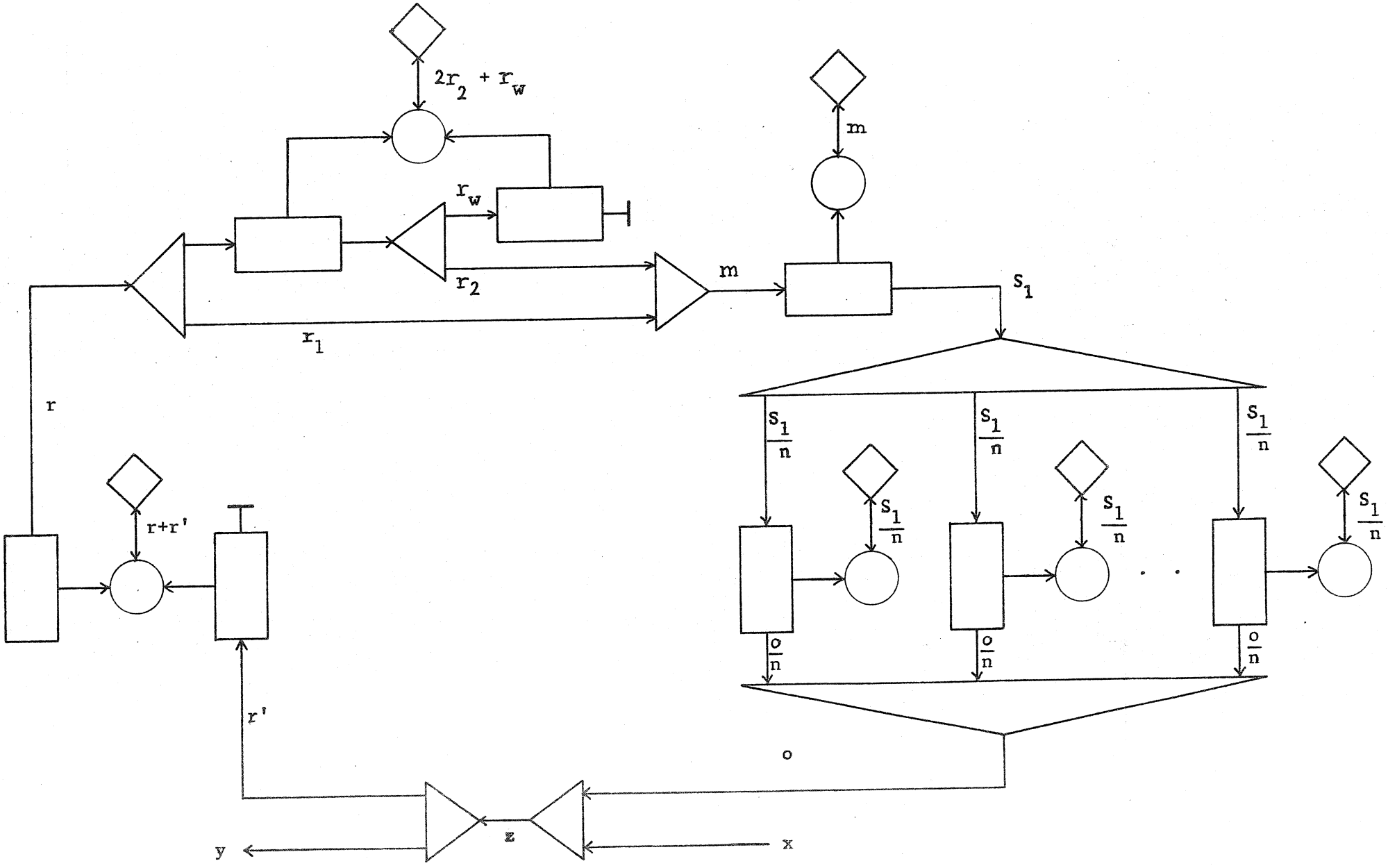


Figure 27 Total numbers of data items passing through the ring structured system during the execution of a program.

ortion producing one output result is $(1-P_{2_o}-P_{0_o})$.

We can now note that :

$$r_1 = S_1 \ (1 - P_{2_i})$$

$$o = S_1 \ (2P_{2_o} + (1 - P_{2_o} - P_{0_o}))$$

$$= S_1 \ (1 + P_{2_o} - P_{0_o})$$

Combining all the above equations, we have :

$$r = S_1 \ (1 + P_{2_i})$$

$$r' = S_1 \ (1 + P_{2_o} - P_{0_o}) + x - y$$

At the end of the execution the result queue is empty. Hence $r = r'$, giving

$$P_{2_i} = P_{2_o} - P_{0_o} + \frac{x - y}{S_1}$$

$$S_1 = \frac{r}{1 + P_{2_i}}$$

The average delay between instructions in the pipeline is given by :

$$\frac{T}{S_1} = (1 + P_{2_i}) \frac{T}{r}$$

where r is the total number of results produced in executing the program and T is the total time taken. The minimum delay between instructions, δ_b, is the minimum value of T/S_1.

To progress further we must consider the (average) delays around the pipelined ring. We will find it convenient to specify a technology-dependent minimum activity time, δ_n, corresponding (roughly speaking) to the delay in a pipeline stage, or the access time of a store module, constructed in a given technology.

If we consider those programs which have little input/output activity (i.e. those for which $x,y \ll S_1$), then it is apparent from figure 27 that the maximum rate of activity is to be found in the result queue, where $r + r'$ $(= 2r)$ store cycles must be made in time T. The minimum period between

any data items in the pipeline is thus T/2r, at the interface to this store. Clearly then :

$$\frac{T}{2r} \geqslant \delta_n$$

Hence :

$$\frac{T}{S_1} \geqslant 2(1 + P_{2_i})\ \delta_n$$

The minimum value of T/S_1 is thus

$$\delta_b = 2(1 + P_{2_i})\ \delta_n$$

when $\delta_n = T/2r$.

Also of interest are the cycle times of the various resources in the pipeline. We may rapidly establish from figure 27 that :

$$t_r \leqslant \frac{T}{2r}$$

$$t_i \leqslant \frac{T}{m} = \frac{T}{S_1} = n(1 + P_{2_i}) \cdot \frac{T}{r}$$

$$t_m \leqslant \frac{T}{2r_2 + r_w} = \frac{T}{3(S_1 - r_1)} = \frac{T}{3P_{2_i}S_1}$$

$$= \frac{(1 + P_{2_i})}{3P_{2_i}} \cdot \frac{T}{r}$$

$$t_p \leqslant \frac{T}{S_1} = n(1 + P_{2_i}) \cdot \frac{T}{r}$$

We would like these resources to handle the most rapid possible rates of instruction execution. Hence we consider the minimum value of T/r in the above expressions. We also place the lower limit δ_n on all cycle times except t_p. We consider the latter to be limited by the minimum of micro-cycles, α, which are required to perform an "average" function. Of course

α will vary from program to program according to the "mix" of instructions. We thus have :

$$\delta_n \leqslant t_t \leqslant \delta_n$$

$$\delta_n \leqslant t_i \leqslant 2n(1 + P_{2_i})\ \delta_n$$

$$\delta_n \leqslant t_m \leqslant \frac{2}{3}\ (1 + \frac{1}{P_{2_i}})\ \delta_n$$

$$\alpha\delta_n \leqslant t_p \leqslant 2n\ (1 + P_{2_i})\ \delta n$$

We now consider figure 28, in which the pipeline stage delays and their relationships to the resource cycle times are shown. Note that every delay must be greater than δ_n. We are particularly interested in the delays δ_p and δ_t ($= \delta_z + \delta_{rr} + \delta_m + \delta_i$). We can see immediately that :

$$\delta_z = \delta_y + \delta_x$$

$$\geqslant 2\delta_n$$

$$\delta_{rr} = \delta_r + \delta_{r'}$$

$$\geqslant 2t_r = 2\delta_n$$

$$\delta_i \geqslant t_i$$

$$\delta_p = \delta_{dp} + \delta_{Pp} + \delta_{ap}$$

$$\geqslant 2\delta_n + t_p$$

To determine δ_m we must appeal to our notion of an "averaged" structure which yields an average delay period comprising two parts, as follows :

(i) with probability $(1 - P_{2_i})$, a delay $\delta_{dm} + \delta_{am}$.

(ii) with probability P_{2_i}, a delay $\delta_{dm} + 2\delta_{mr} + \delta_{mw} + \delta_{am}$

The overall average delay $\delta_m = \delta_{dm} + P_{2_i}\ (2\delta_{mr} + \delta_{mw}) + \delta_{am}$

$$\geqslant 2\delta_n + 3p_{2_i}\ t_m.$$

Hence :

$$\delta_t \geqslant 6\delta_n + t_i + 3P_{2_i}\ t_m$$

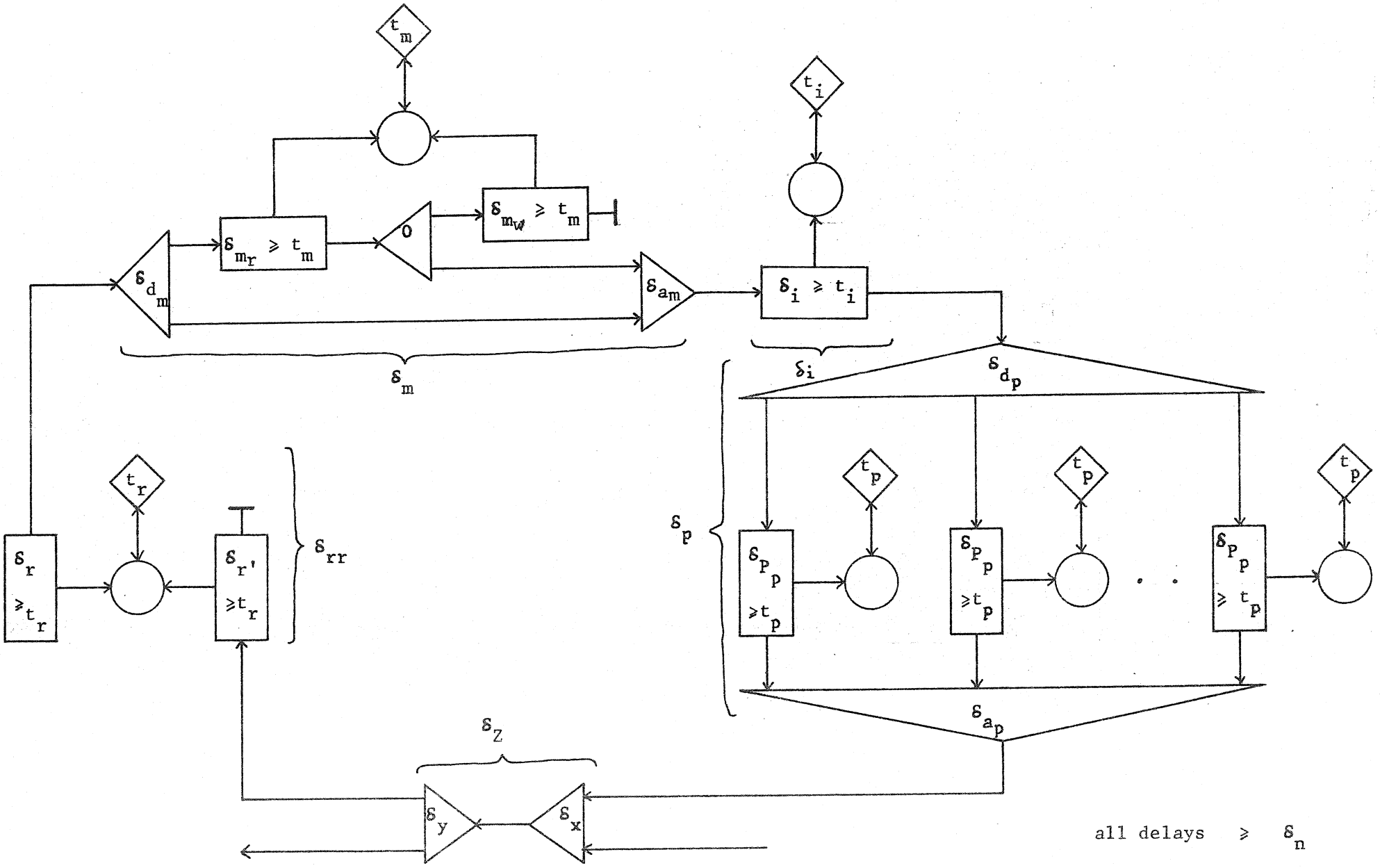


Figure 28 Delays in the ring structured system .

And :

$$\delta_s \geq 8\delta_n + t_i + 3p_{2_i} \cdot t_m + t_p$$

Thus :

$$k = \frac{\delta_s}{\delta_b} \geq \frac{1}{2(1 + P2_i)} \{ 8 + \frac{t_i}{\delta_n} + 3p_{2_i} \cdot \frac{t_m}{\delta_n} + \frac{t_p}{\delta_n} \}$$

For minimum values of t_i, t_m and t_p, of δ_n, δ_n and $\alpha\delta_n$, respectively, we have :

$$k_{min} \geq \frac{\alpha + 9 + 3p_{2_i}}{2\ (1 + P_{2_i})}$$

For maximum values of t_i, t_m and t_p, of $2\delta_n$, $\frac{2}{3}\ (1 + \frac{1}{P_{2_i}})\ \delta_n$ and $2n\ (1 + P_{2_i})\ \delta_n$, respectively, we have :

$$k_{max} \geq n + 1 + \left(\frac{5}{1 + P_{2_i}} \right)$$

Note that :

$$n \geq \frac{\alpha}{2(1 + P_{2_i})}$$

To indicate their expected size, we tabulate the minimum integer values of n, k_{min} and k_{max}, according to the above inequalities, in tables 3, 4 and 5, respectively.

Considering the TTL processing unit design proposed by Zurawski [Zu77], for a program which is dominated by floating point additions and has $P \simeq 1/2$, we find that $\delta_n \simeq 100_{ns}$, $t_p \simeq 3\mu S$ ($\alpha \simeq 30$), and :

$$\delta_b \simeq 300_{ns}$$

$$k \geq 14$$

$$n \geq 10.$$

Using the analysis of section 3.4.2 we also have :

maximum execution rate $(\Omega_{k_{max}}) \simeq 3.33$ MIPS

average execution rate (Ω_k) for programs with $\Psi \geq 14$, always >1.66 MIPS.

		α				
		10	20	30	40	50
P_{2i}	0	5	10	15	20	25
	$\frac{1}{4}$	4	8	12	16	20
	$\frac{1}{2}$	4	7	10	14	17
	$\frac{3}{4}$	3	6	9	12	15
	1	3	5	8	10	13

Table 3 Variation of $\left\lceil \frac{\alpha}{2(1+P_{2i})} \right\rceil \leqslant n.$

		α				
		10	20	30	40	50
P_{2i}	0	10	15	20	25	30
	$\frac{1}{4}$	8	12	16	20	24
	$\frac{1}{2}$	7	11	14	17	21
	$\frac{3}{4}$	7	9	12	15	18
	1	6	8	11	13	16

Table 4 Variation of $\left\lceil \frac{\alpha + 9 + 3 P_{2i}}{2(1+P_{2i})} \right\rceil \leqslant k_{min}$.

		n				
		5	10	15	20	25
p_{2i}	0	11	16	21	26	31
	$\frac{1}{4}$	10	15	20	25	30
	$\frac{1}{2}$	10	15	20	25	30
	$\frac{3}{4}$	9	14	19	24	29
	1	9	14	19	24	29

Table 5 Variation of $\left\lceil n + 1 + \frac{5}{1+P_{2i}} \right\rceil \leq k_{max}$

APPENDIX D

Effective Parallelism in the Multilayered Archtecture

The analysis of aprallelism in a multilayered architecture can be directed along the same lines as that for the single ring architecture presented in Appendix C. However, many of the results are rather unspecific in the absence of sure knowledge of the distribution strategy adopted in the exchange switch. The brief analysis presented in section 4.2.1 is really all that can usefully be presented in a paper of this nature. The purpose of this appendix is to extend the numerical example of Appendix C to encompass a practical multilayer structure.

We have chosen to analyse an $m = 10$ ring system with $\ell = 6$ input/output channels as shown in figure 29. This diagram has not been drawn in detail since the rings are identical to those shown in figures 27 and 28. The important area for study is in the exchange switch where we can see that the delay is some four times greater than that of the input/output switch in the single-ring architecture. The extra delay introduced (if there is no internal buffering in the switch) is $3(\delta_{as} + \delta_{ds}) \geqslant 6\delta_n$. The effective k for each ring will thus be increased by at least $6\delta_n/\delta_b \geqslant 3/(1 + P_{2_i}) \geqslant 2$. Hence the total parallelism in the system is at least 160 (= 10 * (14 + 2)). The maximum possible instruction execution rate is $10/\delta_b \simeq 33.3$ MIPS (for $P_{2_i} \simeq \frac{1}{2}$) with n = 100 processors (each capable of 0.333 MIPS). By an analysis similar to that of section 3.4.2, but using theorem B, any program for which $\Psi \geqslant 160$ will execute in such a way that $\varrho_k > \frac{1}{(10 + 1)}$. Thus the pipeline occupancy is guaranteed to be greater than 9% and the instruction execution rate $\Omega_k > 3.03$ MIPS.

This last figure gives an indication of just how unsatisfactory this analysis of multilayer systems is. Unfortunately it is the best general result we can quote until the optimum strategy for switching between layers has been determined. Work continues in this area.

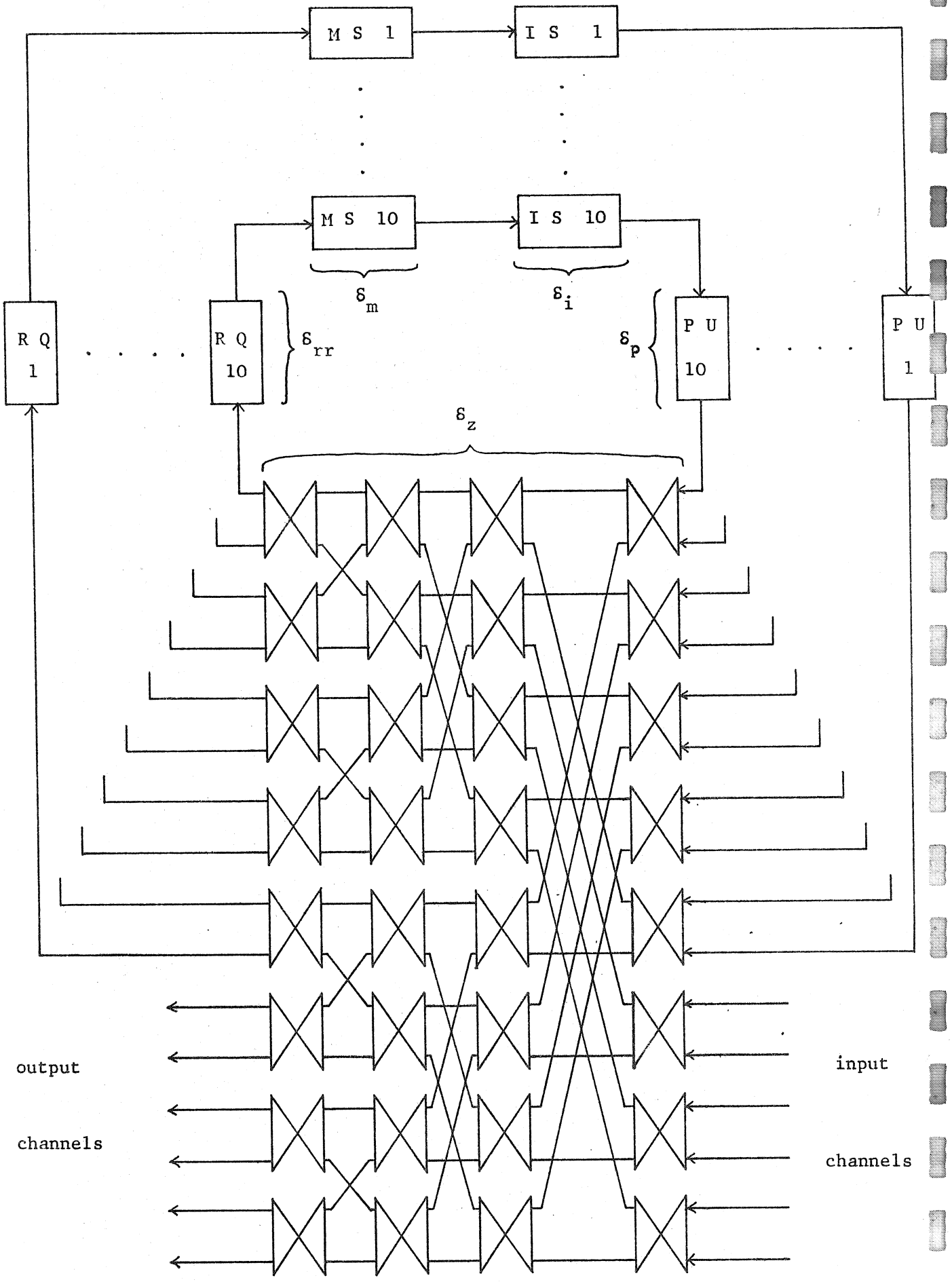


Figure 29 A ten ring multilayered system .

AFFILIATION OF AUTHORS

JOHN GURD and *IAN WATSON* are lecturers in the Department of Computer Science at the University of Manchester, England.

JOHN GLAUERT is at the Computing Laboratory of the University of Cambridge, England.

FOOTNOTES

(1) We discount conventional pipelined systems because they usually employ highly specialised hardware in the pipeline stages, and because they can exploit only a limited amount of parallelism (see Flynn's analysis [Fl72]). We are interested in extensible networks of similar (or identical) processing elements, their configuration and utilisation.

(2) There are a similar class of systems which we call distributed computers (or networks) in which conventional processors are loosely linked and do not share common store directly (e.g. ARPANET). We do not include these as systems for high speed parallel computation since they have been devised with the aim of facilitating the sharing of resources (e.g. giving remote access to file storage or specialised processors such as ILLIAC IV).

(3) This is true of many data flow computers, but the ease of expanding a system varies enormously. We calim that our modular, multilayered system is particularly easy to extend.

(4) It should be noted here that these constraints set this schema apart from many of those mentioned in the introduction. However, they are all equivalent in the sense that they can compute any computable function. The reason for choosing this schema is to limit the size of machine instructions in the hardware system (see section 3).

(5) We make simplifying assumptions in several of our examples, mainly to aid the clarity of the text. In most cases this means that the primitive graphs that are shown are simpler than the code that the LAPSE compiler would generate for the given programs (see section 2.4).

(6) The function ⌈a⌉ (read as "ceiling of" a) defines the smallest integer which is greater than or equal to a.

(7) Notice the unusual firing rules for the MERGE node. It has two input arcs but only requires one token on either (not both) to be eligible. In practice MERGE nodes are not implemented as special functions. Instead two arcs are directed to a common input point. Although this contravenes the regulations regarding computational graphs, the reader can check that the LAPSE constructs specified in this paper render the practice safe.

(8) In a more conventional language, this might have been expressed as a pair of statements :

```
{i,f}  :=  {n,1}   ;
 while old i>1  do {i,f}   :=   {old i-1, old i * old f}  od;
```

However, this form, although safe, breaks the single assignment rule. In Lucid [AW77] we can make plain the difference between the initial assignment and the recurrent assignment using the operators first and next :

```
first {i,f}   ←   {n,1};
next  {i,f}   ←   {old i-1, old i * old f};
```

Unfortunately, the means for extracting the final value of f in Lucid is difficult to implement in data flow primitives. Thus in LAPSE we declare an iteration with formal parameters which are the recurrently assigned identifiers, and correspond to the actual parameters which

are the initial input to the iteration and also the eventual output. In the example given, the statement :

{dummy,facn} ← factorial (n,1)

indicates that the iteration is given the values n and 1 (as i and f, respectively) initially, and that when it terminates, the then current values of i and f will be sent to dummy and facn. It can be seen that for n>1 dummy will receive the value 1 at that time.

(9) Note that only one output arc of a BRANCH node can receive a token for each activation of the function.

(10) The reader may quickly verify that such a restriction will prevent the described situation from arising.

(11) This simple method will only handle single loops. Nested loops must be embedded in function calls which are discussed in section 2.4. In the example given in figure 6, INCREMENT-ITERATION-LEVEL nodes would be placed at points X and Y on the graph. In order to restore the original label at the output, we insert ZERO-ITERATION-LEVEL nodes on the output arcs.

(12) The net effect of the identifier is to colour the tokens inside a function [De74].

(13) The output section of the graph contains pairs of arcs and nodes because the output argument is a record containing two values.

(14) This can be shown to be the minimum possible asymptotic time for this computation. However, the given algorithm is not the only one which can achieve this speed. It is nonetheless neater than the iterative version suggested in footnote 18.

(15) Contrast this with the conventional vector addition program :

for i:= 1 to n do a [i] := b [i] + c [i] od

in which sequence is unnecessarily overspecified.

(16) Please suggest a "better" program for this operation. For example could the array s have dimension [1 ..k]? Note that this algorithm can be modified to perform an iterative evaluation of n! in logarithmic time.

(17) This is not a property language but of the translator.

(18) This time may alter for different programs. The range of execution times for various instructions is large (e.g. "floating point divide" versus "increment iteration level") and the mix of instructions is unpredictable.

(19) This is comparable to the traditional measure of computer speed, i.e. millions of instructions per second (MIPS).

An Efficient Pipelined Dataflow Processor Architecture

Jack B. Dennis
Lab. for Computer Science
M.I.T.

Guang R. Gao

Technical Report TR-SOCS-88.06
Feb. 1988

McGill University
School of Computer Science
805 Sherbrooke St. W.
Montreal, Quebec, Canada
H3A 2K6

(514) 398-4446

Contents

List of Figures

An Efficient Pipelined Dataflow Processor Architecture

Jack B. Dennis
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
617-253-6856

Guang R, Gao
School of Computer Science
McGill University
Montreal, H3A 2K6
514-398-4446

Abstract

This paper discusses design principle and issues for a pipelined dataflow processor architecture. Our goal is to show that the principles of pipelined instruction execution can be effectively applied in dataflow computers, yielding an architecture that avoids the main sources of pipeline gaps during program execution encountered in many conventional processor designs. This is done by showing that the fine-grain instruction scheduling mechanism derived from the data-driven model of program execution can be implemented in a processing element design that keeps the instruction execution pipeline filled because all instructions held in the processor are candidates for initiation. We argue that this unique program execution power can be effectively exploited in many problems of scientific computation by constructing of machine code blocks that correspond to array definitions in the source program.

The new processing element design uses an architecture called *argument-fetch dataflow architecture*. It has two parts: a *dataflow instruction scheduling unit* (DISU) and a *pipelined instruction processing unit* (PIPU). The PIPU is an instruction processor that uses many conventional techniques to achieve fast pipelined operation. The DISU holds the dataflow signal graph for the collection of dataflow instructions allocated to the processing element, and maintains a large pool of enabled instructions available for execution by the PIPU.

The new architecture provides a basis for achieving full performance for many scientific application codes through use of a compiler able to perform global dataflow analysis and identification of array definitions. Its advantages over both conventional RISC architectures and vector processor architectures are outlined. To show that the realization of an efficient dataflow processing element is feasible, a trial design and fabrication of an *enable memory*–a key component of the DISU–is reported.

1 Introduction

This paper discusses design principles and issues for a pipelined dataflow processor architecture. We wish to show how the principles of pipelind instruction execution can be effectively applied in dataflow computers, yielding an architecture that avoids the main sources of pipeline gaps during program execution encountered in many conventional processor designs. This capability stems from the fine-grain instruction scheduling mechanism derived from the data-driven model of program execution which can be implemented in a processing element design that keeps the instruction execution pipeline filled because all instructions held in the processor are candidates for initiation. This unique program execution power can be effectively exploited in many problems of scientific computation by constructing of software pipelined machine code blocks that correspond to array definitions in the source program.

The new processing element design uses an architecture called the *argument-fetch data flow architecture*. It has two parts: a *dataflow instruction scheduling unit* (DISU) and a *pipelined instruction processing unit* (PIPU). The DISU holds the data-flow signal graph for the collection of dataflow instructions allocated to the processing element, and maintains

a record of which instructions are enabled by satisfaction of conditions for execution. This pool of enabled instructions is available for execution by the PIPU.

The PIPU is an instruction processor that uses conventional techniques to achieve fast pipelined operation. The principle of RISC architecture that no instruction is initiated unless it has no data conflict with instructions in the pipeline is honored by this processing element since the dataflow model guarantees that no pair of simultaneously enabled instructions can be in conflict over data. Thus pipeline gaps caused by conditional branch instructions cannot arise. In some RISC architectures, the compiler is expected to ensure that no pair of instructions in the pipeline are in conflict over data. In the dataflow processor, this requirement is satisfied by obeying the dataflow model. In the RISC architecture, only a small window of instructions can be considered for initiation, whereas in the dataflow architecture, thousands of instructions from different code blocks are considered candidates for initiation.

The argument-fetching architecture provides a basis for achieving full performance for many scientific application codes through use of a compiler able to perform global dataflow analysis and identification of array definitions. The architecture has a significant advantage over vector pipelined computers is that it is no longer essential that the array definitions be expressible as vector operations to achieve peak performance.

We begin our discussion in Section 2 with an overview of our current design (architecture) for a dataflow processing element and a discussion of how it differs from our earlier argument flow architecture. The principles and organization of the current design are discussed in greater detail in Section 3. In Section 4 we compare the dataflow design with other pipelined processor designs and explain why the dataflow design is able to keep the pipeline full where other designs fail. In Section 5 we illustrate how array definitions can be converted into efficient code blocks that can fully utilize the performance potential of pipelined instruction processing units. To show that the realization of an efficient dataflow processing element is feasible, we report in Section 6 on a trial design and fabrication of an *enable memory*–a key component of the DISU.

Functional programming languages are best suited for expressing computations for par-

allel computation because programs written in a functional language can be analyzed easily to determine the flow of data between elements of the program, and array definition that correspond to dataflow machine code blocks are readily identified. In this paper we use the functional language Val [1,9] to display our illustrative programs.

2 Contrasts in Static Dataflow Architecture

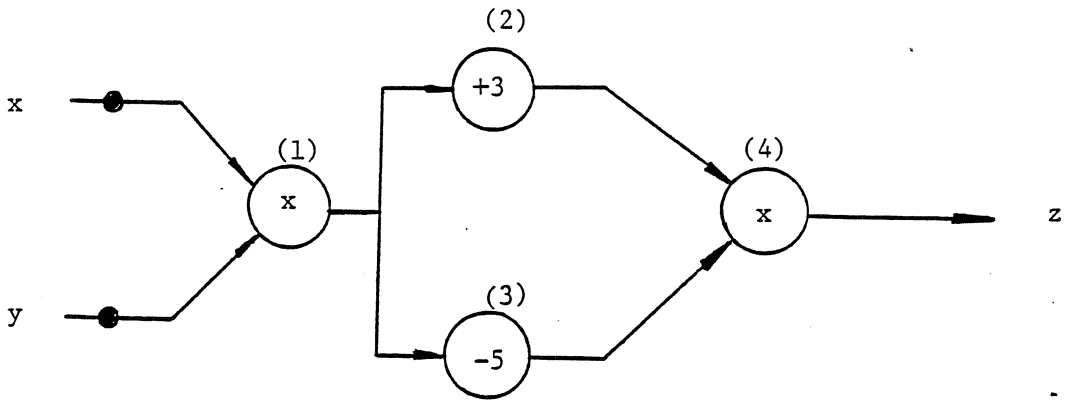
We begin our discussion in Section 2 with an overview of our current design (architecture) for a dataflow processing element and how it differs from our earlier processing element architecture. The principles and organization of the current design are discussed in greater detail in Section 3.

2.1 Efficiency Issues in the Argument-Flow Architecture

Dataflow computer architectures are founded on the operational semantics of dataflow graphs. In static data flow model, an actor becomes enabled when data tokens become available on its input arcs and its output arcs (the places for holding result tokens) are empty. The *firing rules* for static data flow graphs [6] state that any enabled actor may be chosen to fire. Figure 1 illustrates the operation of a static data flow graph for computing the expression $z = (xy+3)(xy-5)$. According to the firing rule, actor 1 is enabled because both of its inputs are available, and each of its result arcs is empty.

In a dataflow processor, instructions loaded into the processor represent, more or less directly, actors of a dataflow program graph. In the argument-flow dataflow architecture, known as a *dataflow circular pipeline*, such an instruction typically has two spaces to receive operand values, and a field that holds a *destination list* indicating the target instructions to which result values or signals are to be sent. The arrival of a result value corresponds to placing a token on the input arc of an actor; the arrival of a signal (an *acknowledge signal*) from a successor instruction means that a result token has been removed from one of an actor's output arcs.

Note that the delivery of a result value to the operand receiver of an instruction template



z = (xy + 3) (xy - 5)

Figure 1: A static data flow graph

accomplishes two purposes: (1) its signals to the target instruction that an input is available; and (2) it transmits the data value itself from an instruction to one of its successors.

A static dataflow processing element based on the argument-flow architecture is illustrated in Figure 2 [3]. Dataflow instruction templates are stored in the *activity store*, and the queue holds the addresses of those instructions which are enabled. The fetch unit picks the address of some enabled instruction, reads an operation packet (containing operand values as well as the destination addresses) from the activity store, and delivers it to an operation unit where the scalar operation called for is applied. Each destination address gets paired with a copy of the result value and forwarded to the update unit. The update unit places values from result packets in the templates of their target instructions and determines whether they are now enabled. The communication between the fetch unit, the execution unit and the update unit is through the sending and receiving of data packets (operation packets and result packets).

An issue in this processor architecture is that more data movement is involved than seems necessary. Two cases in point are: (1) the destination list passes through the fetch unit and the operation unit although information in the list is not acted upon by either unit; and (2) result values are copied and stored in duplicate whenever there is more than one target instruction.

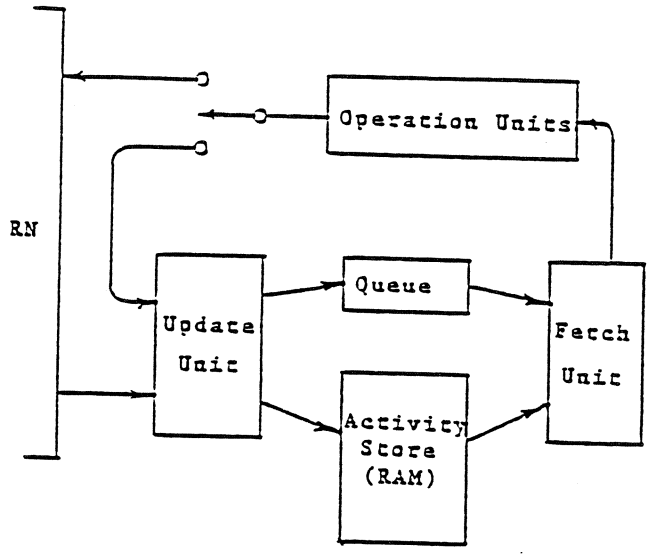


Figure 2: An argument-flow dataflow processing element

These inefficiencies arise from the decision to keep data information (values) and control information (addresses) bound together in packets as they traverse the circular structure of the processor.

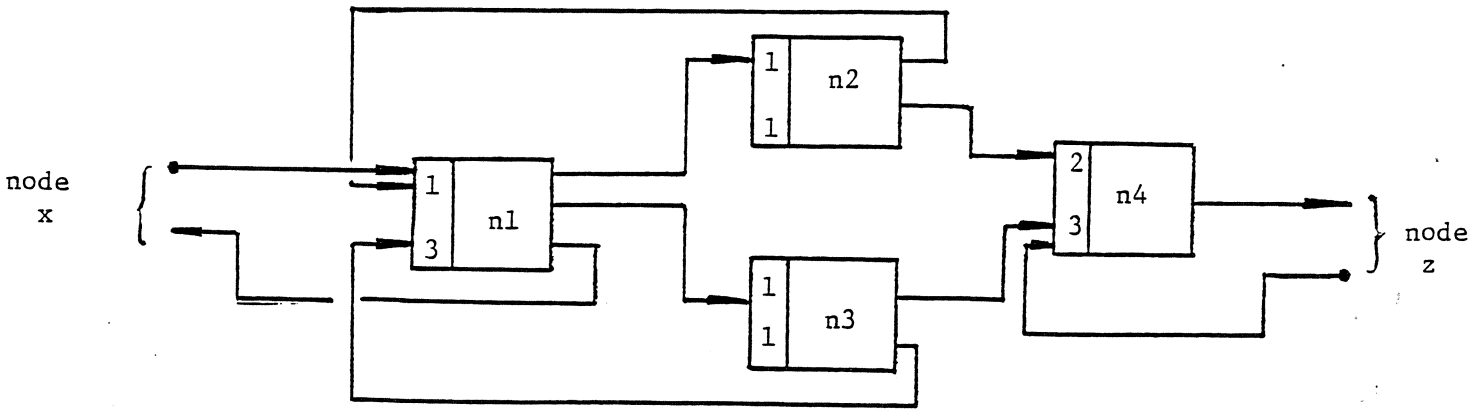
We have learned to view the program execution scheme of Figure 2 as an abstract specification of the activity a dataflow processor should perform. In the course of considering a great variety of hardware schemes for realizing the behavior of dataflow graphs, we discovered the advantage of separating the data and signaling roles of the information packets. Once this has been done, it becomes evident that it is better for an instruction to fetch its own arguments from a data memory than for its predecessor instruction(s) to store result value(s) in the operand fields of several target instructions. The result has evolved into the *argument-fetch architecture.*

Before presenting the architecture, we introduce the *signal graph* model which serves as an abstract model for the *argument-fetch* architecture just as dataflow program graphs provide an abstract model for the argument flow architecture.

2.2 Data driven signal graphs

The signal graph representation of the data flow graph G in Figure 1 is shown in Figure 3. Formally, this dataflow program is a *program tuple* (P, S) in which the *instruction graph*

(a) signal graph s



(b) set of instructions p

```
n1 :  x   x   y   u
n2 :  +   3   u   v
n3 :  -   u   5   u
N4 :  x   v   w   z
```

Figure 3: A data flow program tuple (P,S)

P is a list of instructions each corresponding to one actor of G. The instructions in P do not contain information about the sequencing of instruction execution. Instead, the sequencing information is given separately by the *dataflow signal graph* S. Although the signal graph shown in Figure 3 looks similar to the original data flow graph G, the signal graph S has the following features and differences:

1. each node in S contains a reference to an instruction in P;

2. a node does not contain information about instruction execution; that is specified by reference to the instruction list P;

3. the arcs in S (called *signal arcs*) represent only the sequencing information, i.e. an arc from actor u to actor v specifies that a signal is to be sent to v after actor u is fired; the signal arcs do not specify the treatment of result values;

4. each node in S holds two fields that are part of the dataflow instruction scheduling mechanism: the *count* field indicates how many signals must yet be received for the

node to become enabled; the *reset* field gives the value to be placed in the count field when the node is fired.

In Figure 3, for example, node 1 in S contains a reference n_1 to the instruction (* x y T1) in the set P. The figure also gives the count and reset values for each node. The reset value for node 1 is three because it must receive signals from nodes x and y and two from nodes 2 and 3) before it becomes enabled. However, its count field is only two because signals from node 2 and node 3 are not required for the first firing of node 1.

The individual instructions in P may be interpreted as in a von Neumann computer that uses three-address instructions. For example, the instruction at address n_1 is a three-address ordinary floating-point multiply instruction with operand addresses x and y, and the result address u. During program execution, the fetching and storing of operands and results in data memory happen as in a conventional processor. The primary difference is the mechanism used to select instructions for execution. Instead of using a program counter that advances through a list of instructions selecting one at a time, the dataflow processor uses the signal graph of a program to identify the subset of enabled instructions among the entire collection of instructions held by the processing element.

2.3 Conditional signal graphs

Next we consider the representation of conditional expressions in dataflow models for computation. The following example is a conditional expression written in Val.

$$\text{z} = \textbf{if i} = 1 \textbf{ then i} + \text{x } \textbf{else i} - \text{x } \textbf{endif}$$

The static data flow graph for this expression is shown in Figure 4. The test actor (node 1) produces boolean values used by the T and F actors to select/deselect values of x and i for use by node 2 or node 3. The boolean values also tell the merge actor (node 8) the order in which to forward its input values to the output of the graph [5]. In the argument-flow architecture, the T, F, and merge actors of the dataflow graph are represented explicitly as specialized instructions of the dataflow machine code. The need for these specialized instructions is avoided by use of the argument-fetch model of dataflow program execution.

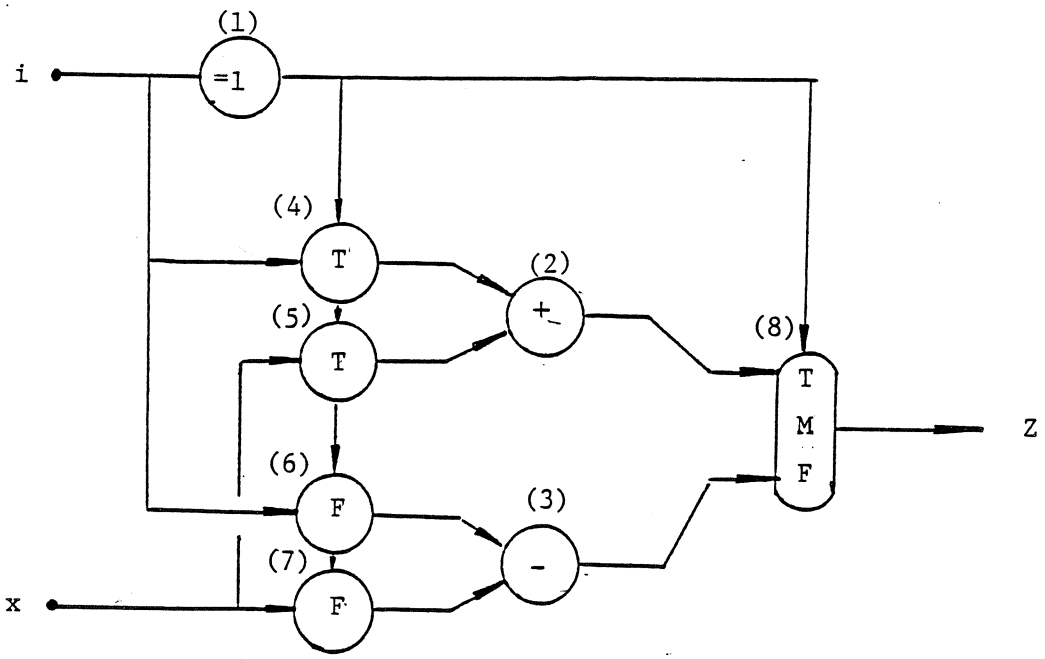


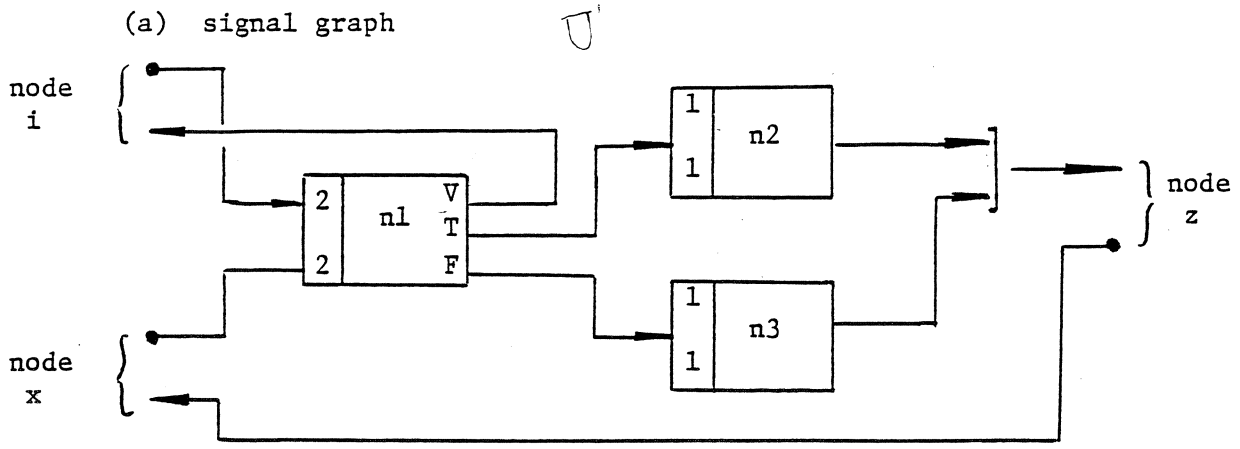
Figure 4: A conditional dataflow graph

The corresponding program tuple (P, S) for the conditional expression is shown in Figure 5. In this model, ***conditional signaling*** takes the place of the T, F, and merge actors. The test instruction of node 1 produces a boolean result that determines which successor nodes receive signals (T means signal if true; F means signal if false; U means signal unconditionally). When node 1 fires, node i is signaled unconditionally, and node 2 or node 3 is signaled depending on whether or not $i > 1$. The merge is implemented by having instructions n_2 and n_3 use the same result address z. Therefore, the destination node z will see only the "merged" result. Note that node z must send a signal back to node x to indicate that it is safe to generate and store a new value at location x.

By eliminating the need for special instructions to implement conditional expressions, the argument-fetch model provides a considerable saving in the number of instructions that must be executed to carry out computations.

2.4 Argument-fetch model vs. argument-flow model

The separation of the instruction scheduling and instruction execution functions in the signal graph model yields important simplifications in implementation of a dataflow



(b) set of instructions

```
n1 :  >   i   1   j
n2 :  +   j   x   z
n3 :  -   j   x   z
```

Figure 5: A conditional argument-fetching data flow program (P,S)

processing element. It is now possible to separate these two functions in the hardware structure as well. This avoids the tight coupling of control and data information into operation and result packets that leads to unnecessary data movement in the argument-flow architecture. In the argument-fetch data flow execution model, the function of instruction scheduling is no longer tightly-coupled into the actual data path of the architecture.

3 The Argument-Fetch Dataflow Processor

The argument-fetch dataflow processor consists of two major sections (Figure 6), each designed to perform its separate function efficiently. The Dataflow Instruction Scheduling Unit (DISU) implements the signal graph of the dataflow program and identifies instructions that are available for execution. The Pipelined Instruction Processing Unit (PIPU) executes enabled instructions and informs the DISU when each instruction finishes execution.

The *fire* link in the figure is for transmitting the addresses of enabled instructions from the DISU to the PIPU. The *done* link is for transmitting the done signals containing the

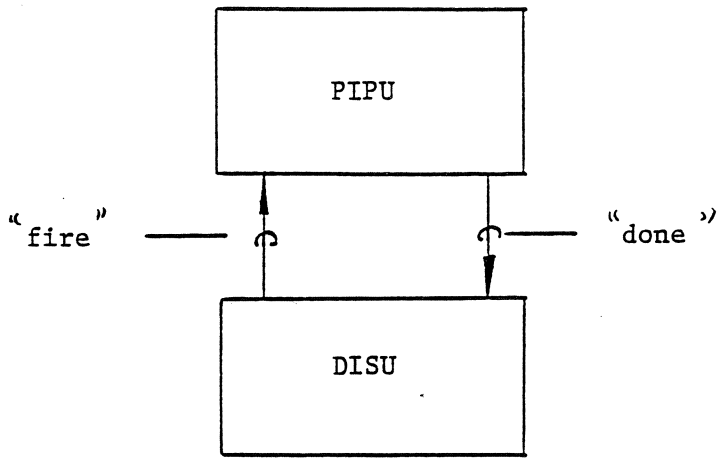


Figure 6: A argument-fetching data flow processor

address of each instruction which have completed their processing in PIPU, together with a *condition code* used by the DISU to control the sending of conditional signals.

Since many instructions will be processed simultaneously by the PIPU, many fire signal will be sent before the first one is acknowledged by a done signal. In a balanced design, the DISU will be able to supply fire signals to the PIPU just fast enough to keep the instruction execution pipeline operating continuously at full capacity.

3.1 The pipelined instruction processing unit: PIPU

The organization of the PIPU is shown in Figure 7. It consists of four major pipeline stages that implement the usual functions of instruction fetch, operand fetch, operation execute and, result store. We assume that the readers are familiar with the basic principles of pipelined SISD architecture (for background knowledge see Chapter 6 of [18]). In this design, any effective address calculation is included in the instruction fetch stage, and the scalar operation unit performs the function of selecting and storing elements of arrays.

The unique character of the PIPU stems from the absence of a program counter and its related control logic. Since no branch instructions are supported in the dataflow architecture, there is no concern about pipeline gaps created by unknown branch outcomes.

Each of the major pipeline stages may be internally pipelined to process some of the

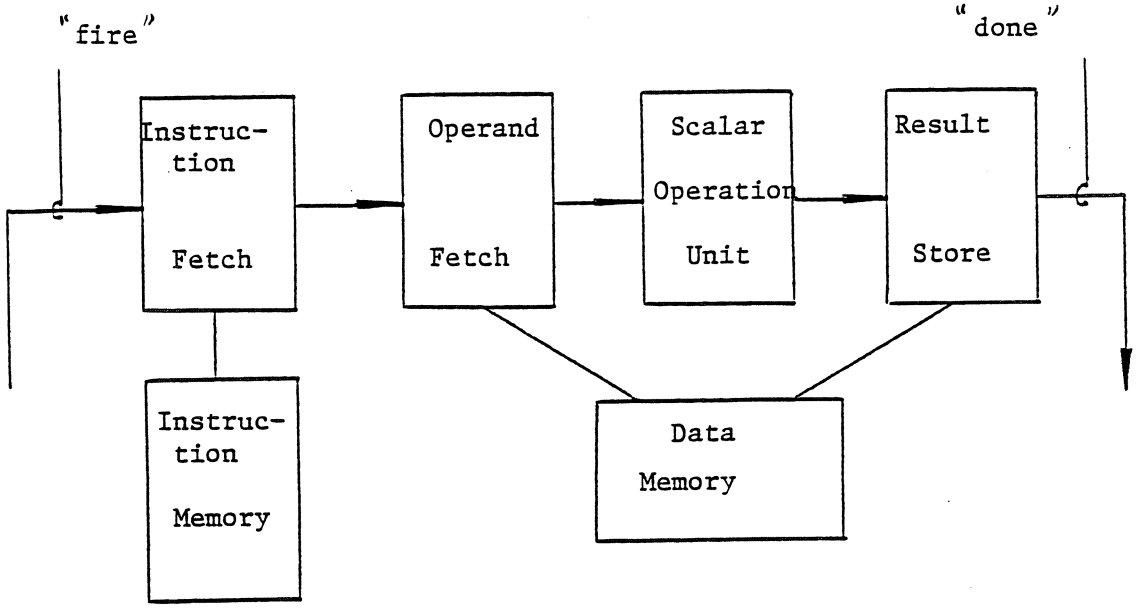


Figure 7: The structure of PIPU

more complex functions, floating point arithmetic operations or fetch accesses to a large data memory, for example. Since different work units may require more or less clock cycles to pass through a major pipeline stage, buffering of work units will be helpful to smooth out the flow of work units while allowing simple instructions to be completed as early as possible.

In Section 4, we compare the pipelining in PIPU with other conventional processor designs, and illustrate how the problems intrinsic to the latter are effectively overcome.

3.2 The dataflow scheduling unit: DISU

The structure of the DISU is shown in Figure 8. It consists of a *signal processing unit* (SP) and an *enable controller unit* (EC). The signal graph of a program is represented in the DISU by information stored in a *signal list memory* (SLM) of the SP unit and the *enable count memory* (ECM) of the EC unit. The SLM holds a *signal list* for each node of the signal graph, and the ECM holds the count and reset values for each node. A signal list represents the set of signal arcs leaving the associated node of the signal graph.

In response to a done signal for instruction n_i, the SP unit reads the corresponding

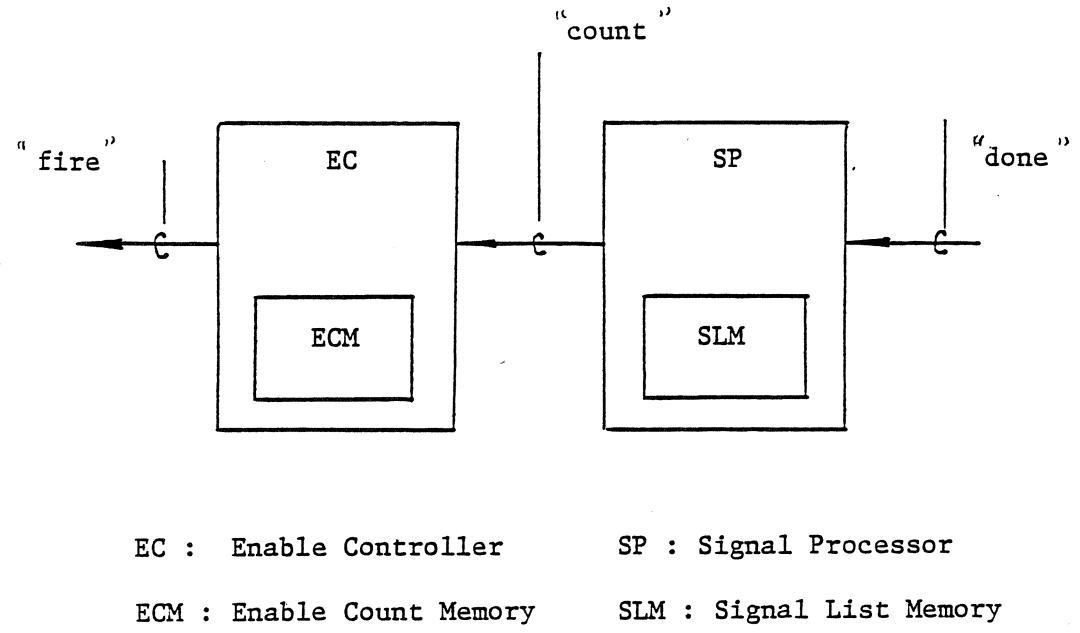


Figure 8: The structure of DISU

signal list and sends a *count command* to the EC unit for each active entry in the list. (The set of active entries may depend on the condition code that accompanies the done signal.)

The EC unit handles each count command by decrementing the count value for the indicated node and testing for zero. If the count becomes zero, an enable flag for the node is set and the count value is set to the reset value. The EC unit continuously monitors all of the enable flags and issues fire commands for enabled nodes. Since the mechanism used to select enabled instructions for execution is crucial to obtaining full performance of the dataflow processor, further explanation of the EC unit and a discussion of an experimental implementation is provided in Section 6.

4 Eliminating Pipeline Gaps

4.1 Conditional branch

Branch instructions raise significant problems in the design of high performance pipelined computers because they interfere with the steady flow of executable instructions into the pipeline. Conditional branches are especially troublesome because the correct choice of

successor instruction may depend on a result yet to be produced by an instruction still in the pipeline. Some studies have shown that if branch instructions appear in only one out of ten instructions, throughput could be reduced by a factor of three. [20], and the dynamic frequency of branch instructions may be very high [2,27].

Computer architects have invented several techniques to reduce the performance impact of conditional branches in conventional pipelined processors. One important technique is the *delayed branch* whereby a branch transfer does not take place until some fixed number k of instructions following the branch instruction have also been executed [19]. The challenge is given to the programmer or compiler designer to find instructions that can make effective use of the k instruction initiation cycles during which the branch outcome is unknown. Although the idea is old, it has only recently been applied in practical machine designs, some recent RISC architectures in particular [17,22].

Another method is using *branch prediction* where the branch outcome is assumed to be true, and the execution is allowed to proceed on that assumption [19]. If the branch outcome turns out to be false, execution is backed up to cancel any effects of the unneeded instructions. The compiler must generate code so the most likely branch outcome is true. If the compiler cannot predict well, the cost of implementing branch prediction may not be justified.

A third idea is to make predictions during program execution based on the branch history of the program. In implementation, a branch history table is needed to record the recent history of branches. An example is the *branch target buffer* in the MU5 computer [20].

Each of these methods reduces the effect of conditional branches on pipeline performance, but none solves the problem completely. The complexity of control required and the hardware cost involved have been a constant source of debate.

4.2 Pipeline gaps from data dependencies

The performance of pipelined processors is limited by data dependencies. Operand fetch for an instruction must not occur until the operand values have been stored. If an operand

is the result of a previous instruction, execution of the current instruction must be delayed until the previous instruction completes its execution.

Hardware and software solutions to this problem can be used in conventional pipelined computers: hardware solutions employ *reservation stations* where an instruction waits for its operands while subsequent instructions may proceed [25]. Software solutions use compiler code scheduling techniques to insure that the dependency distance is sufficiently large that hazards are impossible. Code scheduling is done by reordering instructions and inserting dummy instructions [16].

4.3 Pipeline gaps in dataflow architecture

The dataflow architecture completely changes the nature of these problems. In the argument-fetch architecture, no branch instruction ever enters the execution pipeline, and no instruction is enabled if it uses a result of an instruction that is in execution. This is done by being sure that every data dependence is represented in the signal graph. This is the responsibility of the compiler.

In the dataflow architecture, the only way for gaps to arise in the execution pipeline is that the DISU runs out of enabled instructions. Keeping the execution pipeline full is accomplished simply by being certain that sufficient instructions are enabled at all times. Since any enabeled node of the signal graph may be chosen for firing by the DISU, the issue becomes one of being certain that sufficient parallelism is exposed in the dataflow machine code.

5 Parallelism in Code Blocks for Array Definitions

Despite its attractive features, the dataflow model of computation has raised considerable controversy among researchers. Criticism of dataflow architectures for high-performance numerical computation has centered on array computation. As stated in [10], the principal skepticism about the ability of dataflow computers for handling array operations owes to their emphasis on fine-grain, operational-level concurrency [10,11]. In this section, we out-

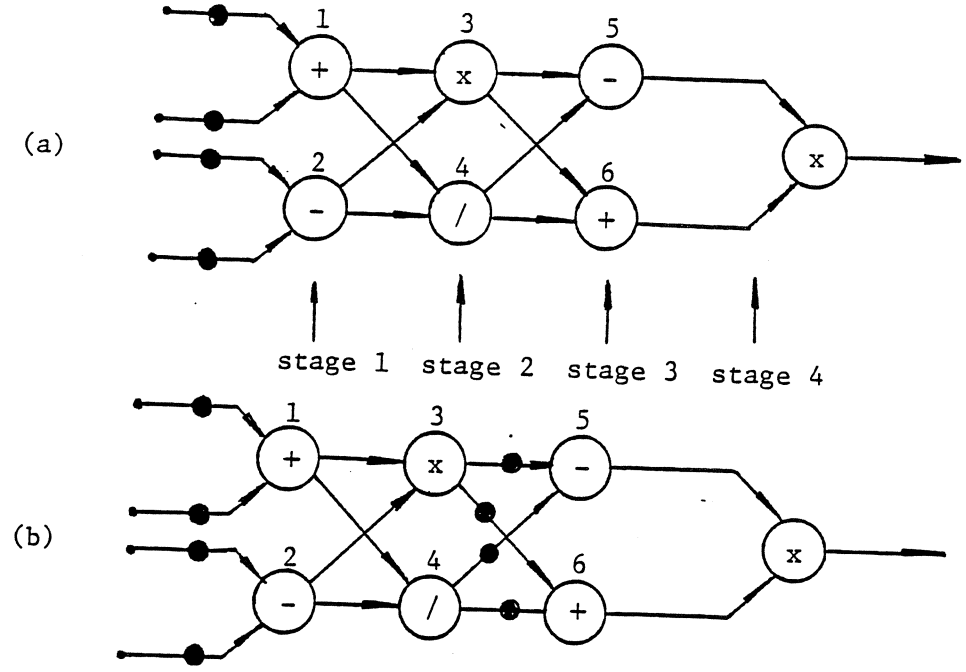


Figure 9: Pipelining of data flow programs

line how arrays (vectors) can be efficiently handled by the proposed dataflow architecture.

5.1 Software pipelining in dataflow programs

The fine-grain instruction scheduling implemented in the dataflow architecture makes possible a very efficient form of software pipelining. This is illustrated by the dataflow graph shown in Figure 9.

Here, seven actors are grouped into four stages. In Figure 9 (a), actors 1 and 2 are enabled by the presence of tokens on their input arcs, and thus can be executed in parallel. Parallelism also exists between actors 3 and 4, and between actors 5 and 6. In the static dataflow model, pipelining means arranging the machine code such that successive computations can follow each other through one copy of the code. Figure 2 (b) show a configuration of the dataflow graph in which the first work unit has advanced to stage 3 and a new work unit (represented by four tokens) is presented to stage 1. Now two sets of tokens are in the pipeline, and the actors of stages 1 and 3 are both enabled and can be executed concurrently. Note that this dataflow graph is a software pipeline because the arcs drawn between actors correspond to addresses in stored dataflow machine code and

not to wired connections between logic elements.

The power fine-grain parallelism for large-scale numerical computations derives from the possibility of large dataflow pipelines in which hundreds of actors in many stages are executed concurrently. We have found that the major sections of several benchmark codes for solving partial differential equations are readily expressed as pipelined dataflow machine code blocks.

5.2 Pipelining of array computations

Computations using arrays of values have a central role in high performance scientific computation. For this reason, vector pipeline processors have dominated the field of supercomputers for many years. These machines can achieve very high peak performance rates for jobs that consist solely of simple operations on vector values, principally element-by-element addition or multiplication. Moreover, vector operations can be *chained* so that, for example, a vector addition may directly use the result vector of a vector multiplication.

A very general form of vector processing is supported by the dataflow architecture. We shall illustrate it by reference to Figure 9. In dataflow computation it is useful to regard an array value as the sequence of its element values carried by successive tokens on a single dataflow arc. In particular, we may suppose that the four input arcs of the dataflow graph each carry the successive elements of four arrays, A, B, C, and D, and the output arc carries successive elements of a computed array X. During the computation, each actor of the dataflow graph performs a simple vector operation: actor 1 does a vector addition, actor 2 does a vector subtraction, etc.

In this way, considerable flexibility is gained. Any instruction execution resources that cannot be utilized by one dataflow pipeline are available to other code blocks loaded into the same processing element. Unlike the vector operations usually provided in a vector processor, there is no requirement that the activities of one such vector operation be continuously processed by one of a group of dedicated function units. The fine-grain scheduling of the dataflow machine allows flexible scheduling of the execution of enabled actors in the pipeline.

The class of expressions that can be compiled into pipelined dataflow machine code is much more general than the class of expressions supported by vector pipelined processors. For example, the dataflow graphs that describe the general element of the result vector may contain conditional expressions, and the indices of the elements of the input vectors need not be simply related to the index of the output element being computed by a pipeline work unit. This flexibility is available because each operation by the dataflow processor is separately controlled. Even so, it is still possible to keep the instruction pipeline full because enabled instructions from all parts of the program allocated to the processing element are available for execution.

In the Val language, expressions that lead to pipelined dataflow code blocks are conveniently written as *forall expressions*. The automatic translation of such high-level programs into efficient machine code blocks has been studied and a code mapping scheme has been proposed in [13].

6 An Experimental Enable Memory in a VLSI Circuit

In the dataflow processor architecture, most components are standard sorts of digital hardware–memories, floating point functional units, and controllers. The controllers in turn comprise datapaths, sequencers, and combinational logic blocks. These kinds of logic components are familiar elements of conventional computers. One component, however, has no counterpart in conventional architectures. This is the the enable memory. It must hold the flags for several thousand dataflow instructions and must be able to select enabled instructions for initiation at a very fast rate. To meet performance demands, both for updating status records and for selecting enabled instructions, a very special design is required.

At M.I.T. an experimental enable memory device was designed to support 128 dataflow instructions. The architecture of the chip is illustrated in Figure 10.

It consists of an 8 by 16 array of one-bit cells (the *Enable State Memory* ESM) and specialized control logic for the count updating and selection functions. For each count

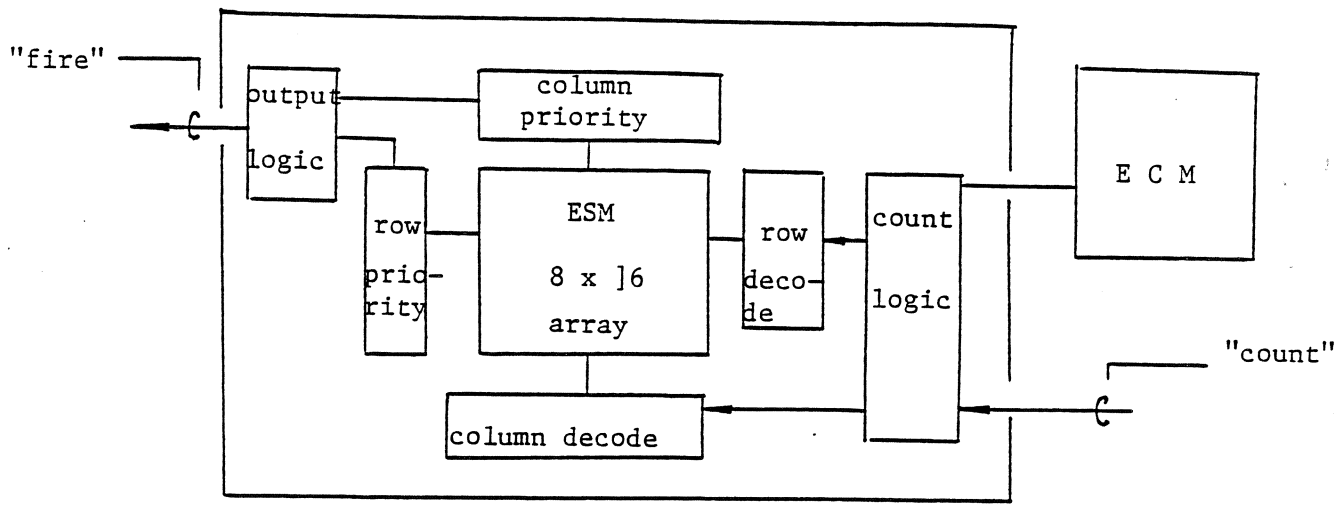


Figure 10: An architecture for the enable memory

command, the count logic decrements the corresponding count field in the Count/Reset Memory. If the count becomes zero, the corresponding bit in the Enable State Memory is set, indicating that the instruction is enabled. At this time the reset field is copied into the count field to prepare for the next cycle of enabling of the instruction.

The selection of instructions for execution is done by the row and column priority logic which is carefully designed to ensure "fairness". The row logic selects each row in turn, and reads the contents of any non-zero row into the column logic. No row is considered again until all other rows have been examined. The column logic scans the selected row from left to right, issuing a fire command for each bit that is on.

This selection scheme is fair in the following sense: if some enabled cell is selected to fire at time t, and if the same cell becomes enabled again at t', it will not be selected to fire again until all cells enabled during the time interval (t,t') have been selected to fire. The motivation for implementing a fair firing mechanism is to be sure that the machine does not repeatedly fire a group of instructions without giving attention to the other enabled instructions.

The design of an enable memory device for 128 instructions has been translated into

a three micron CMOS layout and was fabricated through the MOSIS service of DARPA. Testing of packaged devices has shown they perform the intended functions and can work at a clock rate of 5 MHz [12].

7 Conclusions

The processing element architecture discussed here is the basis for a multiprocessor computer architecture that offers superior performance to conventional architectures for multiprocessor systems. This is because the pipelined execution units of the dataflow machine can be kept fully utilized even when only scalar parallelism is available in the application being run. The stream of initiated instructions may contain interleaved instructions from all parts of the program that have been loaded into the processing element. This possibility is not supported by any other processor architecture known to us.

We have described the processing element as comprising the Pipelined Instruction Processing Unit and the Dataflow Instruction Scheduling Unit. The PIPU may be implemented using fairly conventional design techniques; the DISU, however, requires a special component called the enable memory. A preliminary design of an experimental enable memory has been fabricated in CMOS technology. This device has been fabricated and tested successfully, supporting our belief that practical realizations of an efficient and competitive dataflow computers are now feasible.

A successful compiler for a dataflow computer must perform a global analysis of data dependencies in the source program and identify the array definitions embedded in the program. If the program is written in a functional programming language such as Val ([1,9]), then this analysis is far easier than is the case for a conventional language such as Fortran. Nevertheless, Fortran programs may be successfully analyzed using techniques developed by Kuck, Kennedy and others [21], and rules for good programming in Fortran can be formulated so that all important array definitions can be recognized by a compiler.

8 Acknowledgements

The work presented here is a further refinement of dataflow processing element designs developed in research on dataflow computer architecture conducted at the M.I.T. Laboratory for Computer Science under Professor Jack B. Dennis ([8]). That research program was funded by the National Science Foundation, the Department of Energy Basic Sciences Program, and the National Aeronautics and Space Administration through several research grants. The project was begun with support from the Defense Advanced Research Projects Agency.

Since September 1985, Dr. Dennis has been developing dataflow technology as an independent consultant. Dr. Gao completed his doctoral research under Professor Dennis in the field of analyzing and mapping array definitions for efficient pipelined execution on dataflow computers. Dr. Gao is continuing his work on compiling techniques for dataflow computation as a member of the faculty at McGill University.

References

[1] Ackerman, W.B. and Dennis, J.B., *Val – A Value-Oriented Algorithmic Language: Preliminary Manual* , Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

[2] Clark, D. W. and Levy, H. M., *Measurement and Analysis of Instruction Use in the Vax-11/780*, The 9-th Annual Symposium on Computer Architecture, 10(3), pp. 9-17, April 1982.

[3] Dennis, J. B., *Data Flow Supercomputers*, Computers Vol 13, No. 11, pp. 48-56 Nov. 1980.

[4] Dennis, J.B. and Gao, G.R., *Maximum Pipelining of Array Operations on Static Data Flow Machine*, Proceeding of the 1983 International Conference on Parallel Processing, Aug 23-26, 1983.

[5] Dennis, J. B., Gao, G. R. and Todd, K., *Modeling the Weather with a Data Flow Supercomputer*, IEEE Transactions on Computers, Vol. 33(7), pp 592-603, July, 1984.

[6] Dennis, J.B., *Data Flow Model of Computation*, In Control Flow and Data Flow: Concepts of Distributed Programming, Broy, M., Ed. Springer-Verlag, Berlin, Heidelberg, New York.

[7] Dennis, J.B., *Data Flow Computation–A Case Study*, In Computer Architecture: Concepts and Systems. V. Milutinovic, Ed., Elsevier, New York. (To be published)

[8] Dennis, J.B., *Data Flow Computer Architecture*, Technical Report, Laboratory for Computer Science, MIT, TR-385 Oct., 1987.

[9] McGraw, J., *The Val Language: Description and Analysis*, ACM Transactions on Programming Language and Systems, Vol. 4, No.1, Jan, 1982.

[10] Gajski, D. D., Padua, D. A, Kuck, D. J. adn Kunh, R. H., *A Second Opinion on Data Flow Machines and Languages*, IEEE Computer, Feb. 1982.

[11] Gajski, D. D. and Peir, J. -K. *Essential Issues in Multiprocessor Systems*, IEEE Computer, June, 1985.

[12] Gao, G. R. and Theobald, K., *An Enable Memory Controller Chip for A Static Data Flow Supercomputer*, Computational Structure Group Design Note 18, Laboratory for Computer Science, MIT, Jan. 1985.

[13] Gao, G.R., *A Pipelined Code Mapping Scheme for Static Data Flow Computers*, Ph.D dissertation, Laboratory for Computer Science, Aug. 1986.

[14] Gao, G. R., *Algorithmic Aspects of Balancing Techniques for Pipelined Data Flow Code Generation*, accepted for publication, on Journal of Parallel and Distributed Computing, Academic Press, Aug. 1987.

[15] Gurd, J. R., Kirkham, C. C. and Watson, I., *The Manchester Prototype Dataflow Computer*, CACM, Vol. 28, No. 1, Jan., 1985.

[16] Hennessy, J. L. et al, *Hardware/software Tradeoffs for Increased Performance*, in Proceeding ACM SIGARCH/SIGPLAN Symp. Architectural Support for Programming Language and Operating Systems, Palo Alto, CA, March 1982, pp. 2-11.

[17] Hennessy,J. L., *VLSI Processor Architecture*, IEEE Transaction on Computers, Vol. c-33, No. 12, Dec. 1984

[18] Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill, 1981.

[19] Lee, J. F. K. and Smith, A., *Branch Prediction Strategies and Branch Target Buffer Design*, Computer 17(1), Jan. 1984.

[20] Morris, D. and Ibbet, R. N., *The MU5 Computer System*, Springer-Verlag, 1979.

[21] Padua, A. D., Wolfe, M. J., *Advanced Compiler Optimizations for Supercomputers*, CACM, Vol. 29, No. 12, Dec. 1986.

[22] Patterson, D. A., *Reduced Instruction Set Computers*, CACM, Vol. 28, No. 1, Jan., 1985.

[23] Reed, D. A., Adams, L. M. and Patrick, M. L., *Stencils and Problem partitionings: Their Influence on the Performance of Multiple Processor Systems*, IEEE Transaction on Computers, C-36, No. 7, June, 1985.

[24] Tood, K. W., *High Level Val Constructs in Static Data Flow Machine*, Technical Report 262, Laboratory for Computer Science, MIT, June, 1981.

[25] Tomasulo, R. M., *An Efficient ALgorithm for Exploiting Multiple Arithmatic Units*, IBM J. Research and Developments, Vol. 11, 1967, pp. 25-33.

[26] Wah, B. W., Li., G. and Yu, C. F., *Multiprocessing of Combinatorial Search Problems*, IEEE Computer, 18, No. 6, June, 1985.

[27] Wiecek, C. A., *A Case Study of Vax-11 Instruction Set Usage for Compiler Execution*, Proceedings of the Symposium on Architectural Support for Programming Languages and Systems, pp. 177-184, March 1982.