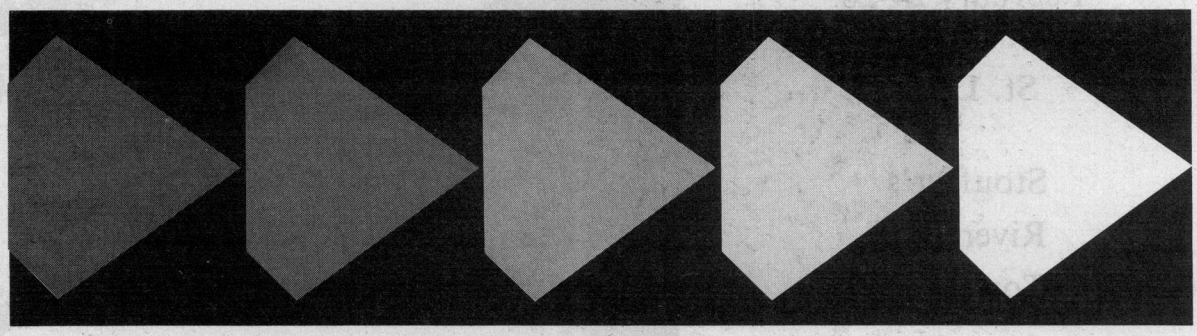
Programmability with increased performance? New strategies to attain this goal include two approaches to data flow architecture: data flow multiprocessors and the cell block architecture.

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 © 1980 IEEE

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.

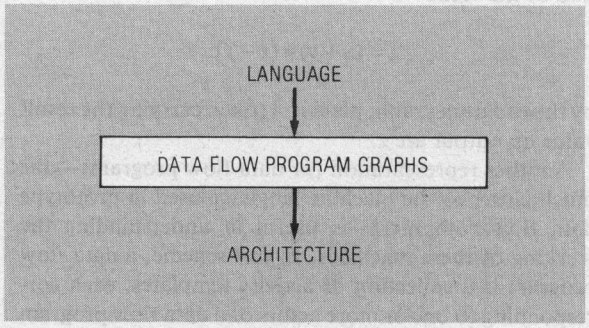


Figure 1. Program graphs as a base language.

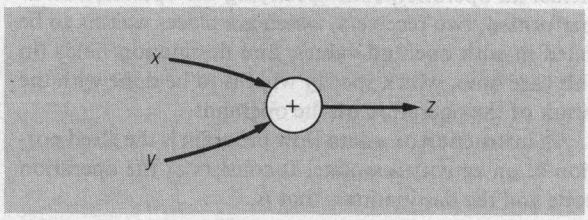


Figure 2. Data flow actor.

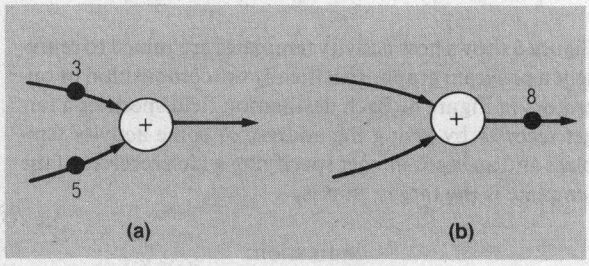


Figure 3. Firing rule: (a) before; (b) after.

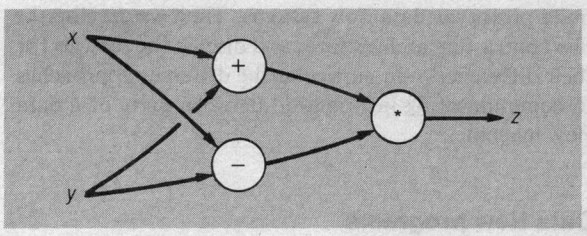


Figure 4. Interconnection of operators.

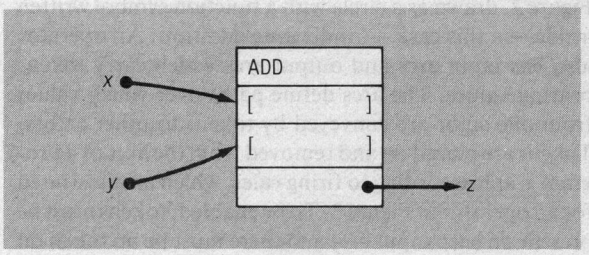


Figure 5. An activity template.

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>

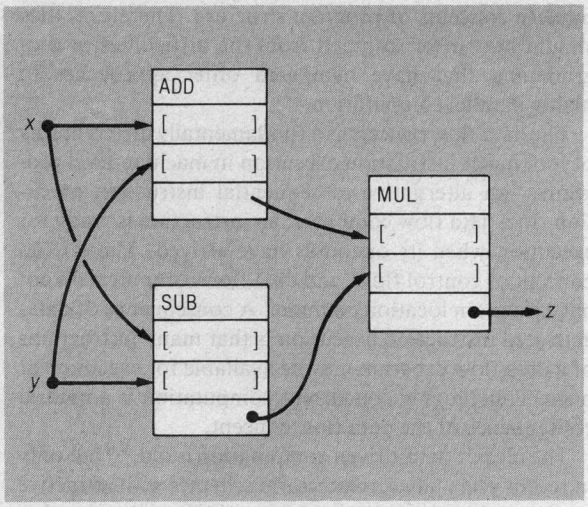


Figure 6. Configuration of activity templates for the program graph of Figure 4.

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

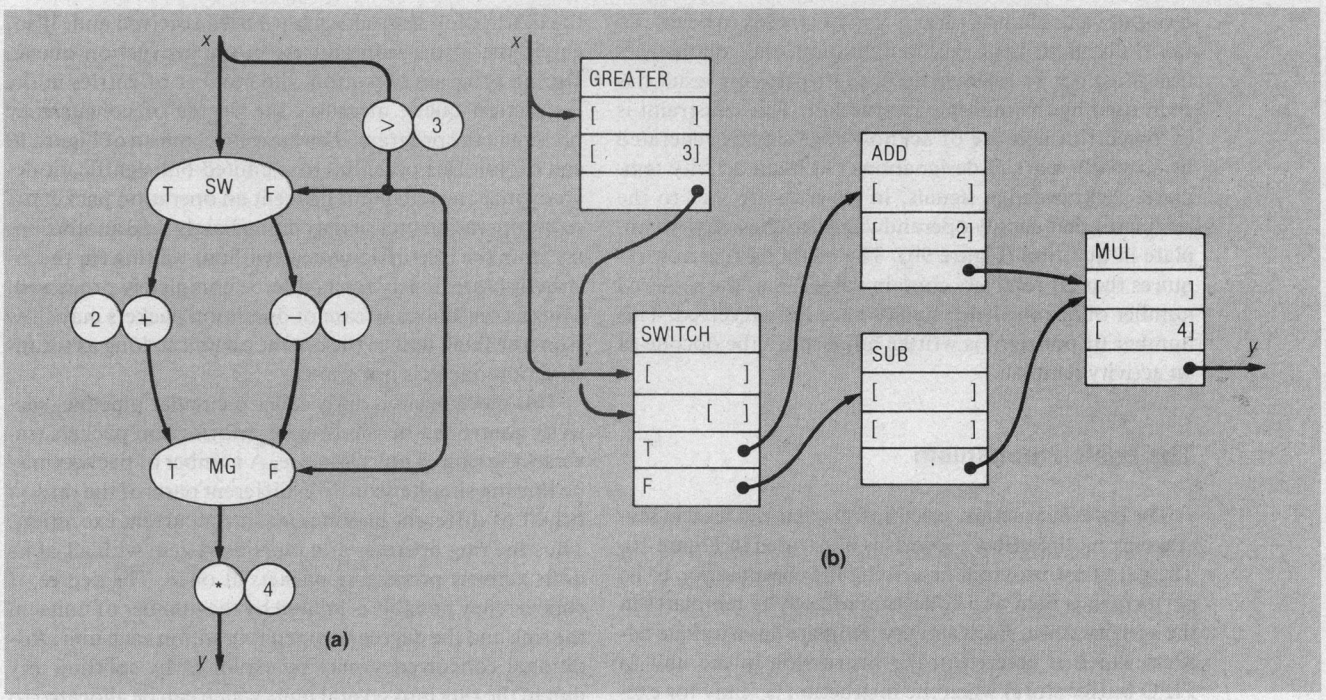


Figure 7. A conditional schema (a) and its implementation (b).

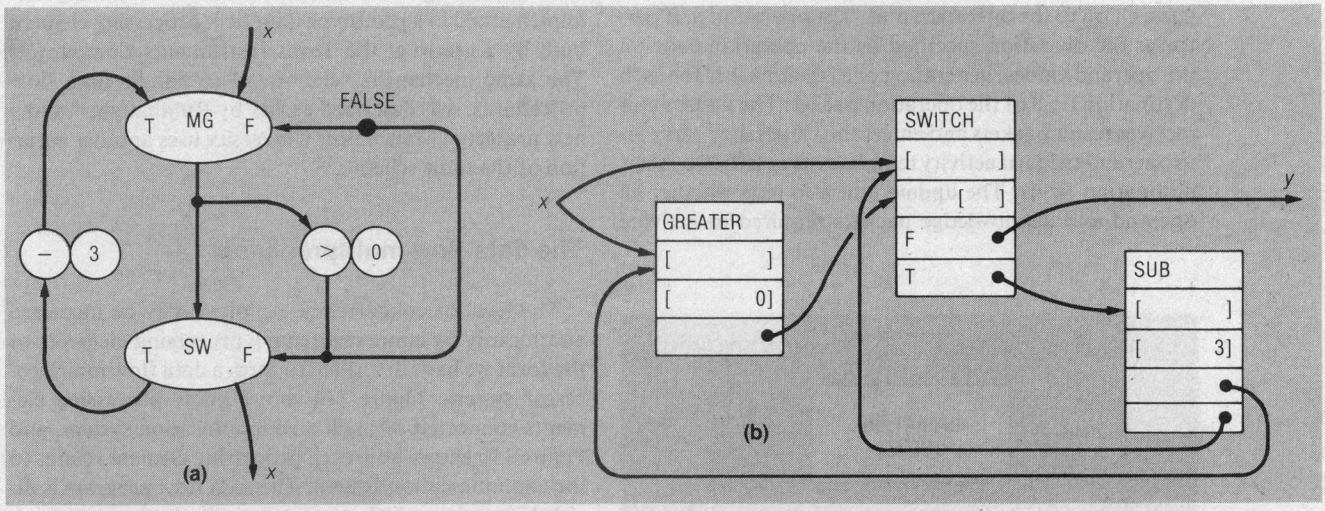


Figure 8. An iterative schema (a) and its implementation (b).

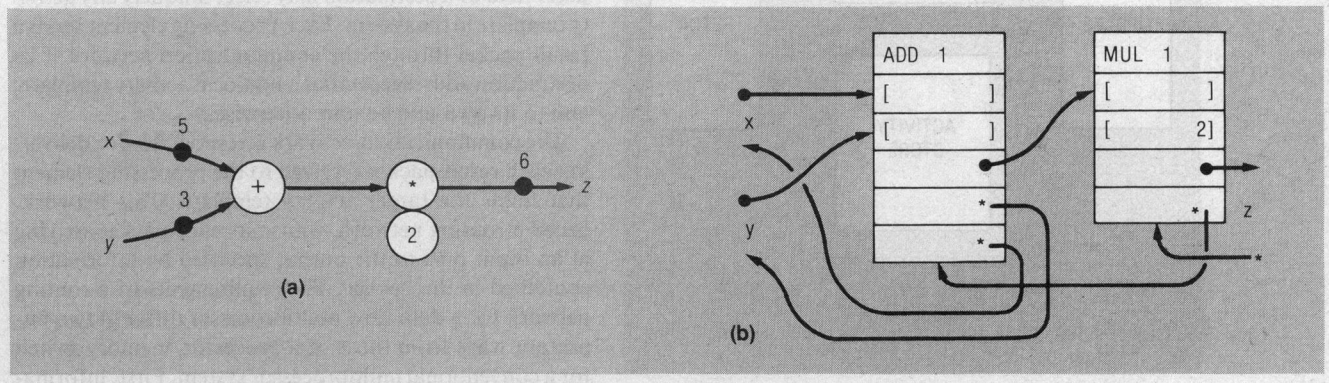


Figure 9. Pipelining in a data flow program (a) and its implementation (b).

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.

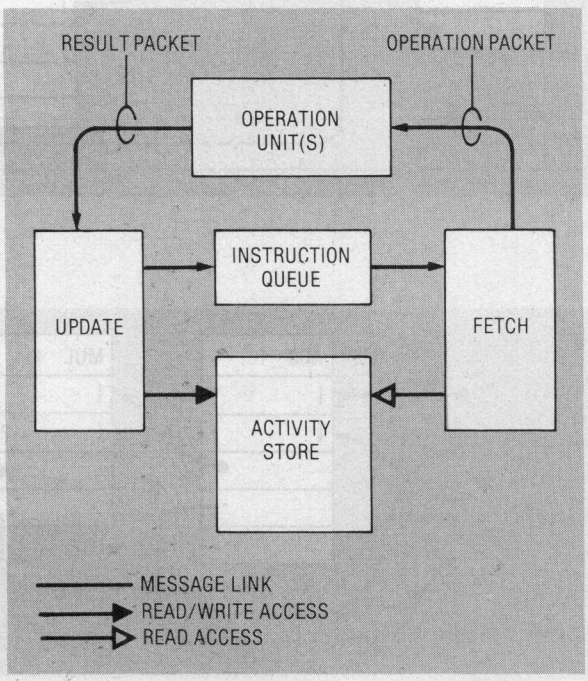


Figure 10. Basic instruction execution mechanism.

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]

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]

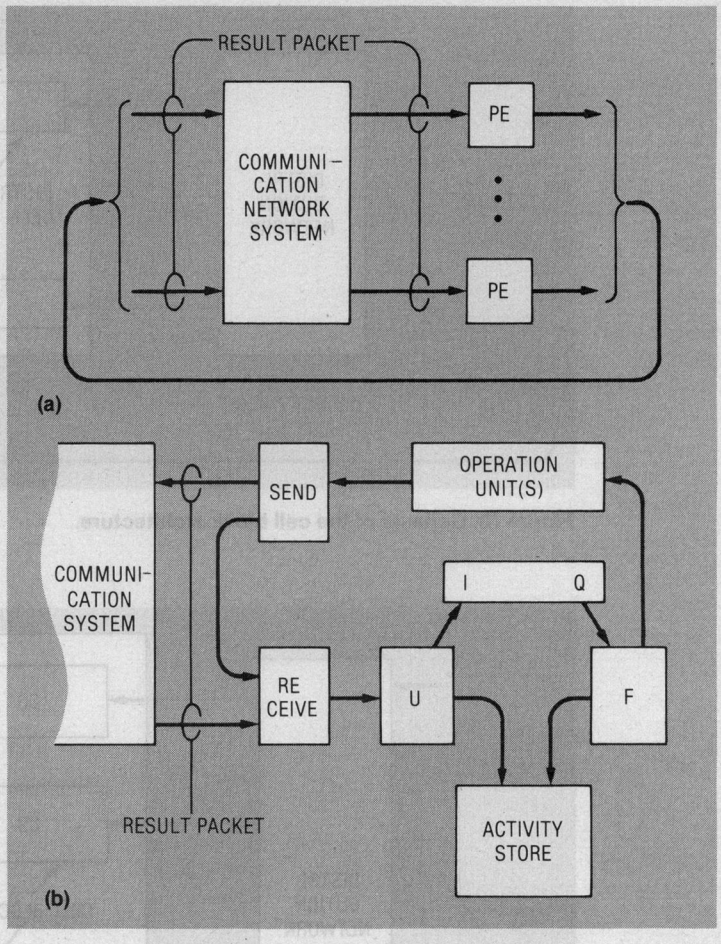


Figure 11. Data flow multiprocessor: (a) connection of many processing elements through a communication system; (b) relationship of each PE to the communication system.

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

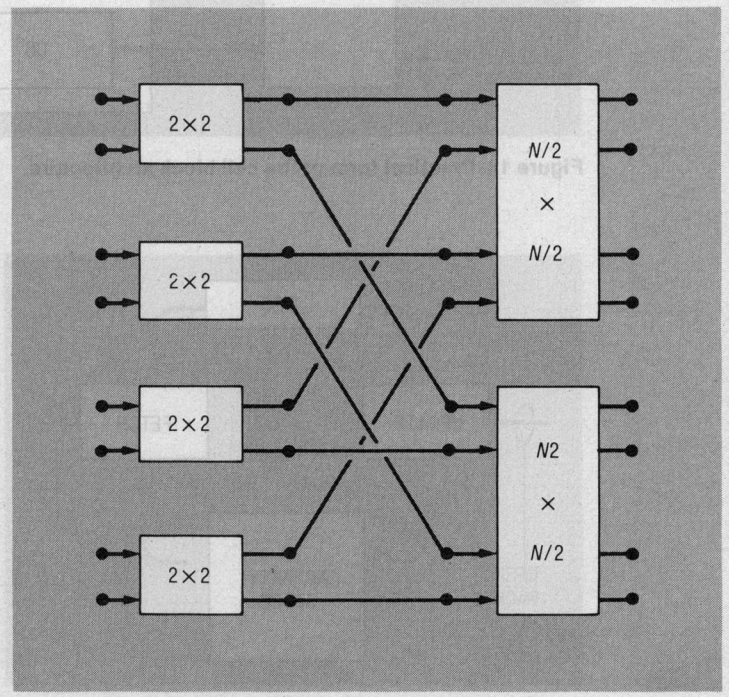


Figure 12. Routing network structure.

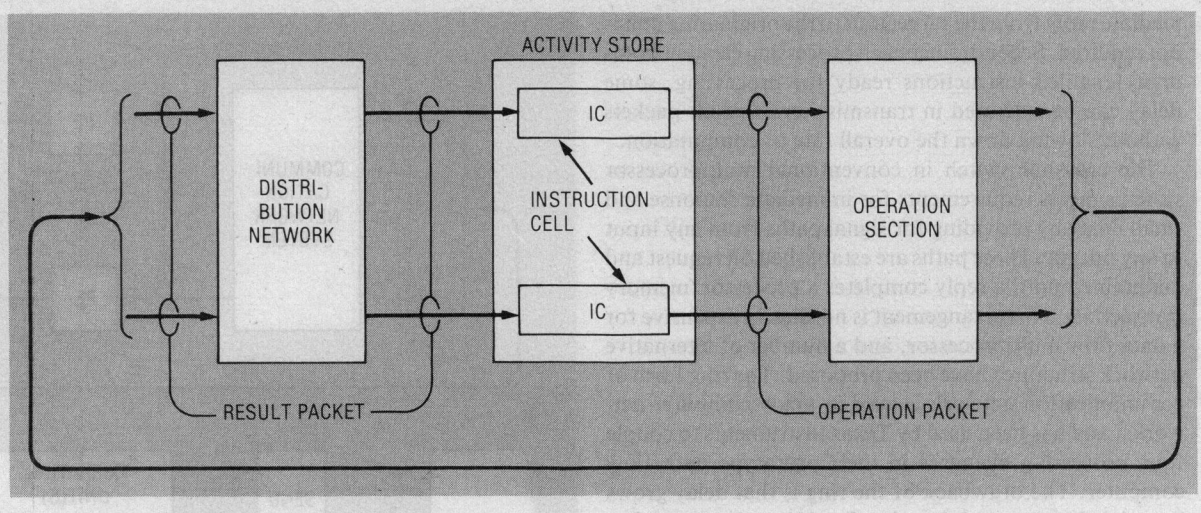


Figure 13. Genesis of the cell block architecture.

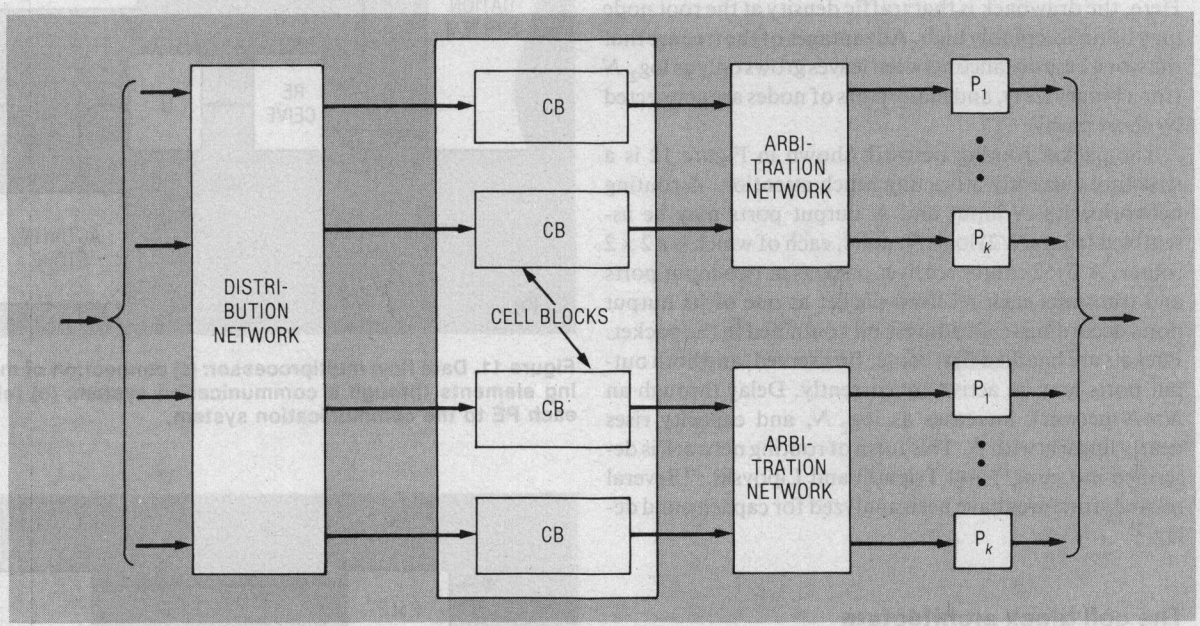


Figure 14. Practical form of the cell block architecture.

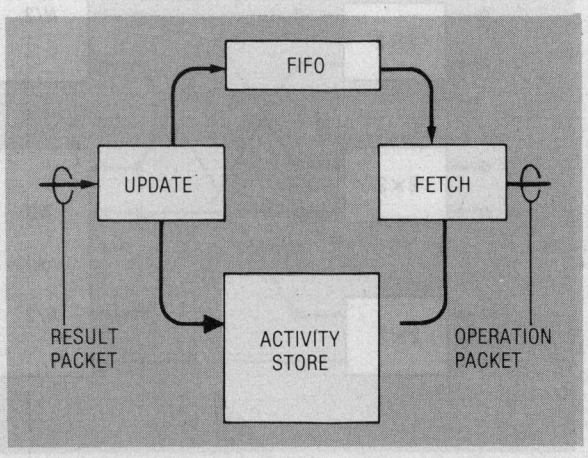


Figure 15. Cell block implementation.

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-

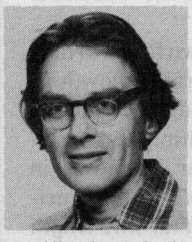
ping high-level programs into machine-level programs that effectively utilize machine resources, but much remains to be done. Creative research is needed to handle data structures in a manner consistent with principles of data flow computation. These are among the problems under study in our data flow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