The Varieties of Data Flow Computers

Computation Structures Group Memo 183-1
August 1979
Revised December 1979

Jack B. Dennis

This paper appeared in the Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, AL, October 1-4, 1979. It was superseded by "Data Flow Supercomputers." *Computer* 13,11, November 1980.

THE VARIETIES OF DATA FLOW COMPUTERS[1]

Jack B. Dennis
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- Architectures of computer systems based on data flow concepts are attracting increasing attention as an alternative to conventional sequential processors. This paper discusses and contrasts several approaches to data flow computation representative of current work on experimental prototype machines.

Introduction

The architects of future computer systems face three challenges. An architectural concept that successfully addresses these challenges will prove a major breakthrough toward computer systems that have high performance and contribute to easing the software problem.

1. Achieve high performance at minimal hardware cost.

This has always been an objective of computer architecture. Of course, the nature of the architecture required changes as one traverses the range of scale from microprocessing to super-computer, and as applications and technology evolve.

2. Utilize effectively the capabilities of LSI technology.

Using LSI devices effectively in medium to large scale computers is a generally recognized problem without generally accepted solutions. Architectures are needed which use large numbers each of a few part types which have a high logic-to-pin ratio. The most popular suggestion having these characteristics is a large number of interconnected microcomputers; however sufficiently good schemes for interconnecting and programming them have not been forthcoming.

3. Programmability

Any radical departure from conventional architectures based on sequential program execution must address the problem that the existing body of software methodology and tools may not be applicable. The architects of supercomputers and multiprocessor systems have not addressed this challenge, trusting that the "software problem" can be successfully attacked by the "software people." This is fallacious.

1. This research was supported by the Lawrence Livermore Laboratory of the University of California under contract 8545403.

A good way to ensure that a radical architecture is programmable is to make the computer system a *language-based* design. This means the system is designed as a hardware interpreter for a specific *base language* in terms of which programs to be run on the system must be expressed [10]. However, much of the work on language-based architecture has not been fruitful because the languages chosen (Fortran and Algol, for example) embody some of the principal limitations of conventional machines (global memory), and lack generality (no provision for expressing concurrency).

Computer designs based on principles of data flow are attracting increasing interest as an alternative to architectures derived from conventional notions of sequential program execution. These new designs offer a possible solution to the problem of efficiently exploiting concurrency of computation on a large scale, and they are compatible with modern concepts of program structure and therefore should not suffer so much from the difficulties of programming that plague other approaches to highly parallel computation: array and vector processors, and shared-memory multiprocessor systems.

Fundamentally, the data flow concept is a different way of looking at instruction execution in machine level programs -- an alternative to the Von Neumann idea of 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 accompaniment of the data flow idea.

The idea of data driven computation is old [21, 22], but it is only in recent years that architectural schemes have been developed that can support an interestingly general level of user language, and are attractive in terms of anticipated performance and practicality of construction. 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 United States and Europe. Several processors using 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* (Dennis [11], which evolved from work of Rodriguez [19], Adams [3] and Karp and Miller [16]. In fact, data flow computers are a form of language-based architecture in which program graphs are the base language. As shown in Fig. 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

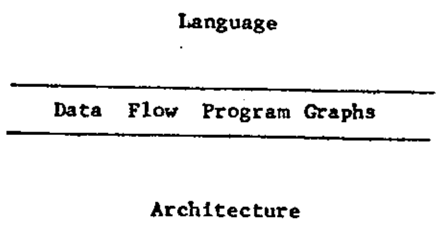


Fig. 1. Program graphs as a base language.

computer system that faithfully implements the formal behavior of program graphs, while 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 [7] 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 to input arguments of a procedure. Removal of these sources of difficulty not only makes concurrency easy to identify, but programs have better structure -- they are more modular, and are easier to understand and verify.

These implications of data flow for language designers are discussed by Ackerman [1]. Moreover, new programming languages have been designed specifically for data flow computations: ID developed at Irvine [4] and VAL designed at MIT [2, 18].

This paper presents a sample from the variety of architectural schemes devised to support computations expressed as data flow program graphs. We explain data flow graphs by means of examples, and show how they are represented as collections of *activity templates*. Then we describe the basic instruction handling mechanism using activity templates that is characteristic of most current projects to build prototype data flow systems. We discuss the reasons for the different hardware organizations used by various projects, in particular, the different approaches to communicating information between parts of a data flow computer.

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 Fig. 2 which is 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 Fig. 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 associated with those tokens, and placing tokens labeled with the result value on the output arcs.

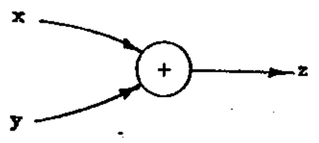


Fig. 2. Data flow actor.

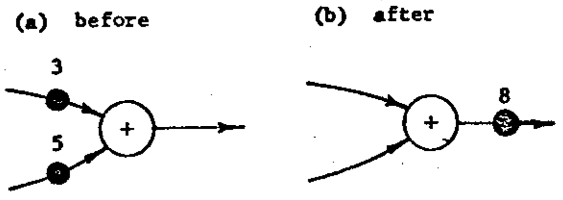


Fig. 3. Firing rule.

Operators may be connected as shown in Fig. 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.

To understand the working of data flow computers, it is useful to introduce another representation for data flow programs -- one that is much closer to the machine language used in prototype data flow computers. 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 (Fig. 2) is shown in Fig. 5. There are four fields for (1) an operation code specifying the operations to be performed; (2) two *receivers*, which are places waiting to be filled in with operand values; and (3) destination fields (in this case one), which specify what is to be done with the result of performing the operation on the operands.

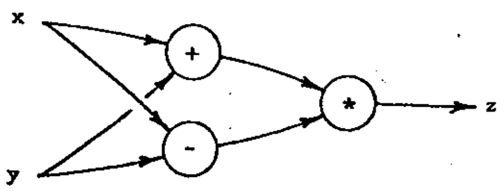


Fig. 4. Interconnection of operators.

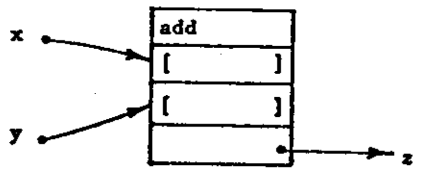


Fig. 5. An activity template.

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

instruction:
<opcode, destinations>

Fig. 6 shows how activity templates are joined to represent a program graph, specifically the composition of operators in Fig. 4. Each destination field specifies a target receiver by giving the

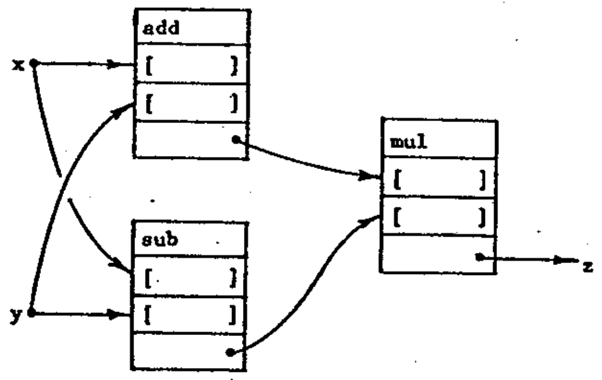


Fig. 6. Configuration of activity templates for the program graph of Fig. 4.

address of some activity template and an *input* integer specifying which receiver of the template is the target.

destination:
<address, input>

Program structures for conditionals and iteration are illustrated in Fig. 7 and Fig. 8. These make use of 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 according as a boolean control input is **true** or **false**. 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 Fig. 7 represent computation of

y := (**if** x > 3 **then** x + 2 **else** x - 1) * 4

and the program graph and implementation in Fig. 8 represent the iterative computation

while x > 0 **do** x := x - 3

Execution of a machine program consisting of activity templates is viewed as follows: When a template is activated by the presence of an operand value in each receiver, the contents of the template from an *operation packet* of the form

operation packet:
<opcode, operands, destinations>

Such a packet specifies one *result packet* having 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.

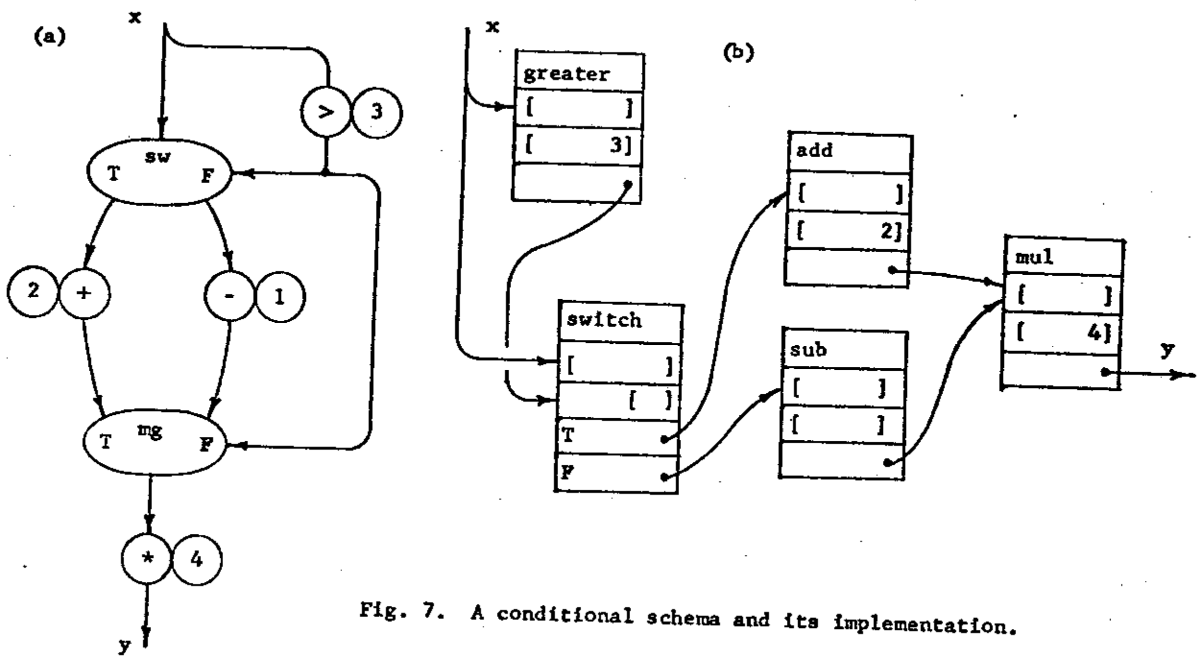


Fig. 7. A conditional schema and its implementation.

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 Fig. 9a. To faithfully represent 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* which are generated by specially marked designations (*) in an activity

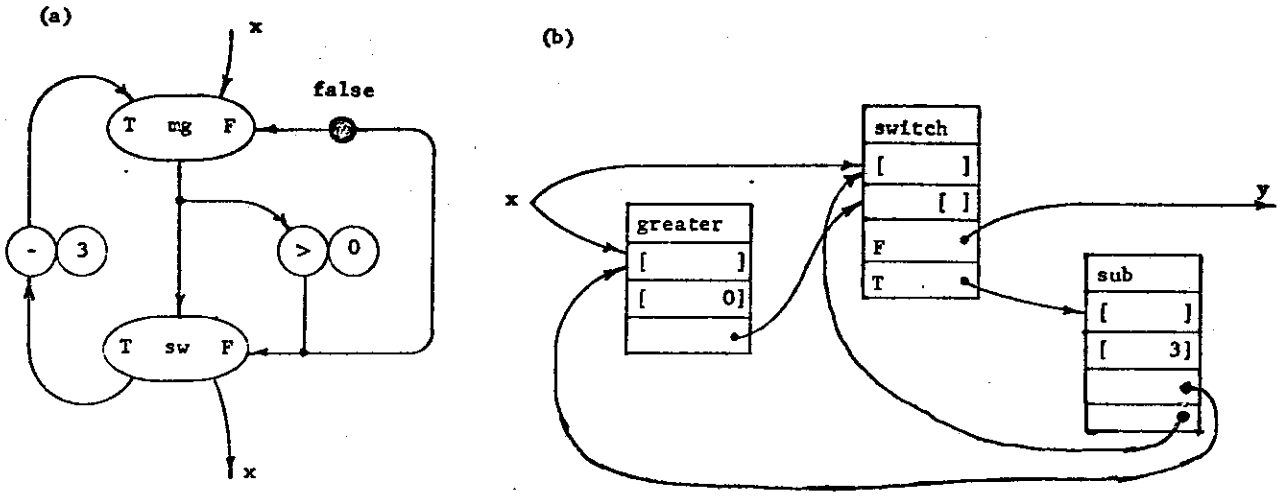


Fig. 8. An iterative schema and its implementation.

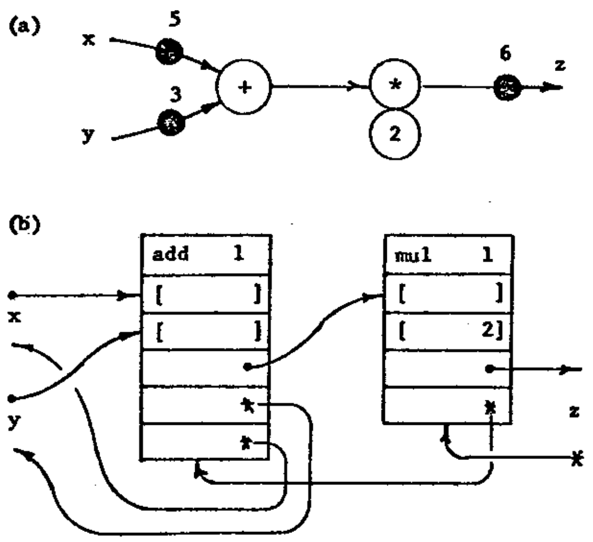


Fig. 9. Pipelining in data flow programs.

template, and in general are sent to the templates that supply operand values to the activity template in question (Fig. 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 a number of current data flow projects is illustrated in Fig. 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.

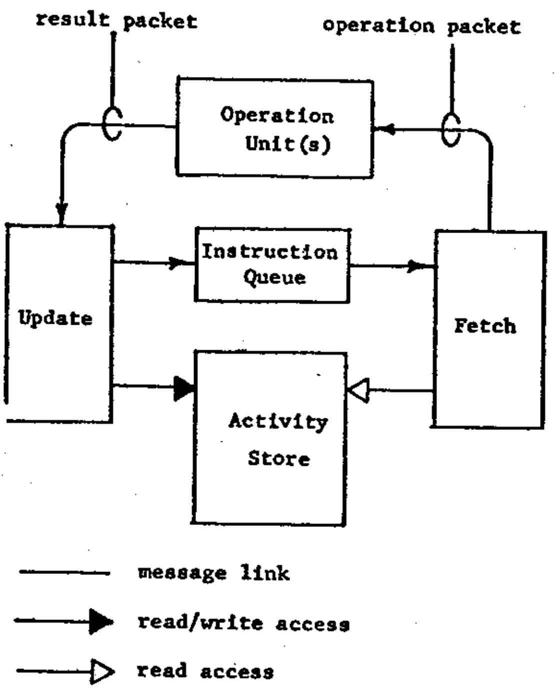


Fig. 10. Basic instruction execution mechanism.

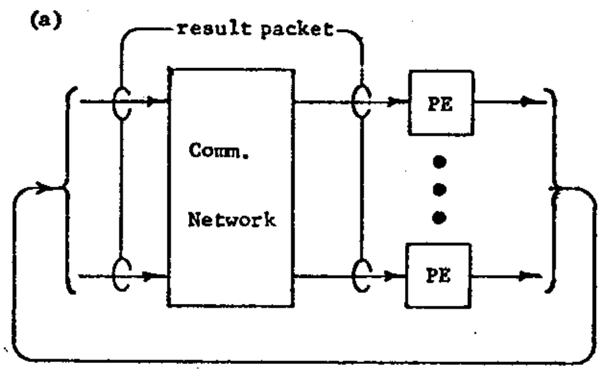
This mechanism is aptly called a "circular pipeline" -- activity controlled by the flow of information packets traverses the ring of units leftwise. A number of packets may be flowing simultaneously in different parts of the ring on behalf of different instructions in concurrent execution. Thus the ring 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 [8]. The same mechanism, elaborated to handle data flow procedures, was described earlier by Rumbaugh [20], and a new project at Manchester University (see below) uses another variation of the same scheme.

Data Flow Multiprocessor

The level of concurrency exploited may be increased enormously by connecting together 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 Fig. 10b 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. Such a network, in which each packet arriving at an input port is transmitted to the output specified by information contained in the packet, is called a *routing network*.



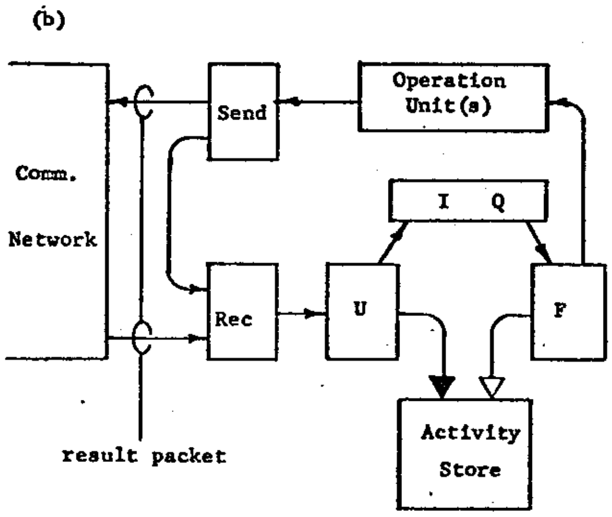


Fig. 11. Data flow multiprocessor.

The characteristics required of a routing network for a data flow multiprocessor differ in two important ways from the properties demanded of a processor/memory switch for a conventional multiprocessor system. First, information flow in a routing network is in one direction -- an immediate 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" used in conventional multiprocessor systems meets requirements for immediate response and small delay by providing for signal paths from any input to any output that 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 has been used in many computer networks and has been used by Texas Instruments to couple four processing elements in their prototype data flow computer. The ring has the drawback 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 [9, 15, 17]. Here, the drawback is that the 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 that many pairs of nodes are connected by short paths.

The packet routing network shown in Fig. 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(N) units each of which is a 2 x 2 router. A 2 x 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 x N network increases as $\log_2 N$ and capacity rises nearly linearly with N. This form of routing network is described in [23], and several related structures have been analyzed for capacity and delay [6].

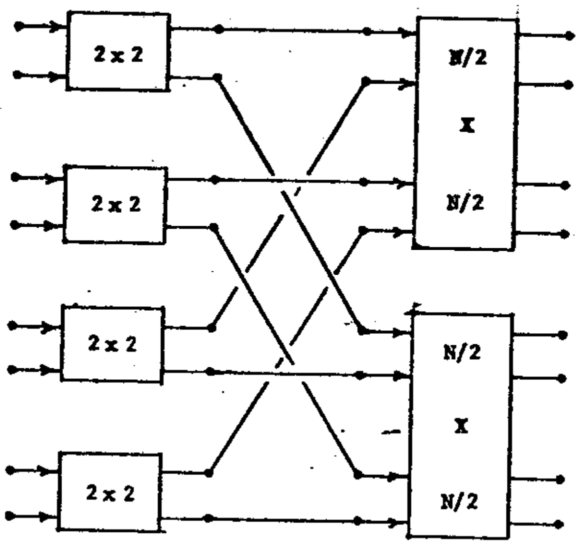
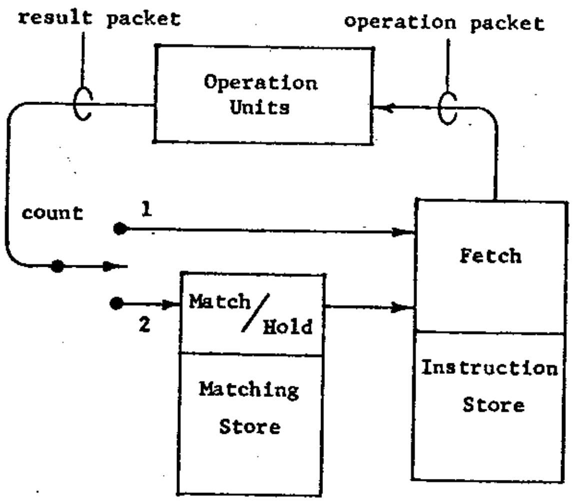


Fig. 12. Routing network structure.

Token Labeling

An experimental data flow computer being constructed at Manchester University, England [24], uses an elaboration of the basic mechanism designed so more than one instance of an instruction may be active at a time. This feature provides for overlapped execution of successive cycles of an iteration, and makes possible a natural machine level implementation of procedure application.

The Manchester processing element design is sketched in Fig. 13. In place of the Activity Store there is an *Instruction Store* and a *Matching Store*. Since more than one instance of execution of an instruction is allowed, the result packet format is extended to include a *label* field used to distinguish instances of the target instruction. No longer can arrived operand values be held in a single activity template for an instruction. Rather, instructions are divided into just two classes -- those that require only one operand, and those that require two operands -- and result packets include an indicator *count* of how many operands the target instruction requires. For single operand instructions, the one result packet is sent directly to the Instruction Store, where the instruction is fetched and an operation packet constructed. For two-operand instructions, the first result packet to arrive at the Matching Store is held until the second result packet arrives. Then information from the two result packets is combined and sent on to the Instruction Store where an operation packet is constructed. The matching store is an associative memory that uses the address and label fields of a result packet as its search key.



operation packet:

<opcode, operands, destination>

result packet:

<value, label, destination>

destination:

<address, input, count>

Fig. 13. Data flow processor with labels.

The MIT Architecture

In a data flow multiprocessor (Fig. 11) we noted the problem of partitioning the instructions of a program among the processing elements so as 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 [12, 13] in response to an opposing view: Each instruction is equally accessible to result packets generated by any other instruction, independent of where they reside in the machine. The structure of this machine is shown in Fig. 14. 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 Operation Section is to execute instructions and to forward result packets to target instructions by way of the Distribution Network.

As drawn in Fig. 14, this design 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 together with the associated portion of the Distribution Network.

Moreover, to further reduce the number of interconnections between Cell Blocks and other units, a byte-serial format for result and operation packets is chosen.

The resulting structure is shown in Fig. 15. Here, several Cell Blocks are served by a shared group of functional units $P_1, \ldots, P_k$. The Arbitration Network in each section of the machine passes each operation packet to the appropriate functional unit according to its opcode.

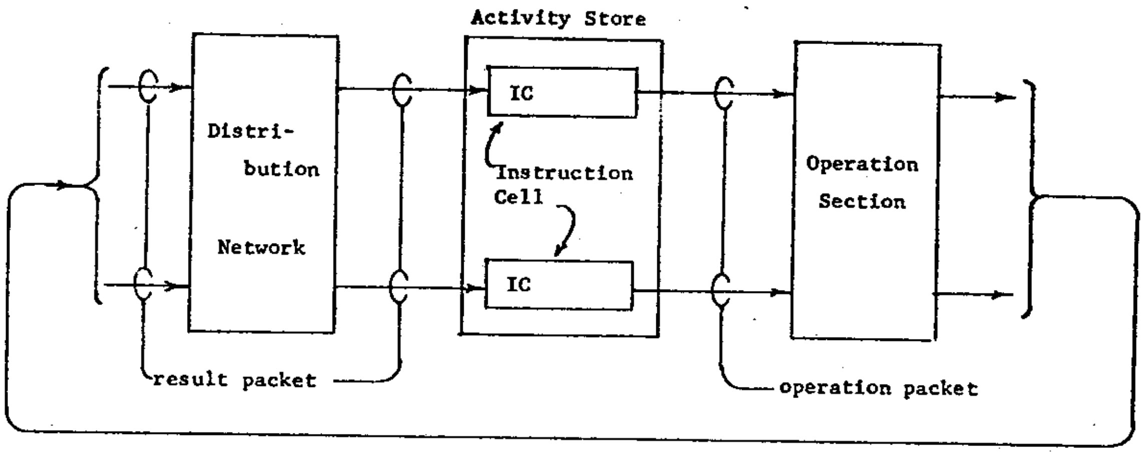


Fig. 14. MIT data flow processor.

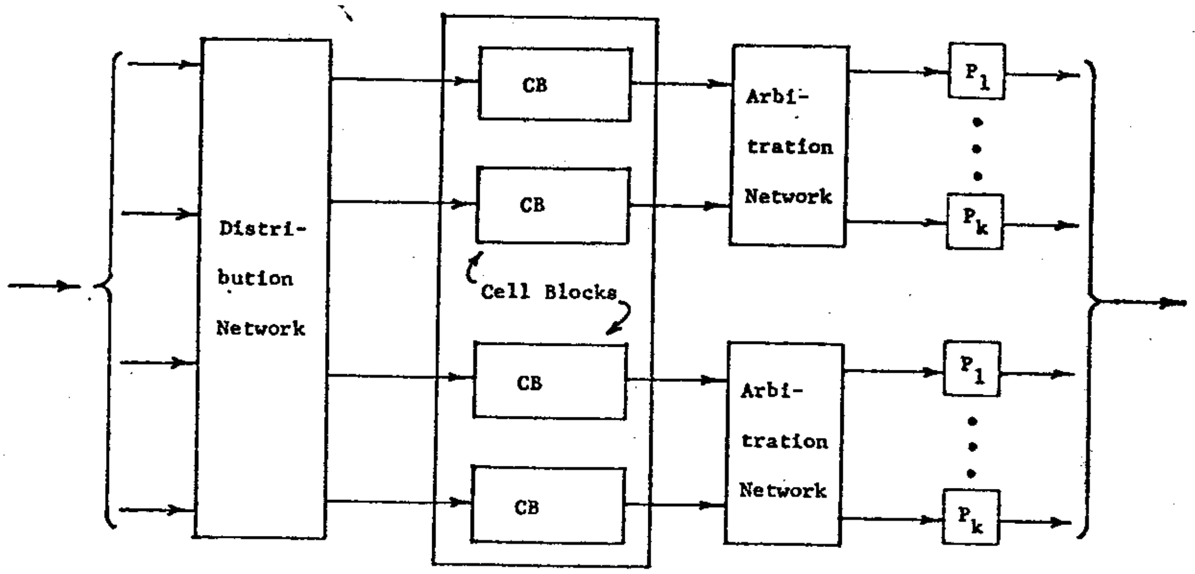


Fig. 15. Practical form of the MIT architecture.

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 MIT architecture and the basic mechanism described earlier becomes clear when one considers how a Cell Block unit would be constructed. As shown in Fig. 16 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 that 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.

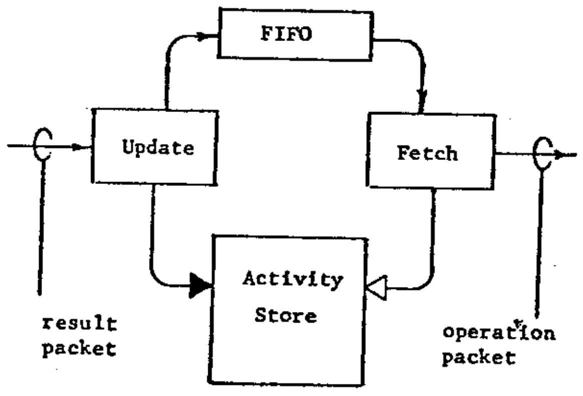


Fig. 16. Cell Block implementation.

Discussion

In the Cell Block machine, 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 so as to minimize communication traffic between clusters, and each cluster is allocated to one processing element. Since it may be handling significantly less packet traffic, the communication network of the data flow multiprocessor will be simpler and less expensive than the Distribution Network of the MIT machine. Whether the cost reduction justifies the additional programming effort is a matter of debate, and depends 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, closer examination reveals that in all suggested physical structures for N x 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 multi-unit systems which support universal intercommunication. If we need yet larger systems, it appears we must settle for arrangements of units that only support immediate communication with neighbors. It is not at all clear how such a system could support a general approach to program construction. A variety of views are currently held as to the circumstances which would favor construction of machines having only local interconnections. A view implicit in most proposals for distributed computing systems is that the programmer (or, alternatively, a *very* smart compiler) will plan how the computation should be distributed so as to optimize resource utilization. A corollary of this view is that programming such systems will be at least as difficult as programming a conventional single processor system; that is, this form of distributed architecture makes no contribution to ameliorating the software problem.

Another view is that the system itself should dynamically allocate its resources among portions of the computation to be performed so that in each interval of computation, only local interactions are required. This view is consistent with current advanced thinking about programming languages and methodology. For this to be possible, very flexible mechanisms must be built into the hardware to support dynamic reallocation of processing and memory resources without imposition on the programmer. Systems proposed from this viewpoint include the Irvine data flow architecture [5], the Utah project toward a demand driven implementation of applicative Lisp [17], and an operational concept of data flow program execution developed by Weng [14, 25]. Whether these proposals can be developed into practical computer systems is an open question.

Extensions

The forms of data flow architecture discussed in this paper are limited in several significant ways. There is no specific mechanism in these systems to provide efficient support for data structures, and only the Manchester University machine incorporates even rudimentary support for multiple instances of instruction execution such as required for implementing concurrent or recursive procedure activations. Moreover, in each of these systems, all instructions are held in the same level of storage and there is no provision for "caching" instructions, so programs beyond some limiting size become impractical due to their need to occupy relatively fast storage.

A variety of proposals have been made of approaches to overcome these limitations of current prototype construction projects, but none have yet reached the stage that even experimental construction of a machine is warranted. It will be fascinating to see how these concepts evolve over the coming decade.

References

[1] W. B. Ackerman, "Data Flow Languages," Proc. of the ACM 1979 National Computer Conference (June 1979), pp. 1087-1095.

[2] W. B. Ackerman and J. B. Dennis, VAL -- A Value Oriented Algorithmic Language, Preliminary Reference Manual, Laboratory for Computer Science, M.I.T., Technical Report TR-218 (June 1979), 80 pp.

[3] D. A. Adams, A Computation Model With Data Flow Sequencing, Computer Science Dept., School of Humanities and Sciences, Stanford University, Technical Report CS 117 (December 1968).

[4] 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 (December 1978), 97 pp.

[5] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77, North Holland (1977), pp. 849-854.

[6] G. A. Boughton, Routing Networks in Packet Communication Architectures, Dept. of Electrical Engineering and Computer Science, M.I.T., S.M. Thesis (June 1978).

[7] J. D. Brock and L. B. Montz, "Translation and Optimization of Data Flow Programs," Proceedings of the 1979 International Conference on Parallel Processing (August 1979).

[8] M. Cornish, Private communication, Texas Instruments Corp., Austin, Texas.

[9] A. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality," Proc. of the ACM 1979 National Computer Conference (June 1979), pp. 1079-1086.

[10] J. B. Dennis, "On the Design and Specification of a Common Base Language," Proc. of Symposium on Computers and Automata, Polytechnic Press, Polytechnic Institute of Brooklyn (1971).

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

[12] J. B. Dennis, and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor, Laboratory for Computer Science, M.I.T., CSG Memo 102 (August 1974), 27 pp.

[13] 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, M.I.T., CSG Memo 134-1 (June 1979), 33 pp.

[14] J. B. Dennis, and K. Weng, "An Abstract Implementation for Concurrent Computation With Streams," Proceedings of the 1979 International Conference on Parallel Processing (August 1979).

[15] A. Despain and D. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," Proceedings of the 5th Annual Symposium on Computer Architecture (April 1978), pp 144-150.

[16] R. M. Karp, and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM J. of Applied Mathematics (November 1966), pp. 1390-1411.

[17] R. M. Keller, G. Lindstrom, and S. S. Patil, "A Loosely-Coupled Applicative Multi-processing System," Proc. of the ACM 1979 National Computer Conference (June 1979), pp. 613-622.

[18] J. R. McGraw, "VAL -- A Data Flow Language," these proceedings.

[19] J. E. Rodriguez, A Graph Model for Parallel Computation, Laboratory for Computer Science, M.I.T., Technical Report TR-64 (September 1969).

[20] J. E. Rumbaugh, "A Data Flow Multiprocessor," IEEE Trans. on Computers (February 1977), pp. 138-146.

[21] R. R. Seeber and A. B. Lindquist, "Associative Logic for Highly Parallel Systems," Proc. of the AFIPS Conference (1963), pp. 489-493.

[22] 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 (December 1971).

[23] A. R. Tripathi and G. J. Lipovski, "Packet Switching in Banyan Networks," Proceedings of the 6th Annual Symposium on Computer Architecture (April 1979), pp. 160-167.

[24] I. Watson and J. Gurd, "A Prototype Data Flow Computer With Token Labelling," Proc. of the ACM 1979 National Computer Conference (June 1979), pp. 623-628.

[25] K. Weng, An Abstract Implementation for a Generalized Data Flow Language, Laboratory for Computer Science, M.I.T., Technical Report, forthcoming.