Software I

Function-level computing

A new programming method, linked to radically different architectures, may greatly simplify software development

Computer hardware has advanced tremendously in the last 25 years, going from vacuum tubes to very large-scale integrated circuits. In the same period neither the architecture of computers nor the programming languages used to control them have changed significantly. As a result, the expanding capabilities of VLSI are not being fully exploited and the costs of much-needed programs are soaring. Recognition of this fact has focused attention on a style of programming called functional programming, which offers the prospect of much cheaper programs and new machine architectures that exploit VLSI. The functional approach rejects the model of computing conceived by the mathematician John von Neumann and others. The von Neumann model is based on a computer consisting of a central processing unit (CPU), a store or memory, and a connection between them that transmits a single unit of data, or "word," between the CPU and the store. Because today's programming languages are modeled on such computers, programs are complex, concerned with the smallest data entities, and seldom reusable in building new programs.

The most unfortunate result has been the enormous cost of software. While computing power and hardware costs get cheaper every year, the writing of software becomes more expensive. The programs to make a given microprocessor useful may cost considerably more to develop than the microprocessor itself. Such costs and the expertise required to write programs discourage millions of people from using computers.

There are two functional programming styles, one—exemplified by the LISP language—was developed over 20 years ago. But while function definition is its central notion, LISP retains some features of von Neumann programming. The second style, called function-level programming, has been developed since the mid-1970s by a number of researchers, including this writer.

In this style, existing programs are put together with so-called program-forming operations to form new programs, which can again be used to build even larger ones. This approach allows parallel operations to be expressed easily; it suggests hardware designs built from large numbers of identical units that can achieve highly parallel operation, designs well suited to VLSI technology.

Function-level computing is still in its infancy; only relatively small resources are being devoted to its development. In part this is because it is more a revolutionary than an evolutionary approach to computing. Many theoretical and practical problems remain to be solved before it can become a reality in the marketplace. Nevertheless interest in function-level computing is steadily growing because it is one of very few approaches that offer a real hope of relieving the twin crises of computing today: the absolute necessity to reduce the cost of programming and the need to find computer designs that make much better use of the power of VLSI and of parallelism.

John Backus IBM Research

The von Neumann bottleneck

The key problem caused by the original design of computers is in the connection between the CPU and the store [see Fig. 1]. Since the huge contents of the store must pass, one word at a time, through this connection to the CPU and back again, one might call this the "von Neumann bottleneck."

This bottleneck blocks parallel operation and the effective use of more VLSI circuits, but, more critically, it is the model for serious drawbacks in programming languages. Programs in present languages alter the data stored in memory one word at a time. Variables in the programs are used to designate the storage cells, and one entire assignment statement is needed to alter the

The software challenge

The growth of computer use in the last few years has been fueled by the wide availability of cheap hardware, but it is becoming more and more clear that software is the limiting factor in putting raw computer power to use. Software packages may now cost more than the machines they run on, and even the several hundred thousand professional programmers in the United States cannot keep up with the demand. Programming, although it is becoming easier, is still too complex and tedious for the average computer user to pick up, and so the ultimate users of computing power—businessmen, accountants, scientists, and engineers—still require a middleman to communicate with their machines.

Some of the attempts at making programming easier and more efficient are aimed at relieving professional programmers of drudgery—by shifting repetitive tasks to the machine—while other attempts are aimed at simplifying and automating program generation so that nonprogrammers can set up complex tasks. These approaches are dealt with in the second and third articles in this special report, "Automating programming" [pp. 28–33] and "Programming for nonprogrammers" [pp. 34–38].

A more radical approach is presented by John Backus of IBM Research in San Jose, Calif., in "Function-level computing" beginning on this page. Mr. Backus, the originator of the Fortran computer language, believes that the problem lies in the basic architecture of current machines and current languages. He proposes that new basic designs for computers and corresponding new programming styles will make far easier the task of building complex programs from simple modules.

One thing is clear from these articles: while software automation is on the way, it is not yet here. Some simple tasks have been automated, but automatic generation of complex programs is still some way off. —*Ed.*

0018-9235/82/0800-0022$00.75©1982 IEEE

data for each variable. Thus programs consist of repetitive sequences of instructions, with control statements governing how many times and under what conditions the sequences of assignment statements are to be repeated.

If programming is to be really simplified, it is absolutely crucial to be able to build high-level programs from existing programs; and one must be able to do this knowing only the purpose of each constituent program without a lot of other details. [To understand how existing programming languages fail in this, see "The stores problem," p. 27].

A shift in focus in programming

Both von Neumann and LISP-style programs are more concerned with "object" building than with program building. For example, "average(x,y) = half(x + y)" defines an "object-level" program that transforms any pair of objects, x and y, into the desired result-object, their average. It uses object-forming operations (half, +) to build the objects x + y and then half (x + y). It says how to build an object, not how to build the program.

Function-level programming seeks to shift the focus and the level of programming from the combining of objects to the combining of programs (programs are now simply mathematical functions or mappings). The goal of this shift from object-level to function-level description of programs is to emphasize the main issue of programming: how programs are put together, rather than how objects are put together.

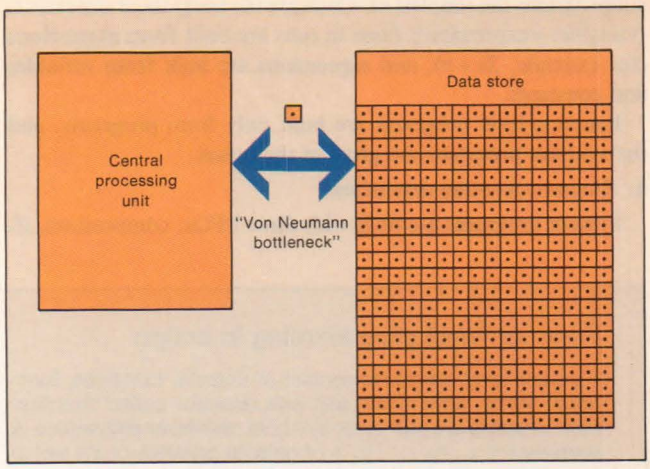
Instead of describing how to form the result-object for a program by applying object-forming operations to objects, the function-level style constructs the program directly by applying program-forming operations (PFOs) to existing programs. For example, the function-level description of average is "average = half ∘ +". Here average is built from two simpler programs (half, +) with the PFO "composition," denoted by the small circle (∘), which means "do the right operation (+) first, then do the left one (half) to the result." Thus, average applied to a pair of numbers is simply the half of their sum.

In the function-level style half a dozen or more PFOs can be used to construct programs. In each case the meaning of a program built by a PFO is simply related to the meanings of the programs from which it is built. The program P ∘ Q always represents the composition of the purpose of P with that of Q, and P ∘ Q is always meaningful if it makes sense to apply P to the things that Q produces.

It is this ability to build up meaningful programs from either simple or complex ones that is the principal strength of the function-level style. It can describe programs that have no object-level counterparts; these tend to be concise, well structured and nonrepetitive.

Another program-forming operation is "construction," which is denoted by square brackets. The construction of two programs does both operations to its argument and returns a pair of results. Thus the program [half, double] applied to 4 gives < 2,8 >. If one now starts with three given programs P, Q, and R and builds the program P ∘ [Q,R], then its meaning is clear: first do [Q,R] to form a pair of objects, the first the result of applying Q, the second the result of applying R (a pair of objects is also an object). Then do P to that pair. P ∘ [Q,R] will be meaningful if it makes sense to apply P to pairs produced by Q and R.

In addition to the power that PFOs provide for building meaningful programs at all levels of complexity, they also have other important properties. For example, composition and construction satisfy the distributive law $[f, g] \circ h = [f \circ h, g \circ h]$, for all programs f, g, and h. Notice that the program on the right side can be made more efficient by transforming it into the one on the left, since the latter uses h only once. Thus even this simple law can be used to improve many programs. There are dozens of similar laws from which many theorems about programs can be derived; these can represent a large body of knowledge that can be used again and again to prove the correctness of programs and to guide their construction.



[1] The von Neumann machine operates on one word of data at a time, even when the same operation must be performed on thousands of words. Finding the location of a piece of data in memory and bringing it to the CPU therefore limits computational speed.

Some program-forming operations express parallel operations very naturally, whereas von Neumann programs are essentially sequential. When the program [P,Q] is executed, it makes no difference whether P or Q is done first or whether both are done together. This potential for parallelism in function-level programming, if incorporated into new computer architectures, would lead to a better use of VLSI.

To see why such languages may some day cut the cost of programming by a larger factor than Fortran achieved 25 years ago when it replaced machine languages, it may be helpful to contrast function-level programming languages with von Neumann languages in five major areas:

1. *Program domains*

Present programs map stores into stores, whereas their real purpose is to map objects into objects. The structure of data objects in the store is known only to the programs that use it; changing the size or structure of the data means changing the programs.

Function-level programs map objects into objects and thus a program directly represents the transformation that is its purpose. Objects can be numbers or symbols or even sequences of other objects. The structure of a data object is part of the object, rather than part of the program, so the same program can treat objects of different structure and size.

2. *Program building*

In the present approach, unless programs have a common data-storage plan, a composite program built from them is meaningless. In the function-level approach, programs can be freely built from others that have suitable purposes. The purpose of a program is simply related to those of the programs from which it is built.

3. *Program structure*

Present programs contain three kinds of structures: programs, expressions, and variables or constants. A program is built from

subprograms the simplest of which are the assignment statements (variable: = expression); these in turn are built from expressions (for example, $2x + y$), and expressions are built from variables and constants.

Function-level programs are built only from programs, and the simplest programs are given at the outset.

4. *Program-forming operations*

Present programs are built with three PFOs: composition, if-then-else, and while. Function-level programs can be built using six or more PFOs: composition, condition, construction, constant, apply-to-all, insert, and others [see "Function-level programming in action," below, for descriptions].

5. *Algebraic treatment of programs and correctness proofs*

With present programs, PFOs satisfy few algebraic laws. There are few general, practical theorems about programs; most apply only to a single program or a small class of programs (for

Function-level programming in action

Function-level programs consist of objects, functions, functional forms, definitions, and one operator called "application." Objects are numbers, symbols, words, or sequences. A sequence $< x_1, x_2, \ldots, x_n >$ of objects consists of x's which are either numbers, symbols, words, or sequences.

"Application" is the operation of applying a function to an object. For example, to apply the addition function (+) to the object $< 1, 2 >$ we write:

$$+ : < 1, 2 > = 3$$

Primitive functions, like all functions, transform one object into another. Examples are:

1. Selector functions, which choose an element of a sequence:

$$1 : < x_1, x_2, \ldots, x_n > = x_1$$
$$2 : < x_1, x_2, \ldots, x_n > = x_2$$

2. Arithmetic functions, such as +, −, ×, ÷, and so on.

3. Transpose:

$$\text{trans}: << 1, 2 >, < 3, 4 >> = << 1, 3 >, < 2, 4 >>$$

4. Distribution functions, such as distribute from the left:

$$\text{distl}: <x, <y_1, y_2, \ldots, y_n >> = << x, y_1 >, <x, y_2 >, \ldots, <x, y_n >>$$

Functional forms are expressions denoting programs that are built from existing programs using program-forming operations (PFOs). Some examples of PFOs and simple functional forms built with them are:

1. Composition (of f and g):

$$(f \circ g) : x = f : (g : x)$$

In words: the composition of f and g, f ∘ g, applied to x gives the result of applying f to the result of applying g to x. If f = arctan and g = sin, then f ∘ g is the arctangent of the sine.

2. Construction (of $f_1, f_2, \ldots, f_n$):

$$[f_1, f_2, \ldots, f_n] : x = < f_1 : x, f_2 : x, \ldots, f_n : x >$$

3. Condition (of p, f and g):

$$(p \rightarrow f; g) : x = f : x \text{ if } p : x \text{ is true;}$$
$$= g : x \text{ if } p : x \text{ is false}$$

4. Constant (of an object y; constant makes a constant-valued function out of an object):

$$y : x = y \quad \text{for any x, the function y gives the result y}$$

5. Insert (of f):

$$/ f : < x_1, x_2, \ldots, x_n > = f : < x_1, / f : < x_2, \ldots, x_n >>$$

6. Apply-to-all (of f):

$$\alpha f : < x, x_2, \ldots, x_n > = < f : x_1, f : x_2, \ldots, f : x_n >$$

Definitions define new functions in terms of old ones. Thus **Def** f = g ∘ [h, k] means that f is to stand for the function g ∘ [h, k]

The difference between function-level and von Neumann programs can be illustrated.

Vector inner product. The vector inner product is obtained by multiplying pairwise the elements of two vectors and adding these products. For example, in a billing system, a vector of the prices of all items would be multiplied by the vector of orders for each item to give the total bill.

(a) Von Neumann program:

```
c : = 0;
for i : = 1 step 1 until n do
        c : = c + a ( i ) × b ( i )
```

(b) Function-level program:

```
Define   Inner Product = (insert +) ∘ (apply-to-all ×)
                                    ∘ transpose
```

Or in abbreviated form:

Def IP = (/ +) ∘ (α ×) ∘ trans

This program is executed from right to left and can be expressed as follows: "The definition of inner product is: transpose the pair of vectors (pair their elements), multiply each pair together, and sum the resulting vector." A preprocessor or "translator" program would not have much trouble in translating language like the above into the FP definition.

To see this program in action, take the vectors $< 1, 2, 3 >$ and $< 4, 5, 6 >$ as an example and apply IP to this pair:

1. Composition gives:

$$(/ +) : ((\alpha \times) : (\text{trans}: << 1, 2, 3 >, < 4, 5, 6 >>))$$

2. Transpose gives:

$$(/ +) : ((\alpha \times) : << 1, 4 >, < 2, 5 >, < 3, 6 >>)$$

3. Apply-to-all gives:

$$(/ +) : < \times : <1, 4 >, \times : < 2, 5 >, \times : < 3, 6 >>$$

4. Multiply gives:

$$(/ +) : <4, 10, 18>$$

5. Insert gives:

$$+ : < 4, + : <10, 18 >>$$

6. Addition gives:

$$+ : < 4, 28 >$$

7. Another addition gives:

$$32$$

The von Neumann and functional programs for inner product have several differences:

1. The functional program is hierarchically built from three generally useful, preexisting programs (+, ×, trans). All the components (the two assignment statements and the 'for' statement) of the other program must be specially written for it alone.

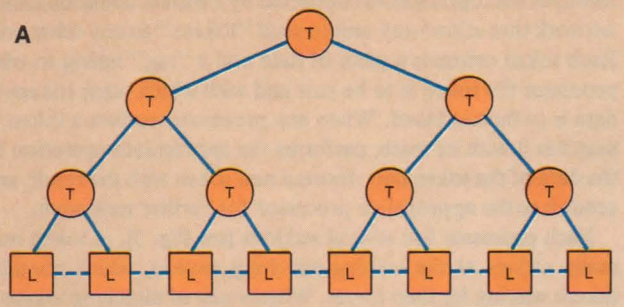
2. The von Neumann program is repetitive—to understand it, one must mentally execute it, or use special mathematical tools. The FP program is nonrepetitive; if its components are understood, its meaning is clear.

3. The von Neumann program computes one word at a time by repetition. The functional program operates on whole conceptual units, not words, and does not repeat any steps.

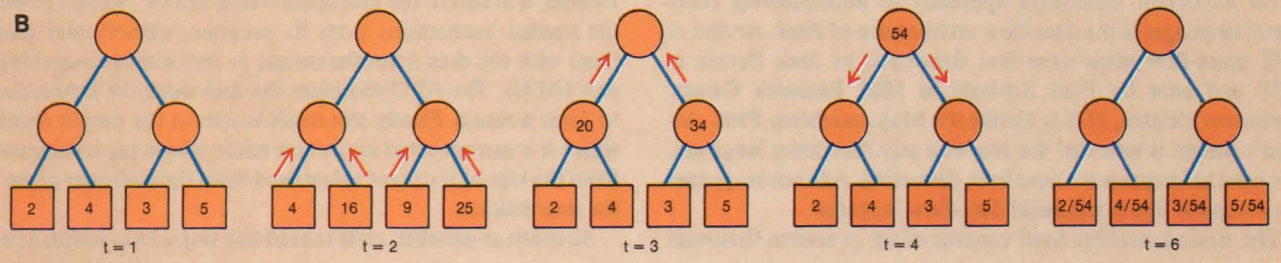
4. The first program mentions the length, n, of the vectors; hence it lacks generality. The functional program is completely general.

5. The von Neumann program names its arguments—it will only work for vectors called a and b. The functional program can be applied to any pair of vectors without naming them.

—J.B.



[2] The cellular computer (A) consists of a binary tree of two kinds of processors: T cells perform simple operations and channel data between L cells, which perform more complex operations. The machine works by sweeping data up through the tree and back down again, with further results of the problem under way being produced with each sweep (B). In this example, the components of a vector are being divided by the sum of their squares; first, the elements are squared in the L cells, then T cells perform a summation up the tree. The sum is then sent down to all L cells, which perform a division.



example, "Theorem: this program is correct").

With function-level programs, PFOs satisfy dozens of algebraic laws that yield many general theorems about program equivalence and about solving equations for programs. Proofs about large classes of programs are possible, and other proofs can be greatly simplified by drawing on standard general theorems, as in mathematics.

This comparison indicates that function-level languages have overcome many of the principle problems in von Neumann languages. However, these languages are very distant from the von Neumann model of computing, and this means that either many problems of optimization must be solved before they can be run on von Neumann computers with acceptable speed or else that new, non-von Neumann computers must be designed to execute function-level programs efficiently.

A non-von Neumann computer

A number of efforts are seeking to develop such a computer. Two projects are typical of these efforts. One is being led by Professor Gyula Mago at the University of North Carolina at Chapel Hill, the other by Professor Arvind at the Massachusetts Institute of Technology's Laboratory for Computer Science in Cambridge.

Prof. Mago's design is the more radical of the two, since the parallelism inherent in functional programs is carried out to the fullest in the structure of the computer, in which storage and processing units are intimately linked. The computer consists of an arbitrarily large number of cells, each of which is one of two types: leaf, or L, cells and tree, or T, cells [see Fig. 2]. The tree cells are connected to form a binary tree, with each cell communicating with one parent cell and two child cells. The leaf cells form the base of the tree, each cell being the child of a T cell; each L cell is connected to its two neighboring L cells in addition to its parent T cell. All L cells are identical, as are all T cells, so that the overall design of the computer is simple and well adapted to VLSI technology.

In operation, functional program expressions, including their data, are fed into the L cells. The T cells then partition the expression, breaking it down into independent subexpressions. Each subexpression is composed of a function and the data to which it is being applied; each can be evaluated at the same time as all the others. The partitioning of the T and L cells in this way divides the whole machine into a number of subtrees. Each subtree is a smaller machine applying its own program to its own data.

The L cells act both as storage units and as processors, while the T cells manage communication between the L cells. By its very structure, the cellular computer avoids the problem of addressing, since its operation ensures that when the moment comes to apply any subprogram, its data will be in the adjacent L cells to its right.

The operation of the computer consists of a series of upward and downward sweeps of information as increasingly large parts of the functional expression are computed. The process begins with the innermost parts, which can be calculated immediately, and ends when the entire expression has been evaluated.

The T cells distribute the microprogram required by each L cell to apply the operational part of the expression to data in neighboring cells. For example, a microprogram might instruct the L cell to send its contents to the T cells above that are working to evaluate the same expression. Thus, during each machine cycle, information will sweep up to the top of the subtreee working on a given subexpression, and the information collected there will be sent down for further use by the L cells. At the end of each downward sweep, another stage in the evaluation of the subexpression will be completed.

Each cell of the machine is a fairly simple microprocessor containing both a CPU and a very small store. The L cells have a small store for microprograms, local storage for the symbol stored in the cell, its level of nesting in the expression, some condition registers, and a CPU.

The T cells are still simpler, basically containing only data registers and very simple processing units to direct the data and perform simple operations on it.

The basic simplicity of the cells and the fact that there are only two kinds means that the cost of designing and building such computers, even very large ones, should be manageable. Prof. Mago envisions computers with as many as a million cells, each with a few thousand circuits, with a number of cells on each VLSI chip.

Prof. Mago has made detailed calculations of the speed of operation of such computers. Performance of several billion instructions per second may be feasible in some applications. Yet,

because of the simplicity of construction, such computers might be no more expensive that current large von Neumann machines. Since these machines implement both primitive programs and program-forming operations with microprograms, their machine language is a higher-level language than current so-called high-level languages.

At present, Prof. Mago is designing the elementary chips for the cellular computer and expects to produce the first prototype chips in a couple of years.

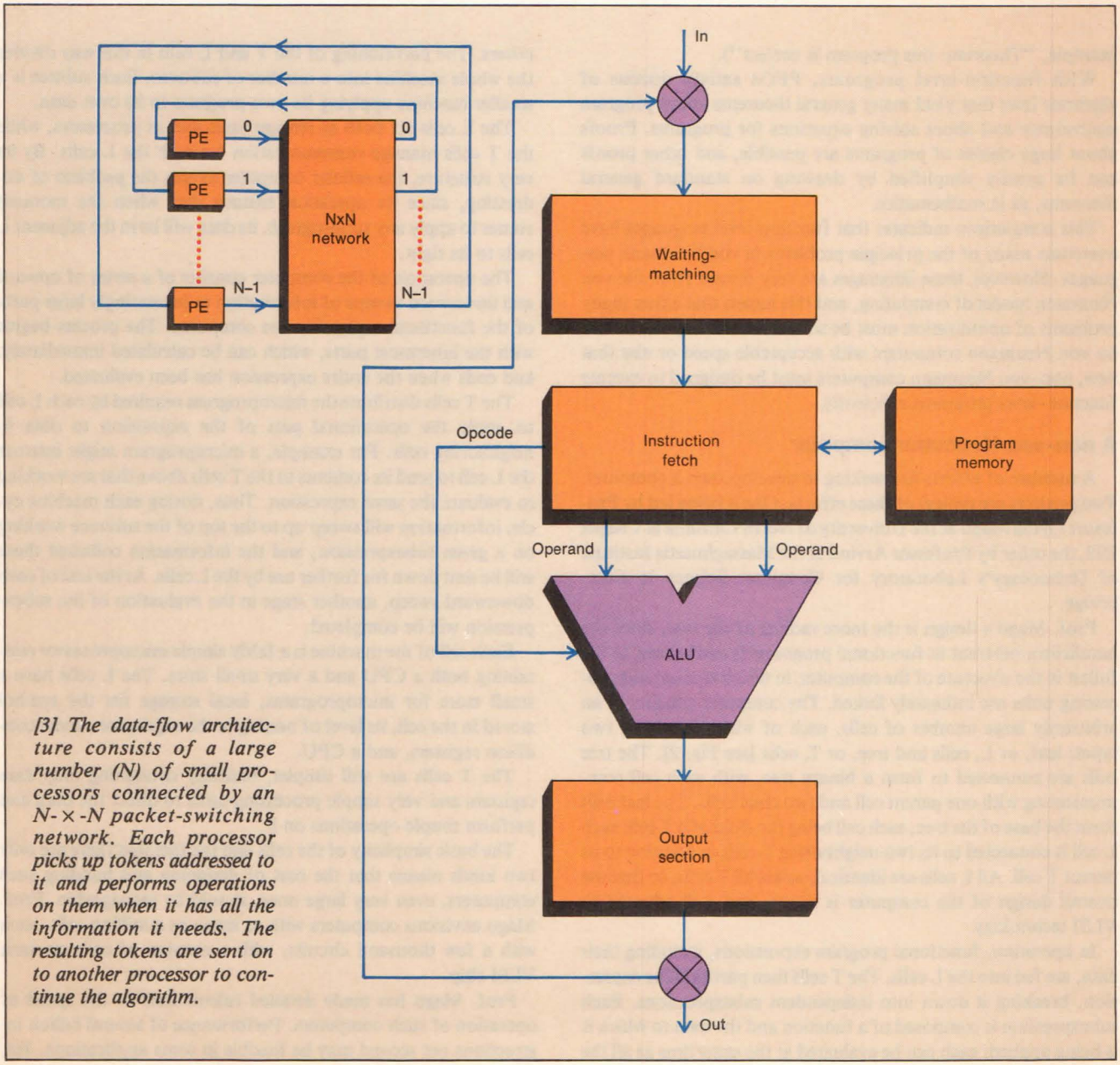
An alternative approach

An important alternative approach to implementing functional languages is the data-flow architecture of Prof. Arvind at MIT (data-flow ideas were first elaborated by Jack Dennis at MIT and later by Paul Kosinski at IBM Research Center, Yorktown Heights, N.Y.). Unlike the Mago machine, Prof. Arvind's design is intended for use with any functional language, not just the language we have been discussing. All such languages are compiled into a graphical data-flow language.

The Arvind machine itself consists of up to several thousand identical microprocessors connected by a packet communication network that allows any unit to send "tokens" to any other unit. Each token contains a piece of data and a "tag" telling to what processor the token is to be sent and with which other tokens its data is to be combined. When any processor receives a token, it matches it with its mate, performs the appropriate operation on the data of the token pair, forms a new token with the result, and sends it to the appropriate processor for further treatment.

Each processor has several sections [see Fig. 3]. A token normally arrives at the waiting-matching section, which contains tokens waiting for their mates. When a pair of matching tokens is formed, it is sent to the instruction-fetch section, which retrieves the needed instructions from its program memory and sends them with the data from the tokens to the arithmetic-and-logic unit (ALU). The ALU combines the data using the instructions to form a result. Finally this result is sent to the output section where it is incorporated into a new token whose tag is computed from the tags of the input tokens and from the addresses given in the instructions.

As much as possible, code related to a single loop within a pro-



[3] The data-flow architecture consists of a large number (N) of small processors connected by an N-×-N packet-switching network. Each processor picks up tokens addressed to it and performs operations on them when it has all the information it needs. The resulting tokens are sent on to another processor to continue the algorithm.

The stores problem

Von Neumann programs depend on the details of the storage plan that positions their data in the store (a store is a set of named cells, each cell containing a word of problem data). Programs must also provide the structure for their data, which resides in the store as an unstructured set of words. Thus larger programs can be built only from smaller ones that share a common storage plan (where is the data and what is its structure?), with the result that they must all be planned and written together. This prevents building up large collections of programs that can be used over and over to make larger ones.

The word-at-a-time nature of programs is another important factor, along with the need for storage plans that interferes with their universal applicability.

The net result of these difficulties is that programming takes a great deal of time to learn and a great deal of time to do. In addition, prepackaged programs are often so inflexible as to severely limit their use, or they offer so large a catalogue of options and features that it is very difficult to learn how to use them. Thus the accomplishments of the vast army of programmers is not even cumulative.

One way to understand the basic problems of von Neumann programming languages is to observe that their programs always map stores into stores. However, the *purpose* of a program is to map objects into objects—for example, to map a matrix into its inverse, or a file of transactions into a file of responses. The purpose of a program is *never* to map stores into stores, yet this is what all von Neumann programs do. Thus the programmer must translate the purpose of his program—say, to map matrices into their inverses—into a mapping of stores in which the input matrix occupies certain cells and the results others.

This disparity between the purpose of a program, on one hand, and its actual store-to-store mapping, on the other, is the source of the difficulty in building von Neumann programs from smaller ones. Suppose there are two programs, one to invert matrices, the other to transpose them, but they have not been planned together. Now a program to calculate the inverse of the transpose of a matrix is desired. It would seem a simple matter to form the composition of the two program, which first does "transpose" and then does "inverse," to get the desired program. But unless both programs have a common storage plan (and independent programs generally do not), with the output cells of "transpose" coinciding exactly with the input cells of "inverse," the composition of the programs will not achieve the composition of their purposes: the resulting program will be meaningless.

It is possible to write special von Neumann programs called subroutines whose storage plans can be altered when they are *used*, rather than being fixed when they are written. Such subroutines can be reused more easily than ordinary programs but are less convenient for building larger programs than functional programs, which do not depend on storage plans. —*J.B.*

gram is assigned to a physically related group of processors so that communication time between processors is minimized.

Prof. Arvind's group is now working on the detailed design of a 64-processor prototype machine, which they expect to be operating by the end of 1985.

It is too early to say which of these approaches to functional-style hardware will prove more fruitful, or whether some other approach may in the end be more suitable than either. Once prototypes are operating, it will be possible to compare the actual performance and costs of the various designs.

Problems remaining

A significant number of issues in functional programming need elaboration before commercialization can be considered. The most important relates to the handling of secondary and permanent storage. In addition, hardware must be built and tested, microprogramming systems elaborated, and problems run.

Before any of the projects developing parallel computers for functional programs have built a cost-effective model, some simple sequential computers for functional languages may evolve to fill the gap between today's computers and the parallel, non-von Neumann computers of tomorrow.

Were non-von Neumann computing to be widely adopted, its impact would likely be profound. Not only would programming time be reduced and repetitive programming of similar problems be largely eliminated, but programming of many applications could be simplified so that each one of millions of potential users could write programs for his own needs without the help of a professional programmer. Combined with the increased speed that would be possible with new computer architectures and VLSI design, a vast expansion of computer applications is entirely conceivable. Many tasks, such as visual recognition and computer graphics, involving many parallel computations could become much easier with such an approach. Finally, the design and manufacturer of hardware based on many identical parts could become much cheaper than current hardware. Overall, the possible advantages of non-von Neumann computing seem to justify a much larger commitment of resources to its rapid development than is currently being made.

To probe further

The problems of conventional programming methods are discussed in "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," by John Backus, *Communications of the ACM*, August 1978, pp. 613–41. The difference between the function-level style and the LISP style of programming is elaborated in "Functional-level programs as mathematical objects," *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, pp. 1–10, Association of Computing Machinery.

For a more complete description of the Mago machine see "A Network of Microprocessors to Execute Reduction Languages," Parts I and II, *International Journal of Computer and Information Sciences*, October and December 1979, pp. 349–85 and pp. 435–71.

"The U-interpreter," by Arvind and Kim P. Gostelow, IEEE *Computer*, February 1982, pp. 42–49, contains a description of the Arvind machine; a further description is found in "A multiple-processor data-flow machine to support generalized procedures," by Arvind and Zinod Kathail, *Proceedings of the Eighth, International Symposium on Computer Architecture*, IEEE and ACM, pp. 291–302.

About the author

John Backus is an IBM fellow at the IBM Research Laboratory in San Jose, Calif. He headed the group that produced the Fortran language and its first compiler. He also participated in the design of the international programming language Algol and proposed the language called Backus-Naur Form, or BNF, used to describe its syntax. A member of the National Academies of Sciences and of Engineering, Mr. Backus holds B.S. and A.M. degrees in mathematics from Columbia University. ◆