HIGHLY CONCURRENT VS. CONTROL FLOW COMPUTING MODELS

by

ROBERT CLARENCE MARSHALL

B.S., University of Rochester, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:

[illegible]

Major Professor

ACKNOWLEDGEMENTS

Thanks are due to my major professor, Dr. David Gustafson, for his guidance and encouragement. Although I am solely responsible for final form and content, Dave suggested the general organization of the report. I would also like to thank NCR Corporation who contributed many resources during report preparation.

I would like to give special thanks to my wife, Mary Ann, for her patience, love, and understanding, and for typing the manuscript.

TABLE OF CONTENTS

APPENDICES

PAGE

LIST OF TABLES

PAGE

LIST OF FIGURES

CHAPTER 1

Introduction

This report reviews, contrasts, and compares two classes of computing models:

- highly concurrent models, in which concurrent operation is implicitly assumed;
- control flow models, in which sequential operation is implicitly assumed.

The highly concurrent class is represented in the report by two models: the data flow and functional models. Highly concurrent models are being made practical for commercial implementation by advancing technology.

The control flow model is represented by von Neumann computing principles. This model has been identified with digital computers since the inception of discrete computing machines.

The next section will summarize the organization of the report. Two key properties which significantly shape the form of the report will also be briefly introduced.

1.1 Introductory Remarks

The control flow model has had a dominant influence on digital computers. Until the advent of Large Scale Integration and Very Large Scale Integration (LSI/VLSI), it had literally become synonomous with "computers."

Recently, the importance of formalizing the study of models has become more apparent as LSI/VLSI techniques have introduced technological and economic changes in design that favor non-control-flow models.

The data flow and functional models are different in two important respects from the control flow model:

- they are designed for implicitly concurrent operation;
- they are not history sensitive.

The first property has evolved as a natural consequence of the improving technology and is highly advantageous, but the second is more a product of our current scientific position and is not always a desirable property. History sensitivity is just the ability for data values to be stored internally for indefinite periods and utilized whenever desired.

History sensitivity is at once a strength and a weakness of the control flow model. Internal storage of data values enhances high-volume commercial and data file processing capabilities, but it also introduces the side effects so well known to commercial computing. These side effects result in unexpected, additional values of variables assigned to memory locations which are multiply named. The multiple naming occurs in global portions of procedures. Global and common storage areas require synchronization primitives to be used in multiprocessed sections of code. This severely restricts the ability of the control flow model to be used well in the design of concurrent routines.

Models that eliminate unwanted side effects by restricting or eliminating history sensitivity allow easy and efficient concurrent design, but only at the expense of internal storage capabilities. Examples of such models include the data flow and functional models. Functional models have the capability of being extended to add a history-sensitive property (FFP model in Section 2.1.2).

Together, concurrency and history sensitivity present the best opportunity to compare and contrast the highly concurrent models with the control flow model. The report will return to considerations of these two properties frequently, particularly in Section 2.3.

Chapter 2 will present discussions of the abstract highly concurrent models and of the abstract control flow model. Some key points of comparison between the two types of models will be discussed in Section 2.3.

Chapter 3 will consider implementations of the data flow and functional models. A discussion of parallel taxonomies will close the chapter.

Chapter 4 concludes the report. Included is an allegory representing the fallacy of designing do-everything programming languages without due consideration for the attributes of programmer ease of use, algorithm complexity, and underlying technological advances. Complex von Neumann designs may someday find it difficult to locate an architecture for implementation. Languages of the future must never lose sight of architectures upon which they can be realized.

A third significant area of difference between highly concurrent and control flow models is not so apparent until one attempts a comparison between them. Highly concurrent models, such as the data flow and functional models, are much easier to consider apart from their implementations, simply because their abstract structures (i.e., their "models" as opposed to their "implementations") were developed separately from any fixed ideas about specific hardware realizations. During the early development years of control flow computing, the concept of "model" was rarely considered separately from implementation, and the development of hardware realizations drove the structure of the model. As a result, no separate theoretical structure now exists for the control flow "model" which can rival the comparable highly concurrent models. This report considers the von Neumann "model" in Chapter 2, and many von Neumann concepts will be seen to require some reference to hardware concepts, such as "registers" and "memory locations". Since so much is known of von Neumann implementations, little would be gained by presenting one in Chapter 3; therefore, Chapter 3 concentrates on data flow and functional implementations, while von Neumann implementations are discussed only during Sections 3.3 (on parallel taxonomies) and Section 3.4 (comparison of control flow and highly concurrent implementations).

The next section summarizes the impacts of LSI/VLSI technology which are bringing highly concurrent models to the forefront. Impacts on hardware, software, and design will be discussed.

1.2 Structural Impacts of LSI/VLSI Technology

The control flow model was the model for almost all digital computers in the early 1970's, and few designers had given much thought to any other. The cost functions of computing included expensive (global) memory, expensive discrete components, and a "medium" scale of integration allowing chips fabricated with, perhaps, 1000 transistors per chip. Control flow models tended to minimize the total cost of computing. At about that time, techniques for Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) began to emerge.

VLSI and LSI techniques were revolutionary and would offer the promise of fabricating chips containing 10^5 individual transistors by 1980 and 10^7 or 10^8 transistors by 1990 ([NECH79], [SCHW80]). Meanwhile, the cost of memory was decreasing substantially. With VLSI technology, it became easy to implement interconnecting networks of vast numbers of processing and memory elements on single silicon chips. The cost functions for these kinds of chips were dramatically changed from all that had gone before; now, cost (and efficiency) of a device was more dependent on the total lengths of interconnecting paths between elements than on the elements themselves ([MEAD80], [MAGO80], [SCHW80]). The global memory structures of control flow computing were no longer acceptable, since local memory with each processing element minimized interconnections and improved processing efficiency.

The primary problem posed to computer scientists and engineers by VLSI became one of how to best exploit this technology. Sugarman in [SUGA80] envisions VLSI design tasks falling into two categories:

- structuring control flow designs into VLSI;
- abandoning control flow designs totally to utilize the full power of VLSI.

Only in the latter category can the full promise of concurrency available in VLSI systems be tapped. However, as Rem notes in [SUGA80], computer scientists are only now mastering the theory of sequential programming, and they are currently ill-prepared to supply programming techniques to make VLSI structures a reality. Computer design engineers have discovered that design times of fifty man years could be required to design and fabricate a 100K device chip without improved computer-aided design techniques [NECH79]. (Remember, a 100K device chip is feasible today; by 1990, 10^5K device chips may be feasible!) The challenges inherent to VLSI design are many, but the rewards could be very great.

Section 1.2.1 discusses some impacts of LSI/VLSI technology on hardware structures, while section 1.2.2 discusses impacts on languages and software. Section 1.2.3 presents a change in viewpoint for total system design that is necessary for VLSI design.

1.2.1 Impact on Hardware Structures

How will VLSI affect conventional hardware structures? Mead and Conway [MEAD80] provide some interesting insights. This section is a summary of their findings.

Both processing and memory elements can easily be implemented in VLSI: "A human brings to an organization what VLSI brings to a circuit: both combine processing and memory effortlessly." Long interconnecting wires which impede communications are eliminated. The resultant systems support very high degrees of concurrent operations.

Mead reviews processor/memory architectures (control flow machines) in terms of resource usage. For large global memory systems most memory and memory wiring is idle most of the time. A four megabyte memory of 32 bits/word width, for example, may access only one word of four million 32-bit words at one time. Many resources are expended by communication of data words over relatively large distances (buses, etc.). A discussion of memory locality and how it's implemented in a memory hierarchy illustrates an inverse relationship between memory size (M) and speed of access. The access time, T, is proportional to the square root of memory size, M. For register memory (M_r), cache memory (M_c), primary memory (M_p), and secondary memory (M_s) (i.e., disks), a model for memory access time is presented:[1]

$$T_{avg} = F_r\ (M_r)^{\frac{1}{2}} + F_c\ (M_c)^{\frac{1}{2}} + F_p\ (M_p)^{\frac{1}{2}} + 100\ F_s\ (M_s)^{\frac{1}{2}}$$

Typical frequency values are:

$$F_r = 0.6$$
$$F_c = 0.38$$
$$F_p = 0.02$$
$$F_s = 5 \times 10^{-6}$$

Access to secondary storage dominates.

Two other methods have been used to increase speed:

- pipeline structures;
- multiprocessor structures.

Pipeline structures with local memory increase processing power to a greater factor than just by the number of processors provided because each processor can have a smaller local memory. For example, a two-processor pipeline more than doubles available processing power:[2]

---

[1] [MEAD80], pp. 266-7.

[2] [MEAD80], p. 267.

$$T_e = \frac{1}{2} (M/2)^{\frac{1}{2}}$$

T_e, execution time, is about 1/3 the time for one processor (note that this formula ignores interconnection costs). This effect occurs as the result of two factors:

- doubling number of processors doubles speed;
- localizing memory to each processor and reducing memory size for each increases speed.

Effective multiprocessor systems in VLSI will probably be hierarchical structures, such as binary trees of processors (see section 3.2). Simple systems are combined into large, complex structures consisting of perhaps hundreds or thousands of elemental processor and memory combinations. The binary tree is a structure with some ability to utilize all processors concurrently. In general, trees also have other advantages:

- can be tested comparatively easily;
- general computing structures for a general class of problems are well-represented by trees.

On VLSI chips it is extremely important to minimize wire length to minimize both time delay and energy dissipation. There is a definite tradeoff between increasing processor/memory combinations and the resultant area required for wires:

- hierarchical structures improve performance to a point;

- if a hierarchical structure gets too large, it begins to require too much interconnecting wire area.

With the emergence of VLSI problems must be framed from the beginning in terms of concurrency. In this environment communication is expensive and computation is not. VLSI presents a challenge to computer science: "Develop a theory of computation that accommodates a more general model of the costs involved in computing. The current VLSI revolution has revealed weaknesses of a theory too solidly attached to the cost properties of a single sequential machine." [MEAD80]

Summarizing these considerations in [MEAD80] we can list some properties advantageous to VLSI hardware structures:

- large numbers of fairly simple processors connected together in complex hierarchies, such as binary tree structures;
- small amounts of local memory associated with each processor;
- pipeline structures;
- techniques to optimize wire area (minimum) versus hierarchy size;
- concurrency implicit to the model;
- new theories of computation embracing concurrent rather than sequential operation.

Finally, the huge area of parallel algorithms is still in its early developmental stages. Kung ([KUNG80] and [MEAD80]) reviews this field. Because so many of these new algorithms will be implemented in hardware structures, there is going to be a major impact on computer scientists to interact with other disciplines during computer design. Lattin ([NECH79] and [SUGA80]) cites a growing crisis in VLSI design in which the sheer numbers of devices in a structure such as a microprocessor can require inordinate design times. This affects computer science in two ways:

- more must be known about parallel algorithms in general, so structures can be designed using standard devices and/or techniques, rather than custom-designed devices, etc.;[3]
- much more of the design process must be done by utilizing computers (computer-aided design - CAD).

The area of parallel algorithms is so large it would require a separate report to cover it adequately.

---

[3]In [SUGA80] Lattin maintains the ratio

$$D = DT/DC$$

Where DT = devices of all kinds,
and DC = custom designed devices
for the intel 8086 was such that D = 4.4. He feels D = 20 must be attained to cut a 60 man-year effort to 5 man-years.

The ultimate impact upon conventional control flow computing will obviously be very large.

1.2.2 Impact on Software Structures

The impact on sequential programming languages consists in part of techniques to translate conventional high level language programs to equivalent parallel representations as in [ALLA76], or to compile conventional language programs into code for one of the parallel architectural models, as in ([JOHN80], [KUCK79]). Kuck discusses compiling techniques for structures consisting of arrays of microprocessors. This report will not examine these techniques in detail.

Newer highly concurrent languages and processing techniques are also appearing. Brock and Montz [BROC79], Gurd and Watson [GUR680], and Treleaven [TREL79] all discuss some of these language structures. Treleaven includes an example program written in a data flow language which will be examined in chapter 2 (Section 2.1.1). This kind of language will require programmers to alter their views of machine communications and structures to fit the highly concurrent models. Gurd and Watson contains an excellent discussion of some flow-graph techniques for structuring parallel software.

1.2.3 Impact on Design

Section 1.2.1 ended with a discussion of parallel algorithms and the impact these would have on the design of software structures. An important additional consideration for these algorithms in the VLSI environment was that many would also affect hardware design structures. In control flow computing the hardware design activity was distinctly separate from both the language and application design activities. Hardware design actually drove the other two activity areas, and, to a great extent, language design drove application design. Thus, a design hierarchy with hardware design at the top and application design at the bottom was typical. In the era of expensive discrete hardware components, expensive banks of global memory, and the sequential emphasis on computing structures, this made some sense. In VLSI design it is becoming a much less relevant approach.

Schneck [SCHN79] outlines a new design approach in which the application, design, and implementation areas of algorithms, hardware, and software are very intertwined. In this approach the algorithm design activity for the application, not hardware design, drives the total effort. Hardware and software design activity areas will be at the same level in this hierarchy and will be nearly indistinguishable in some important ways. See Figures 1 and 2 for illustrations of the old and new design hierarchies.

In the control flow environment, computer scientists have unfortunately grown too accustomed to their niche in the old hierarchy. Feature laden, complex, von Neumann based "new" languages such as PL/1 and Ada are always appearing, while comparatively little has been done on the design of truly innovative languages which would fit other existing models more satisfactorily, or help to define new models. This design attitude will have to change, since the inputs of computer scientists will affect machine design much more directly in the VLSI era. Chapter 4 will return to this subject.

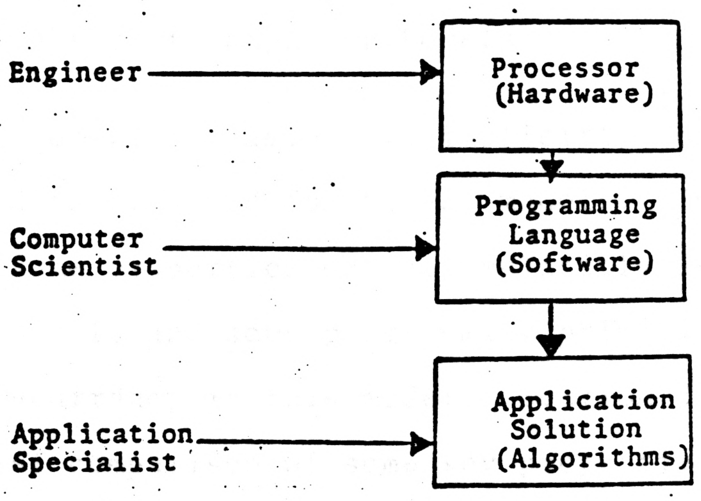


Figure 1

Control Flow Design Hierarchy

[SCHN79]

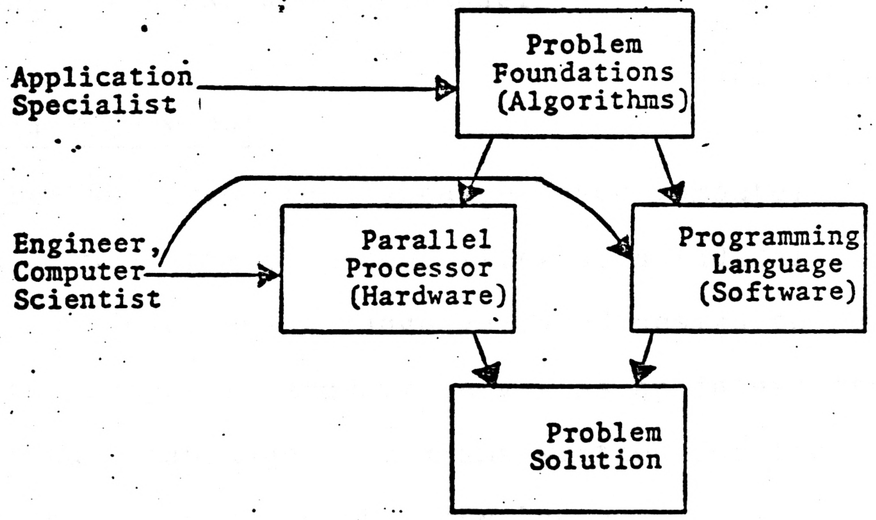

Application Specialist
Problem Foundations (Algorithms)
Engineer, Computer Scientist
Parallel Processor (Hardware)
Programming Language (Software)
Problem Solution

Figure 2

VLSI Design Hierarchy

[SCHN79]

CHAPTER 2

Abstract Computing Models

This chapter will examine two implicitly concurrent models: the data flow and functional models. Section 2.2 will examine some properties of the implicitly sequential control flow model, and some programming primitives necessary to realize concurrency in this model. Section 2.3 will then present a brief comparison of some key properties of highly concurrent and control flow models.

2.1 Highly Concurrent Computing Models

Section 2.1.1 will examine the data flow model, and Section 2.1.2 the functional model. Chapter 3 will discuss implementations for these abstract models.

2.1.1 The Data Flow Model

Dennis [DENN80] advocates language-based computer design, which ensures the programmability of a radical architecture. He describes a language-based design to be one in which the computer hardware serves as an interpreter for a specific base language. Programs written for the computer must be expressed in the base language.

Future supercomputers must support massive concurrency in order to achieve significant performance increases; therefore, a base language for such machines must necessarily allow simple, implicit expression of concurrency on a very large scale.

Dennis feels that conventional control flow languages have an intrinsic, fatal design flaw: they are based on a global state model of operation. In the next computer generation, at least for large scale scientific computation, he believes this flaw will force abandonment of control flow languages. At this time he recognizes only two alternatives: the functional (applicative) languages to be discussed in Section 2.1.2 and the data flow models.

Dennis' subsequent explanation of the data flow model is now reviewed. His discussion has a simplicity and precision which makes the topic easy to understand.

In data flow models machine-level programs present a new view of instruction execution which departs radically from the sequential one. An instruction is automatically ready for execution when all operands have arrived. Relative positions of instructions are irrelevant, and data flow computers do not have location counters. A direct consequence of data-activated instruction execution is that many instructions may be ready to execute at once. Therefore, highly concurrent operation is an integral part of the data flow concept.

The base language for most data flow architectures is a representation called data flow program graphs. In most cases data flow computers are a form of language-based architecture in which these graphs are the base language. Thus, the language and the architectural concepts of data flow models are explicitly bound together at design time, and architectural concepts do not force language representation as happened in control flow computing. Data flow program graphs are a formally specified set of interfaces bridging system architecture and the user source programming language. Figure 3 illustrates the concept.

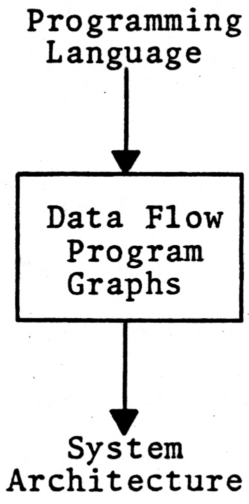


Figure 3

Language-Based Design Hierarchy for Data Flow Computers

[DENN80]

In the design environment implied by Figure 3, the computer architect and language implementor have sharply defined tasks:

- the architect must define a computing machine which implements the formal behavior of program graphs;
- the language implementor must devise translators for source language programs which translate source into equivalent data flow program graphs.

The cooperating nature of the design process is clear when the role of the program graphs in the scheme is understood.

Data flow graphs are represented by collections of activity templates, which are information packets stored in memory. The role and structure of activity templates will become clear as the discussion proceeds. Basically, an activity template represents an action entity, such as an operator, which requires a finite number of operands in order to execute. The template records all operand fields and their readiness to be used in an operation. After execution, template fields are utilized to record and forward results to succeeding templates.

Data flow program graphs are composed of actors and arcs. Actors are connected by arcs and consist of both input and output arcs which carry data values in the form of tokens. Thus, arcs are communication paths between actors, and values travel upon these paths as tokens.

Figure 4 shows two actors connected by an arc upon which a token is being transmitted from actor 1 to actor 2.

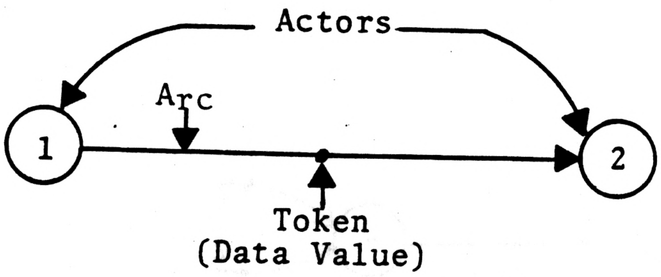


Figure 4

Segment of a Data Flow Program Graph

Firing rules for tokens govern the placement onto and removal from input and output arcs of tokens and their associated values. For an actor to be enabled, a token must be present on each input arc, and no tokens can be present on any output arcs. An enabled actor may be fired. If the actor is an operator, firing entails applying the specified actor function to each input token value and placing the resultant tokens with computed values on the output arcs. Figure 5 illustrates the firing process.

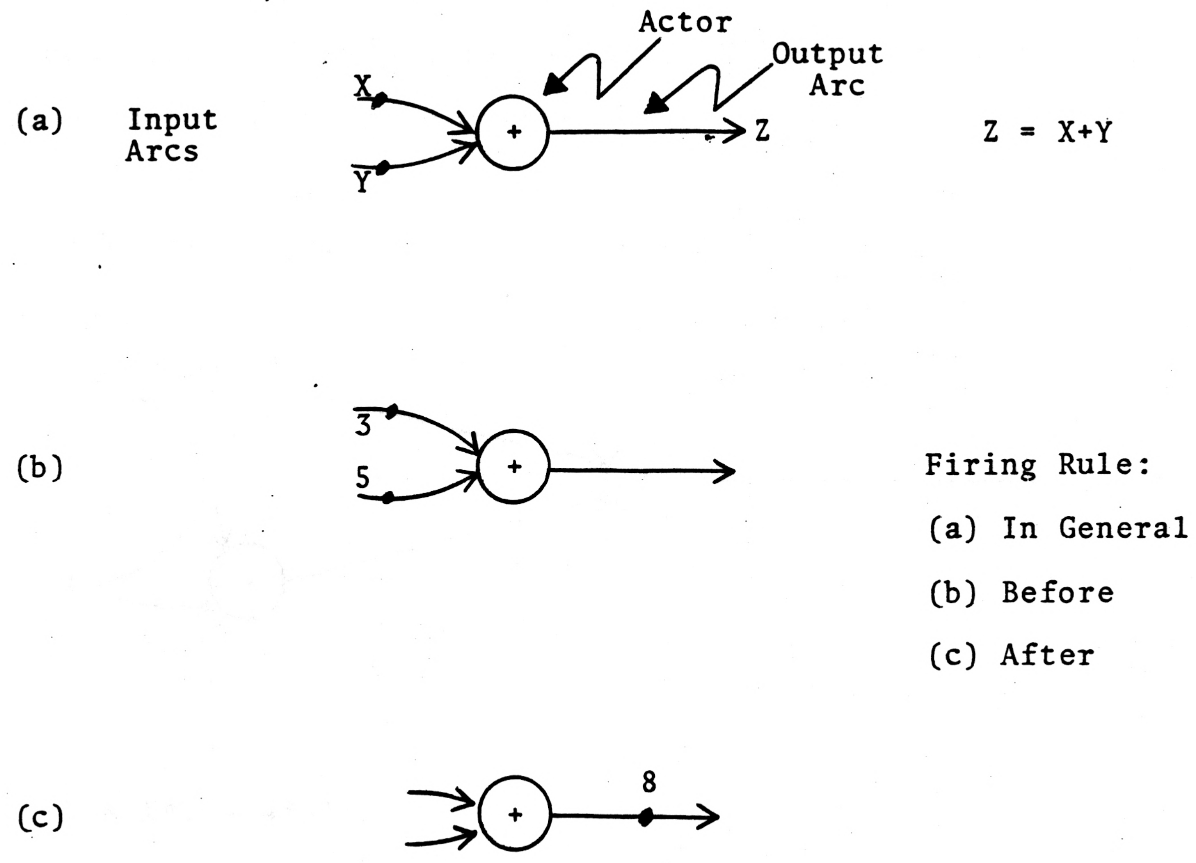


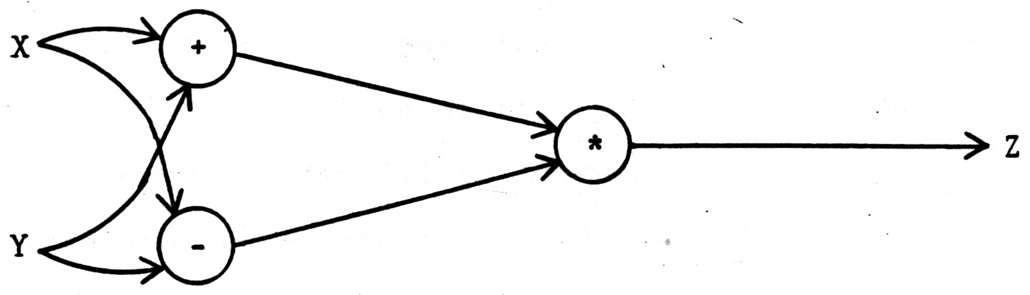
Figure 5

Firing Rules

[DENN80]

An arbitrary number of operators (or actors) may be connected to form program graphs. Figure 6 presents some examples of program graphs.

(a) Z = (X+Y)* (X-Y) [DENN80]



(b) Z = X*Y - 4*A*(X+Y)

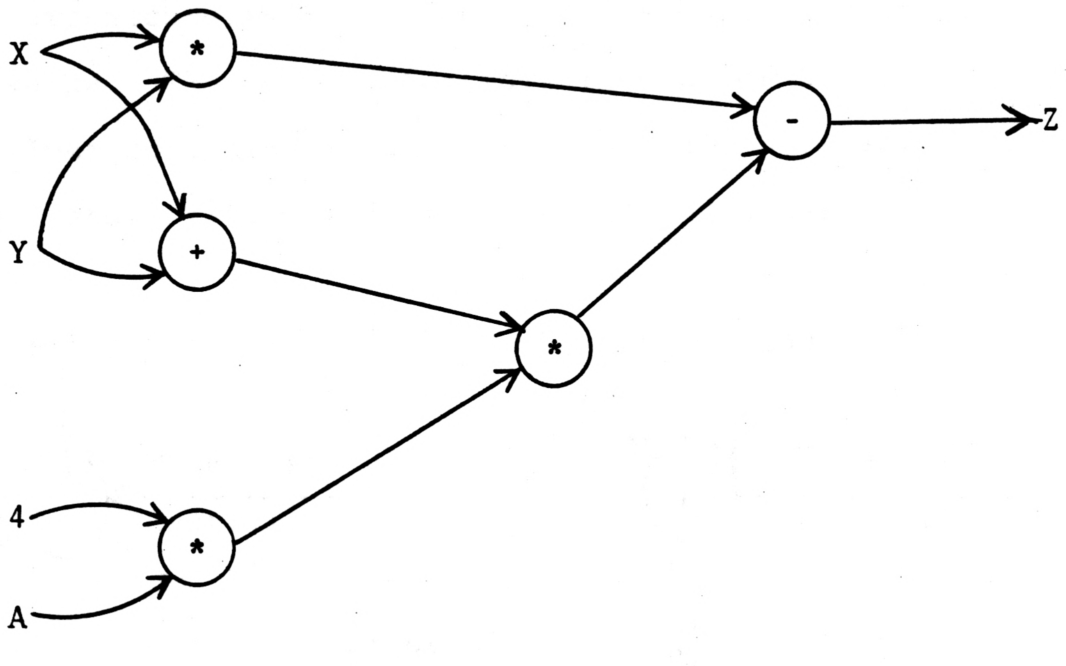


Figure 6

Examples of Program Graphs

There are many different types of actors: actors for each arithmetic operator, actors for copying data values to arbitrary numbers of output arcs, actors for merging data values, etc. The natures of these actors make conditional executions, iterations, and recursive computations fairly easy to implement. For a complete discussion of different types of actors, refer to [GUR680]. For the purposes of this report only simple switch and merge actors will be discussed.

Switch and merge actors control conditional executions and iterations. They do this by controlling the routing and selection of data values. An actor of one of these types operates by testing a boolean input value on one of its input arcs. The switch actor selects an output arc according to a true or false boolean control input value. The merge actor forwards one of two input data values according to an input (control) boolean. Figure 7 shows switch and merge actors and arcs.

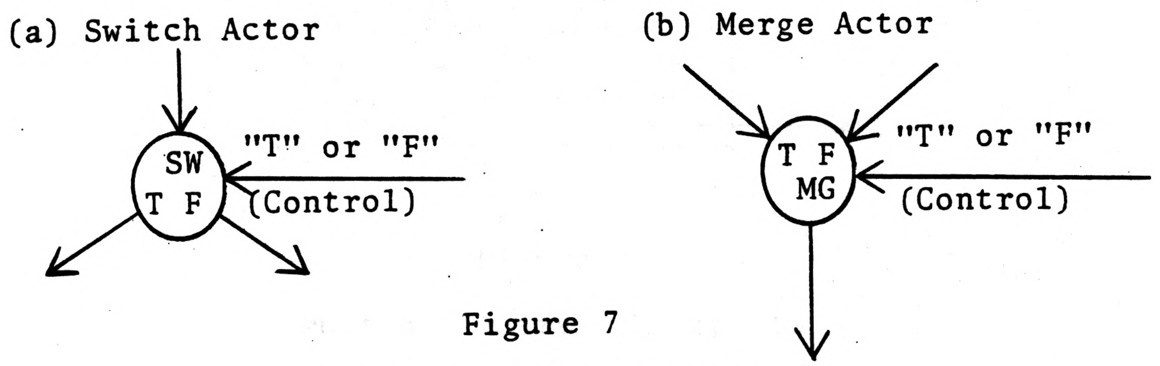


Figure 7

Switch and Merge Actors

[DENN80]

At the machine level, data flow programs are represented by activity templates. A program is a collection of these templates. Each activity template corresponds to one or more actors of a data flow program graph. An activity template consists of a collection of data value fields. For example a multiply template consists of four fields:

- an operation code (Multiply);
- two receiver fields to receive input operand values from previous operations;
- one destination field to store and forward the resultant product value to succeeding operations.

Figure 8 displays a multiply template. Figure 9 shows the corresponding composite structure of templates for one of the data flow program graphs in Figure 6.

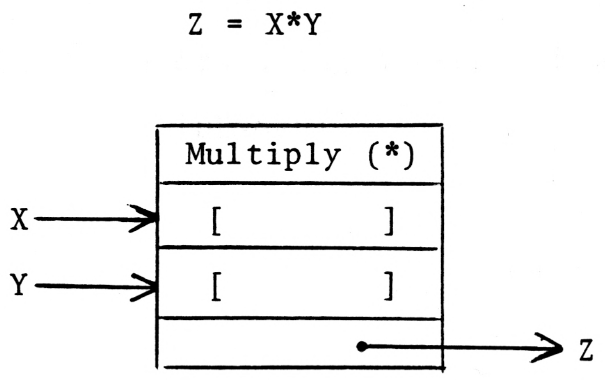


Figure 8

Multiply Activity Template

Z = (X+Y)*(X-Y)

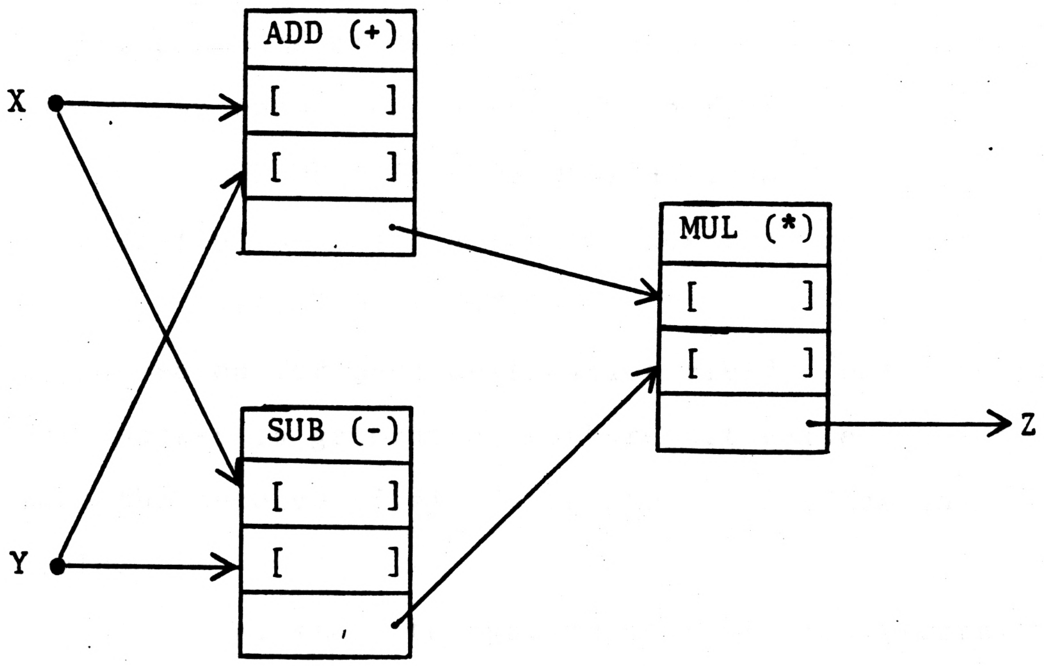


Figure 9

Composition of Operators using Activity Templates

[DENN80]

Activity templates control the execution of a machine program. Execution of a template is activated by the presence of an operand value in each receiver field. An operation packet of the form

<OPCODE,OPERANDS,DESTINATIONS>

is operated upon, and a result packet of the form

<VALUE,DESTINATION>

is passed on for each destination field. When the result packet is generated, each result value is placed into the receiver field designated by its destination field.

It is possible to analyze control flow programs and produce data flow machine object programs ([ALLA76], [JOHN80]). Indeed, conventional compilers with optimizing phases seem fairly easy to adapt in this way, since many of these compilers represent programs as directed graphs, and such representations are very close to the machine language of a data flow computer. A prototype computer of this kind has been successfully built, and the optimizing phase of a conventional compiler has been modified to generate code for it [JOHN80]. This approach holds much promise, since the underlying data flow model should be fairly transparent to the high level language programmer.

However, the semantics of data flow and control flow languages differ greatly [TREL79]. In data flow models the order of assignment statements is irrelevant, and these are interchangeable since they are activated only by the availability of input data. To insure determinate operation, assignment statements must obey a single-assignment rule: an identifier can be assigned values at only one point in a program. This is necessary since an identifier is mapped to an arc, or data path, in data flow models and not to a memory location. History sensitivity is not a property of the model.

Side effects are not present in data flow languages. In the control flow model, mappings of multiple identifiers to the same storage location can cause unexpected results to occur. This happens ordinarily through subroutine parameter mappings and common storage shared by multiple modules. This phenomenon depends upon the property of history sensitivity, and thus it cannot occur in data flow languages. Because there is no necessity to coordinate common storage areas, side effects are absent from the data flow model and concurrency is highly enhanced. However, the price of eliminating history sensitivity from the model is not all positive; Chapter 4 will return to this subject.

One reason data flow is a popular research area is that textual data flow programming languages may be developed that share a few properties with control flow languages. For example, they can utilize assignment statements, arithmetic expressions, conditional statements, iteration, recursion, and function declaration [TREL79].

Representation in data flow languages is straightforward. Data identifiers are mapped to data paths and operations to data flow instructions. The von Neumann principle of program-data indistinguishability is lost, since these mappings are not to memory locations. One author reached the conclusion that this indistinguishability principle should be re-established in the data flow model [SLEE80]. This would probably entail the establishment of a history sensitivity property.

Since the data flow model supports concurrency at a low level, this model will support the optimal data flow language directly and allow individual operations to be initiated as soon as input data are available. Studies of speed-up ratios show the best ones are linear in P, where P is the resource replication factor [STON73]. In a data flow model with large P (i.e., a very large number of processing units), the best way to achieve this speed-up is by supporting concurrency at as low a level as possible, since all higher level concurrency will then be automatically supported. Introduction of explicit statements, such as CALL and WAIT, to support concurrency will cause a negative effect on the linearity of P. When represented at a low enough level, there is the possibility of achieving a better increase in performance for a broad class of problems, since the system can then utilize the detailed representation of a program to maintain a very high overall resource utilization [TREL79].

Figure 10 illustrates a data flow representation of a quadratic roots program in a Pascal-like form [TREL79]. Recall that there is no relevance to the relative placements of the assignment statements in the QUAD-ROOTS function.

Due to time constraints, this report does not discuss many of the more advanced data flow concepts. Where appropriate, references are made to papers discussing recursion, acknowledge processing, and data flow multiprocessors. [DENN79] discusses another important concept: concurrent computation with streams. It is intended, here, only to discuss the basic concepts of the data flow model. The great potential for concurrent computation should be very clear.

```
function QUAD-ROOTS input (a,b,c:real)
                   output (x1,x2:real);

var temp:  real;

begin
     temp := SQRT(b*b-4*c);
     x1  := (-b+temp)/(2*a);
     x2  := (-b-temp)/(2*a);
end;

"main program..."
var i1,i2,i3: real; r1,r2: real;

begin
     i1 := ..;  i2 := ..;  i3 := ..;
     (r1,r2):= QUAD-ROOTS (i1,i2,i3);
     ...
end;
```

Figure 10

Example Data Flow Program Representation

[TREL79]

2.1.2 The Functional Model

Backus introduced a functional model of programming languages which is highly mathematical. He describes his functional structure in two different ways [BACK78]:

- informal discussion of functional programming, or FP, systems;
- formal functional programming (FFP) systems.

The FFP systems can be studied in detail in [BACK78]. In general, this section concentrates on FP systems.

Backus recognizes two parts of a programming language:

- framework: defines the overall rules of the system;
- changeable parts: existence is provided for by language framework, but specific behavioral aspects are not specified.

An example of changeable parts portions of control flow languages is the CALL/RETURN procedure mechanism, which in many languages is used to invoke modules of arbitrary function. The language framework always describes its fixed features and provides the minimal features and environment for its changeable features.

Backus strives to define a minimal framework which could generate most other features as changeable parts. His exact quotation follows: "if a language had a small framework which could accommodate a great variety of powerful features entirely as changeable parts, then such a framework could support many different features and styles without being changed itself."

According to Backus, von Neumann languages have large frameworks and limited changeable parts. Two properties of the von Neumann model seem to dictate this:

- word-at-a-time programming in which semantics are closely coupled to state, and every detail of computation changes the state;
- semantics closely coupled to state transitions implies every detail of every feature must be built into the state and its transition rules.

As an example of the rigidity of von Neumann languages, consider the primary techniques used for passing control to subroutines. The expression itself, "passing control," reveals the only real purpose of the techniques, which never evolved as expressive parts to alter the structure of a language to fit a problem. The purpose of such constructs are only to "modularize" large portions of program code. Typical CALL/RETURN mechanisms function as tying statements used only to glue sequentially-related but functionally independent portions of logic together. In themselves, they contribute little meaning to the language.

Functional techniques, such as the FORTRAN function defining and manipulating statements, are better in von Neumann computing than are their CALL/RETURN cousins. They can be utilized in a more expressive manner, since they can be embedded into complex arithmetic expressions. However, the nature of a function in the von Neumann model demands that a single-word result value always be computed and returned. This restriction keeps such techniques from having the power to expand von Neumann languages very significantly.

A CALL/RETURN scheme is often used to implement concurrency in von Neumann languages. Concurrency in these languages is not "fine grain" (i.e., concurrency is not consciously built into the von Neumann model at the lowest levels). Thus, some explicit technique is needed to implement a grosser kind of concurrency at the language level. It seems to follow that CALL/RETURN, the basic statements for "passing control," would often be extended to serve as concurrency controlling statements. Much problem continuity and clarity is lost by the usage of such constructs for concurrency, particularly since the original purpose of CALL/RETURN was for sequential passing of control, a technique which opposes a concurrent view.

In terms of problem clarity and understanding, CALL/RETURN mechanisms tend to detract from languages. Such compensating techniques as extensive English-language commentaries in the source code are necessary to maintain logical continuity of understanding. Very complex features must be added to these languages to strengthen them significantly and allow the language statements themselves to maintain logical clarity at the problem level. The resulting structure is very rigid and large.

Two of the basic problems with von Neumann languages seem to be:

- word-at-a-time programming;
- changeable parts have too little expressive power.

Backus' goal is to provide a language framework which can be expanded naturally, while simultaneously increasing the expressive power of the language. He approaches the problem at the point where new procedures must be created to solve a problem. A goal of his functional style is to allow this process of procedure creation to happen within this basic framework of the language while leaving the language problem oriented, and not construct oriented.

In order to provide powerful combining parts in a language, good combining forms must be available which can be used to fabricate new procedures from old ones. The control flow model provides primitive combining forms and makes using them difficult. Backus notes the split between what he refers to as the "expression world" and the "statement world" in the von Neumann model. "Functional forms naturally belong to the world of expressions; but no matter how powerful they are, they can only build expressions that produce a one-word result. It is in the statement world that these one-word results must be combined into the overall result."

As an example, consider the sequence of FORTRAN statements

```
A = SQRT(B*B+C*C+EPS(W-U))
D = X+Y*Y+A**3
```

Certainly, the expression to compute A does not lack elegance. It involves numerous arithmetic and functional applications; yet, its primary purpose is to produce a sequential result value to store in the location associated with A. This value can then be used in the following statement. No computation can be performed on the expression associated with D until the value for A is available, although the values for the subexpression X+Y*Y are independent of A and available for use while A is being computed.

The constant combining operations of single words necessary in control flow languages is something which detracts from the power attainable if the split between statements and expressions were not present. One goal of the functional model is to eliminate this arbitrary split.

Backus also aims to eliminate the usage of elaborate naming conventions in his functional model. Naming conventions require complicated mechanisms in the language framework which interfere with the use of simple combining forms. For example, subroutines require dummy arguments which must be mapped to the storage locations corresponding to the arguments of the invoking procedure.

Finally, Backus wants to provide powerful mathematical properties in his functional language framework which aid program proof and construction tasks. Control flow languages generally lack these properties; hence, they are difficult to reason about and prove. In functional programs "... programs can be expressed in a language that has an associated algebra. (The) algebra can be used to transform programs and to solve equations whose 'unknowns' are programs, in much the same way one solves equations in high school algebra." In the FP style algebraic transformations and proofs can utilize the language of programs directly, rather than the (extra) language of logic (which only talks "about" programs).

Iverson demonstrated that there can be programs which are neither word-at-a-time nor dependent on lambda expressions. With APL Iverson introduced new functional forms. Since APL assignment statements can store entire arrays at once, the functional forms are greatly extended beyond those of von Neumann languages. However, Backus notes three problems with APL:

- the split into expressions and statements is still there, albeit on a larger scale for expressions;
- APL has only three functional forms (inner product, outer product, reduction) which are not sufficient and are difficult to use;
- APL semantics is still too closely coupled to machine states.

As the experience of APL suggests, matrix operators introduce more powerful functional forms, but they do not (in themselves) solve all the problems of von Neumann languages. For example, Backus feels the effort to write one-line programs in APL by using the powerful matrix combining forms is partially motivated by the desire to remain in the "more orderly world of expressions."

Backus' eventual goal with FP systems is to utilize them in the design of applicative state transition (AST) systems. AST systems have the following properties:

- history sensitivity;
- loosely-coupled state-transition semantics in which a state transition occurs only once in each major computation;
- simple states and state transitions;

- dependence upon an underlying applicative system to provide the basic programming language and to describe state transitions.

An AST system is composed of three elements:

1) an applicative subsystem (i.e., an FFP system);
2) a state D that is the set of definitions of the applicative subsystem;
3) a set of transition rules that describe how inputs are transformed into outputs and how the state D is changed.

The programming language of an AST system is defined: it is that of the applicative subsystem (i.e., can be FFP system). The FP programming style described later can be used. The state D cannot change except at output time. The old state is replaced by the new state at output time. State transitions can have useful mathematical properties. Programming is not divided into expressions and statements.

Some other key advantages of AST systems are as follows:

- since the state cannot change during a major computation, side effects are eliminated, and independent applications can be evaluated concurrently;
- major new features are introduced by utilizing the common language framework;

- the framework is minimal and is the only fixed part of the system;
- the functional nature of names is exploited.

Backus feels that the new classes of history-sensitive models utilizing applicative styles and languages are key developments. If their superiority over conventional languages can be proven, the economic basis for developing new kinds of computers to best implement them will be established. The full power of large scale integration can then be better utilized in computer designs to produce more concurrent and efficient machines.

With this final goal for AST systems in mind, Backus outlines an approach for designing non-von Neumann languages:

- an (informal) functional style of programming (FP) without variables based upon the usage of combining forms for constructing programs;
- an algebra of functional programs;
- a formal functional programming system (FFP) to serve as the basis for AST systems;
- AST systems.

Magó's [MAGO80] cellular architecture in Section 3.2 is based upon this approach, and the resultant FFP.

FP systems are members of a class of simple applicative programming systems in which the only operation is that of "application." Programs in this type of system are functions without variables. In the language framework, a fixed set of combining forms called functional forms are defined. To these fixed functional forms are added some simple definitions: the combinations of fixed functional forms and definitions are the only building blocks available to construct new functions from existing ones. Variables and substitution rules are specifically excluded from the system. New functions become new operations in an associated algebra of programs.

The functions of an FP system map objects into objects and always require one single argument, or a tuple of arguments. These simple, highly-structured forms define the behavior of FP programs unambiguously and allow program proofs by algebraic methods.

An FP system is constructed of the following sets:

- a set O of objects;
- a set F of functions that map objects into objects;
- a (single) operation called "application";
- a set FF of functional forms used to form new functions in f;
- a set D of definitions that define some functions in F and assign a name to each.

Backus provides examples of these entities. Some examples from [BACK78] follow:

- objects

  1 ϕ 7.8 CDX <X,1,4.7>

  <xy,w,<<x>,h>,wz> $\perp$

- applications

  $+:\langle 1,2\rangle = 3$

  $2:\langle A,B,C\rangle = B$

  These are read "+ applied to the sequence <1,2> yields 3, and"the selector 2 applied to the sequence <A,B,C> yields B."

- primitive functions

  These functions are supplied with the basic FP system.

- selector functions

  $1:x \equiv x = \langle x_1,\ldots,x_n\rangle \rightarrow x_1;\ \bot$

  This is read "the selector function 1 applied to x is defined as the first element in the sequence (x_1) when $x = \langle x_1,\ldots,x_n\rangle$ and is undefined otherwise."

- identity

  $id:x \equiv x$

- reverse

  $$\begin{aligned} reverse:x \equiv\ & x = \phi \rightarrow \phi; \\ & x=\langle x_1,\ldots,x_n\rangle \rightarrow \langle x_n,\ldots,x_1\rangle;\ \bot \end{aligned}$$

- functional forms

  These are basic forms which are used to produce other functions by combination.

- composition

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

  f and g are preexisting forms.

- apply to all

  $\alpha f:x \equiv x = \phi \rightarrow \phi;$

  $x=\langle x_1,x_2,\ldots,x_n\rangle$

  $\rightarrow \langle f:x_1,f:x_2,\ldots,f:x_n\rangle;\ \bot$

- <u>definitions</u>

  A definition in an FP system is an expression of the form

  $$\underline{\text{def}}\ \ell \equiv r$$

  where ℓ is an unused function symbol and r is a functional form.

  - <u>def</u> IP≡(/+)·(αx)·transp

    This is Backus' definition of inner product, IP, using the following functions: insert (/), apply to all (α), and transpose (transp).

An object x (in O) is either an atom, a sequence $<x_1,x_2,\ldots,x_n>$, where x_i is an object, or ⊥ ("bottom" or "undefined"). The set A of atoms determines the set O of objects. The empty sequence is denoted by ϕ and is the only object which is both an atom and a sequence. The atoms T and F denote the familiar boolean values "true" and "false". An important constraint in the construction of objects is associated with ⊥: if x is a sequence with ⊥ as an element, then x=⊥. That is, the "sequence constructor" is "⊥-preserving." A proper sequence never has ⊥ as an element.

An FP system is not burdened with a large number of operations; it has exactly one: application. If f (in F) is a function and x (in O) is an object, then

$$f:x$$

is an application which denotes the object resulting from applying f to x. f is called the operator of the application and x is the operand. Functions f (in F) are bottom-preserving:

$$f: \perp = \perp \quad \text{(all f in F)}.$$

Every function in F is either primitive (i.e., supplied), defined, or a functional form.

$f: x = \perp$ has some properties which are important in talking about the mapping:

- if the computation for f:x terminates and yields the object $\perp$, f is said to be undefined at x. f terminates but has no meaningful value at x.
- when f does not terminate, it is said to be non-terminating at x.

A functional form (FF) is an expression denoting a basic function which is supplied with the model. The function depends on the functions or objects which are the parameters of the expression. As an example, for f and g in F, f·g is a functional form called the composition of f and g. The composition denotes the function such that, for arbitrary x in O,

$$(f\cdot g):x = f:(g:x).$$

Table 1 lists some FP functional forms [BACK78].

A definition in an FP system is an expression of the form

$$\underline{\text{def}}\ \ell \equiv r$$

Where the left side ℓ is an unused function symbol and the right side r is a functional form which may depend on ℓ. It means that symbol ℓ is to denote the function represented by r. A defined symbol is applied by replacing it by the right side of its definition. A definition may

Functional Form (FF)	Notation
Composition	$(f \cdot g):x \equiv f:(g:x)$
Construction	$[f_1,\ldots,f_n]:x \equiv \langle f_1:x,\ldots,f_n:x\rangle$
Condition	$(p\rightarrow f;g):x \equiv (p:x)=T\rightarrow f:x$ $(p:x)=F\rightarrow g:x;\ \perp$
Constant	$\overline{x}:y \equiv y = \perp\rightarrow\perp;x$(x an object parameter)
Insert	$/f:x \equiv x = \langle x_i\rangle \rightarrow x_1;\ x = \langle x_1,\ldots,x_n\rangle$ $(n\geq 2)$ $f:\langle x_1,/f:\langle x_2,\ldots,x_n\rangle\rangle;$
Apply to All	$\alpha f:x \equiv x = \phi \rightarrow \phi;$ $x = \langle x_1,x_2,\ldots,x_n\rangle \rightarrow$ $\langle f:x_1,\ldots,f:x_n\rangle;\ \perp$

Table 1

Some FP Functional Forms

[BACK78]

be a non-terminating function. The set D of definitions is well formed if all left sides are unique symbols. For examples of definitions, see Table 1 on page 43.

Backus presents an example of a functional program for inner product [BACK78]. This example will now be discussed.

The definition of the functional program for inner product is:

<u>DEF</u> Inner product ≡ (Insert+)·(Apply to All*)·Transpose.

In more symbolic form:

<u>DEF</u> IP≡ (/+)·(α*)· Trans.

The set FF of functional forms is determined by:

- combinations of existing (primitive) functions to form new ones;
- Composition "·";
- Insert "/";
- Apply to All "α".

Figure 11 shows IP and the steps involved as it is applied to the vector pair (<1,2,3>,<6,5,4>).

The semantics of an FP system are determined by the choice of four sets and the manner of computing functions from them. The FP system itself is determined by the four sets:

- the set of atoms A, which determine the set of objects;
- the set of primitive functions P;

- the set of functional forms FF;
- a well formed set of definitions D.

There are only four possibilities for computing f:x:

- f is a primitive function, and is computed from its description;
- f is a complex function produced using functional forms, and the description of the forms define how f is to be computed in terms of parameters and rules;
- f is defined in the set D;
- none of the above, or $f{:}x \equiv \perp$

If f does not terminate for a given rule, then $f{:}x \equiv \perp$. The definition of expansion and the Expansion Theorem stated in Appendix B will prove whether f terminates. If it does not, f will be undefined and will not produce a predictable value when applied to x.

FP systems can be viewed as programming languages, but they are very minimal in terms of conventional languages. When so viewed, f is a program, object x is the initial contents of the store, and f:x is the final contents of the store. The set D of definitions is the program library. The primitive functions and functional forms provided in the language framework are the basic statements of a specific programming language. Depending upon the choice of primitive functions and functional forms, the FP-language framework provides for a large class of languages with varying styles and capabilities. The algebra of programs associated with each is dependent upon its particular set of functional forms.

Backus states the limitations of FP systems as follows:

- a given FP system is a fixed language;
- FP systems are not history sensitive;
- input and output can be treated only in the sense that x is an input and f:x is an output;
- if the sets of primitive functions and functional forms are weak, all computable functions may not be expressible.

As an example of a major weakness of FP systems, an FP system cannot be used to compute a new program, since functions are kept distinctly separate from objects. The process of computing new functions would require the "apply" operator such that

$$\text{apply}:\langle x,y\rangle \equiv x:y$$

where x is an object on the left and a function on the right. A second major weakness with FP systems is that new functional forms cannot be defined within the system.

Lack of history sensitivity is the primary limitation. FP systems must be extended before they become practically useful; FFP and AST systems do this.

The advantages of FP systems are as follows:

- they use names only to name functions in definitions, and names can only be treated as functions that can be combined with other functions;
- they are based on reduction semantics which eliminate the need for word-at-a-time constructs which are too closely tied to machine states;

- they offer a core of primitive constructs from which higher level constructs and techniques can be naturally developed.

FFP systems are developed from the consideration of FP systems. Backus defines the primary goal of FFP systems as follows: "FFP systems develop a foundation for the algebra of programs that disposes of the theoretical issues, so that a program can use simple algebraic laws and one or two theorems from the foundations to solve problems and create proofs in the same mechanical style used to solve high school algebra problems." See Appendix B for a discussion of the algebra of programs and proofs and an example of a correctness proof.

In FP systems the set FF of functional forms is fixed. In FFP systems this restriction is lifted and new functional forms can be created. In FFP systems objects are used to represent functions; otherwise, FFP systems are very much like FP systems. In FFP systems

Apply: <x,y> = (x:y)

is a legal construct, but not in FP systems.

To end this section, we will review the definition of applicative state transition systems (AST) and use Table 1 and Figure 11 to step through the definition of a new function called "inner product", or "IP". The discussion will reveal the natural extensibility of such systems.

<u>Definition</u>: An AST system is composed of three elements:

1) an applicative subsystem, such as Backus' FFP system;
2) a state D that is the set of definitions of the applicative subsystem;
3) a set of transition rules that describe how inputs are transformed into outputs and how the state D is changed.

"Applicative" implies the application of definitions and functions (supplied and derived) to arguments to produce results. For example, some definitions in FFP are related to the basic functions, "+" and "*". The results of applying these functions are defined by the language framework as follows:

$$+:\langle x,y\rangle \rightarrow x + y$$

$$*:\langle x,y\rangle \rightarrow x*y.$$

Table 1 defines some functional forms that are supplied in the basic language framework: Composition, Construction, Apply to All, etc. These basic definitions and supplied functional forms can be combined within the basic language framework to define more complex functions, which can be used with the basic definitions and functional forms to define still more complex functions, ad infinitum. The line-by-line detail of the transition rules and states obtained by successive applications of the definitions and supplied functional forms in FFP to define a more complex function, IP, is now given. Figure 11 summarizes the discussion.

- DEF IP

  The new function (defined function) IP is defined in terms of supplied forms and definitions:

  $$\underline{\text{DEF}}\ \text{IP} \equiv (/+)\cdot(\alpha*)\cdot\text{TRANS}.$$

  This being an applicative subsystem, it is meant that the new function IP can be applied to a sequence of vectors in the system:

  $$\text{IP}:<x_1,x_2>$$

  where

  $$x_1 = <x_{11},x_{12},\ldots,x_{1n}>$$

  $$x_2 = <x_{21},x_{22},\ldots,x_{2n}>$$

  $$x_{mn} \varepsilon R \quad (m=[1,2]).$$

For the sake of example, suppose IP is to be applied to the vector pair <<1,2,3>,<6,5,4>>. The application implied by the definition is then

$$(/+)\cdot(\alpha*)\cdot\text{TRANS}:<<1,2,3>,<6,5,4>>.$$

This is the initial state of the application.

- Composition $(\cdot)$

  By the rule of Composition from Table 1, the last result is equivalent to

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

  or

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

- Transpose (TRANS)

  TRANS is not defined in Table 1. In FFP, it is defined for two sequences as follows:

  $$\text{TRANS}:<<a_1,a_2,\ldots,a_n>,<b_1,b_2,\ldots,b_n>> \equiv$$
  $$<<a_1,b_1>,<a_2,b_2>,\ldots,<a_n,b_n>>.$$

  Hence

  $$\text{TRANS}:<<1,2,3>,<6,5,4>> \rightarrow$$
  $$<<1,6>,<2,5>,<3,4>>.$$

  Substituting the expression resulting from the application of TRANS to the vector pair back into the original string derived by applying Composition, above, we get

  $$(/+):((\alpha*):(\text{TRANS}:<<1,2,3>,<6,5,4>>))$$
  $$\rightarrow (/+):((\alpha*):<<1,6>,<2,5>,<3,4>>).$$

  This latter expression defines the next state of the system, following the application of TRANS.

- Apply to All (α)

  Referring to Table 1,

  $(\alpha*):<<1,6>,<2,5>,<3,4>>$

  is equivalent to

  $<*:<1,6>,*:<2,5>,*:<3,4>>$

where "*" is applied to all members of the outer sequence. Using this result, we obtain the next state of the system as follows:

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

$(/+):<*:<1,6>,*:<2,5>,*:<3,4>>.$

- Apply (*)

In an AST system, innermost applications are always performed first. In the last expression, three innermost applications are present:

*:<1,6>,

*:<2,5>,

*:<3,4>.

"*" is applied to these as follows:

$*:<1,6> \rightarrow 1*6 = 6,$

$*:<2,5> \rightarrow 2*5 = 10,$

$*:<3,4> \rightarrow 3*4 = 12.$

Substituting, we obtain the next state of the system:

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

$(/+):<6,10,12>.$

- Insert (/)

Here, apply the functional form from Table 1 to obtain the next system state:

$(/+):<6,10,12> \rightarrow +:<6,+:<10,12>>.$

- Apply (+)

Applying the innermost application first:

$+:<10,12> \rightarrow 10+12 = 22.$

The state transition is given by

$$+:<6,+:<10,12>> \rightarrow +:<6,22>.$$

- Apply (+)

  The final application yields the final system state and the final result:

$$+:<6,22> \rightarrow 6+22 = 28.$$

Some small liberties were taken with this example, as a comparison of the state transition for the "Insert" step will show. But basically, all state transitions to the final result are shown. Notice how the set of basic definitions and supplied functional forms are combined to define more complex functions. Each defined function in the system can then be applied to arguments without using any naming conventions, except for names attached to functions. Once IP is defined as outlined, we can write

$$IP:<<1,2,3>,<6,5,4>> \rightarrow 28$$

and utilize IP to define progressively higher functions, all within the language context. The language is thus naturally extended, accordingly.

Many details of FP, FFP, and AST systems are omitted, or discussed only briefly in this section. Refer to [BACK78] for full details.

<u>DEF</u> IP	→ (/+)·(α*)·Trans:<<1,2,3>,<6,5,4>>
Composition (·)	→ (/+):((α*):(Trans:<<1,2,3>,<6,5,4>>)
Transpose	→ (/+):((α*):<<1,6>,<2,5>,<3,4>>)
Apply to All (α)	→ (/+):<*:<1,6>,*:<2,5>,*:<3,4>>
Apply (*)	→ (/+):<6,10,12>
Insert (/)	→ +:<6,+:<10,12>>
Apply (+)	→ +:<6,22>
Apply (+)	→ 28

Figure 11

Inner Product Functional Program Application

[BACK78]

2.2 The von Neumann (Control Flow) Model

This section examines the model reflected by conventional computers and programming languages, the von Neumann, or as Treleaven calls it, the control flow model [TREL79]. A model can be studied by comparing its properties with those of other models, by examining its properties in detail, and by examining its structures. This section studies the control flow model from all three of these perspectives.

Backus studies the control flow model by comparing it to others [BACK78]. He presents a theoretician's classification of computing models. The data flow model discussed in Section 2.1.1 does not fit well into this scheme, which was presented in Backus' 1977 ACM turing award lecture. However, the classification highlights some relevant properties of control flow machines. It also provides a good comparison of control flow and applicative models.

Backus presents a list of criteria to classify computing models:

1) foundations - is there a useful mathematical description of the model?
2) history sensitivity - can information be passed from one program to a successor at runtime?
3) semantics - does a program in the model use state transition semantics or reduction

semantics? If state transitions are used, are these simple or complex?

4) program clarity - are source representations clear and conceptually useful in that they embody concepts that can be used to reason about processes?

Using these criteria, he defines three classes of computing system models:

1) simple operational models;
2) applicative models;
3) control flow models.

Table 2 summarizes these classifications in chart form with an example of each.

It is difficult to fit data flow languages into Backus' scheme (the data flow line listed in Table 2 was not in Backus' original table). They seem to partially fit the class of operational models, but with much clearer programs than other members of the class. Backus believes that some data flow languages could even be considered to possess the beginnings of reduction semantics [BACK78]. Certainly, data flow languages are not ordinarily history sensitive.

The general properties of the control flow model as charted in Table 2 summarize Backus' view of this model. As these properties are studied, one should not forget that Backus has been one of the innovators of the young science of electronic computation and,

BACKUS MODEL CLASSIFICATIONS

Class	Example	Foundations	History Sensitivity	Semantics	Program Clarity
Operational	Turing Machines	Concise Useful	Sensitive	State Transitions Simple States	Unclear Not Conceptually Useful
Applicative	Functional Programming	Concise Useful	Not Sensitive	Reduction Semantics (No States)	Clear Conceptually Useful
Control Flow (von Neumann)	Conventional Computers and Programming Languages	Complex Bulky Not Useful	Sensitive	State Transitions Complex States	Clear Not Conceptually Useful
Data Flow	Figure 10	Concise Useful	Not Sensitive	(Beginnings of) Reduction Semantics	Clear

Table 2

A Chart Illustrating Backus' Classification Scheme

[BACK78]

due to his role as an original developer of the FORTRAN programming language, one of those most responsible for the current primary position of the control flow model in practice.

The foundations of the control flow model are judged to be complex, bulky, and not useful. Backus notes the lack of a satisfactory mathematical description of the model. He feels it to be so complex and bulky that its description has scant mathematical value.

Programs in the control flow model are history sensitive. That is, one program can pass information to another that can affect the behavior of the latter. This may well be at once a primary strong point and yet a concurrency-limiting property of the model.

```
Z = X*Y-4*A*(X+Y)

Move B to A
```

Figure 12

Typical Control Flow Assignment Statements

The semantics of control flow programs involve complex machine state transitions. Observe the typical control flow assignment statement involving a moderately complex arithmetic expression as shown in Figure 12. Some idea of state-transition complexity can be gained by "mentally-executing" this statement. If this is done, a rapid series of memory fetches of literal values and values associated with named variables are "seen" passing from memory to the arithmetic-logic unit for arithmetic combination as the arithmetic expression is evaluated. When this sequence of operations is complete, the final computed value passes from the ALU to memory (i.e., it is "stored" in a location associated with the named variable "Z").

Each passage of a value between memory and the ALU defines a state transition, and each combination of sets of values in ALU and memory cells defines a state of the control flow machine. Even the simplest assignment, such as the simple COBOL "MOVE" of Figure 12, involves multiple state transitions.

Consider state transitions in the functional model discussed in section 2.1.2 in contrast to this situation. State transition rules in the functional model are entirely defined within the model and depend only upon the manner in which inputs are transformed into outputs and the subsequent change in the state D representing the set of definitions of the underlying

FFP system. Thus, a state transition in this system is not related to any complex rules involving machine operations on any physical entity such as global memory.

Finally, the control flow program clarity property is deemed "clear but not conceptually useful" by Table 2. Generally, programs of the model do provide clear expressions of a process or computation, but they do not provide concepts that help people to reason easily about processes. One need only reflect on the excessive requirements of the simplest program proof to understand that some inherent properties of control flow programs seem to make formal reasoning about them very difficult. Reasons for this will become clear as we consider the structures and properties in greater detail.

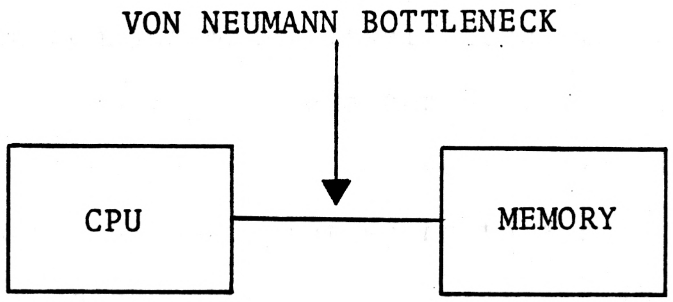


Figure 13

Basic Structure of a Control Flow Computer

[BACK78]

In the ensuing discussion, it can be argued that we are discussing implementation and not model, since we must speak of the control flow model in terms of registers, global memory accesses, etc. Indeed, this seems to be a failing of our current views of von Neumann computing, where many aspects of model and implementation have become almost interchangeable. Nevertheless, program counters, hidden registers and register transfers involving state transitions, CPU-to-memory paths, variable naming conventions equated with memory mappings, etc., are all at this point in history intimately associated with the von Neumann "model."

Backus does not consider one obvious alternative to replacing the von Neumann model with another that has a better theoretical structure: the alternative of separating the von Neumann model itself from its many historical implementations and strengthening its theoretical structure. His purpose does not seem to fit that particular approach. Without pretending to assume anything about what he thinks about this matter, it is possible he believes the alternative to be not particularly viable.

Perhaps the alternative approach could be the subject of other reports. In this report we must consider the concept of the "von Neumann model" as it now exists in theory and practice. Certainly, a definite strength of this view is the history sensitivity property, which makes commercial and business computing pragmatic. One contrast between the von Neumann model and later models does seem to arise simply because the conceptual environment in which they have arisen and evolved is much different than that which spawned von Neumann computing.

Conceptually, a von Neumann, or control flow computer is composed of the three parts illustrated in Figure 13:

1) central processing unit (CPU);
2) memory store;
3) connecting tube.

The connecting tube can transmit a single computer word between the CPU and the memory, or vice versa. One memory cell, for example, can only be moved to another by traversing the tube from the memory to the CPU and back again. The CPU contains central storage cells, called "registers":

- central registers available to the programmer;
- central registers "hidden" from the programmer;
- special register(s) called the "memory address register(s)" (MAR);
- special register called the "program counter" (PC).

Only one value at a time can flow on the connecting tube. A machine state transition is initiated by placing a word on the tube for transmission to or from the store. A machine state is represented by each successive set of values of cells in memory and in the CPU registers during operation.

[BACK78] outlines the task of a program in the control flow environment: It must change the contents of memory in some major way. This task can only be done by shipping one word at a time through the connecting tube, or von Neumann bottleneck. Variable names are always associated with memory locations, and much of the activity on this avenue is in addition to the main task the program is designed to accomplish and is related to manipulating and computing names, etc. The PC and MAR registers, for example, simply provide data names for instructions themselves and their operands, respectively, during operation. Each instruction must be fetched (by name) from memory to the CPU (across the bottleneck) to begin its execution cycle. Each of its operand names must then be fetched into the CPU using the same mechanism. "Programming" a control flow machine consists primarily of managing the enormous flow of words across the connecting tube, and much of that flow concerns not only data relevant to the problem, but also data names in the form of memory addresses.

Backus believes the connecting tube to be both a literal and an intellectual bottleneck:

- literal bottleneck for problem traffic;
- intellectual bottleneck that has kept computer engineers and scientists tied to word-at-a-time thinking.

The intellectual bottleneck has blocked designers from thinking in terms of the larger, conceptual units of the problem to be solved.

All control flow computers of this sort possess the CPU register called the "PC", above. Machines based on this model tend to be very serialized, step-at-a-time mechanisms admitting no real concepts of concurrent operation. This property of control flow machines will become more obvious when the control flow language structures are discussed in the next section.

2.2.1 The Structure of Conventional Programming Languages

The control flow model existed first in hardware and was programmed in machine language. Conventional symbolic assembly languages evolved as aids to the machine programmer, and high-level languages were developed solely for the same reason at a point in time following the development of assembly language concepts. In the case of the control flow model, the hardware development drove the language interface, as symbolic languages were viewed strictly as man-machine communication aids.

Treleaven states the result of the evolution very well in [TREL79]: "Conventional programming languages, which are often called 'high level', display a model of computation that is, in some important respects, actually at a very low level, not far removed from the von Neumann machine. These languages are based on the processor/memory model of program execution in which a processor performs operations on values stored in a memory (a sharable and modifiable resource)."

Treleaven isolates the basic structure of all programs based on the control flow model. Whether the language is PASCAL, FORTRAN, BASIC, COBOL, ALGOL, PL/1, etc., a program for this model has three basic parts:

1) a set of sharable memory cells called variables;
2) a set of data instructions that modify variables;
3) a set of control instructions that determine the order of instruction execution.

In a program, "normal" flow of control between the execution of control instructions is determined by assuming that each non-control instruction execution sequentially follows that instruction execution for the instruction stored immediately preceding it (by memory location). This is an obvious result of the primary hardware control register mechanism, the program counter (PC). "Programming" then consists in specifying the various sequences of instruction execution to solve a problem. Data instructions involved with solving the problem are intermingled among control instructions.

[TREL79] defines variables as follows: "variables" are named memory cells that serve two roles within a program:

- provide a technique to communicate partial results within instruction executions;
- provide semipermanent data storage, which allows multiple references to (named) data values.

The control flow program structure has two properties which will be important later:

- the flow of control mechanism results in program executions which are explicitly time sequential by instruction with sequence specified by the programmer;
- the variable/memory location mapping is at once a strength and a weakness of the model.

The mapping is a strength in that storage can be retained and reused. It's a weakness in that it causes great implementation overhead for manipulating names and allows a phenomenon called "side effects," which will be discussed later.

2.2.2 Synchronization Primitives

[TREL79] reviews the problem of representing concurrency in the conventional control flow model. Treleaven notes that an important requirement for a new computing model is that it support concurrency at a low (preferably hardware) level. This requirement is basically incompatible with the control flow model:

- overspecification of sequence in the model;
- concept of a variable as a shared memory cell.

These two properties demand the usage of explicit control and synchronization statements in the programs of the model.

Synchronization primitives are of two types:

- concurrency initiating statements, to activate parallel instruction streams (processes);
- synchronization statements, to synchronize multiple process terminations and resume processing in a resultant stream.

They are necessary because multiple instruction streams may modify shared memory cells, and the effects of such modifications are time-dependent and must be controlled. Some examples of concurrency initiating statements are "CALL", "FORK", "ATTACH", etc. Some examples of synchronization statements are "WAIT", "JOIN". Figure 14 shows an example of FORK and JOIN in [TREL79].

```
FOR J:= PIVOT_ROW + 1 TO NO_COLUMNS DO
     "ACTIVATE PARALLEL INSTRUCTION STREAM:"
     FORK PARA;
N := NO_COLUMNS - PIVOT_ROW;

" WHEN N = NO_COLUMNS - PIVOT_ROW, THE ABOVE
     SPAWNS N-1 INDEPENDENT PROCESSES, EACH
     WITH DIFFERENT VALUE OF J."

PARA:  FOR K := COLUMN TO NO_COLUMNS DO
     A[J,K] := A[J,K] - A[J, COLUMN]
     *A[PIVOT_ROW,K]/A[PIVOT_ROW, COLUMN];
     JOIN N;
```

Figure 14
Example of Fork and Join Synchronization
Primitives in [TREL79]

[TREL79] lists disadvantages of synchronization primitives:

- the programmer's task is further complicated by the need to encode extra information;
- extra information detracts from program readability;
- the present style of architecture cannot utilize the extra parallelism well unless each concurrent element is represented as a process.

The last point stresses the fact that parallelism in the control flow model is not fine grain.

2.2.3 Monitors

The development of the monitor concept was one of the more interesting efforts originating in control flow computing. Three eminent computer scientists, E. W. Dijkstra; C.A.R. Hoare; and per Brinch Hansen, contributed in some measure to this effort. Two of these men published numerous papers and books dealing with concurrency in control flow computing ([DIJ168], [DIJ268], [DIJK71], [HANS77], [HANS79]).

Hoare's chief contribution is noted in [HANS79]. He noted that concurrent operations have predictable effects only if statements within each of them operate on different variables; otherwise, effects of concurrent operations will be time dependent. This would prove to be a key observation in the development of the monitor concept.

The truly creative aspect of the monitor concept stemmed from the way in which Dijkstra and Hansen invented language and compiler constructs to solve concurrency problems within the control flow model. Finally, someone realized the advantages of approaching the concurrency problem from language and data structure viewpoints. Dijkstra [DIJI68] invented a "concurrent statement" to initiate concurrent processes from a high-level language and suggested combining all operations on a shared data structure into one program module. Hansen proposed a language notation for this "monitor" concept and developed a compiler to support it [HANS77]. The idea to utilize the compiler in this way had novel goals which were beyond simply improving the man-machine communication interface:

- replace hardware protection mechanisms by compilation checks;
- improve program testability;
- solve the problem of controlled access to shared variables by providing an easy-to-use modular language interface to handle synchronization and racing conditions;
- allow the compiler to verify many of the shared memory accesses, allowing execution checks to be omitted.

The last goal was done both in the interests of program efficiency and the desire to prevent (rather than simply avoid) problems.

Despite the amount of work done by these men, Hansen states in [HANS79] that the theoretical understanding of concurrency is still in its infancy.

2.3 Comparing Highly Concurrent and Control Flow Models

Before proceeding with the functional comparison, we need to briefly review the new cost/performance goals introduced by LSI/VLSI technology. These goals in themselves present a marked contrast with those of control flow computing.

Why do computer scientists and engineers consider the property of implicitly concurrent operation at the hardware level to be so important? A large part of the answer seems to be that LSI/VLSI implementations will radically alter control flow concepts of cost/performance. Implementing highly concurrent operation at the hardware level introduces the potential for realizing a performance increase over "equivalent" control flow implementations of huge magnitudes [GOST80].

Dennis [DENN80] lists three goals which he feels future computer architects must meet in the next supercomputer generation:

1) extremely high performance at acceptable cost;
2) something approaching the full potential of LSI/VLSI technology must be exploited;
3) architectures must admit concurrency without requiring explicit programming techniques.

In order to compete successfully in the next generation, he believes new LSI/VLSI implementations must be capable of doing such things as executing floating point instructions on the order of magnitude of billion(s) per second. Control flow models cannot realistically approach this goal with reasonable cost expectations.

With such cost/performance goals in mind, how do highly concurrent and control flow models currently compare? Section 2.1 examined two implicitly, highly concurrent models: the data flow and functional ones. Section 2.2 examined structures and properties of the implicitly sequential control flow model. The present structure of concurrent models differs in some key ways from the structure represented by control flow models. The remainder of this section discusses a few of the most important properties which differ appreciably between the models.

Probably the most important way in which the control flow model differs is in the philosophy and evolution of the model itself. Both concurrent models have stronger abstract structures than does the control flow model. These theoretical structures distinctly preceded any implementations. This level of abstraction clearly allows the abstract models to stand distinct and separate from their various implementations. The more pragmatic evolution of von Neumann computing does not so clearly allow this differentiation between model and implementation. In fact, it's very difficult to separate a distinct theoretical structure of control flow computing from its implementations.

In the highly concurrent models, concurrent operation is the assumption at the hardware design levels. The models are structured to implicitly account for the presence of multiple processing elements, each of which can execute when all processor inputs and required resources are available. Adding additional processors will often raise the level of concurrency with no need for external programming support. Conversely, the control flow model assumes sequential, statement-by-statement operation in which external programming support is necessary in order to support increased processor levels.

In highly concurrent models, only the availability of operands and resources determines a processor's availability for execution. At the programming level, the concept of flow of control between statements is not a determinant of expression execution. For example, a sequence such as

```
X = 3
A = 5
B = 4
C = A*B+6
D = C+4*B
E = X+17
```

is not bound at execution time by statement boundaries. Computation of E can proceed in parallel with that for C, and the value for D may be partially computed by the time the value for C is determined. The value for E may be available before either of the values for C or D are computed. In the control flow model, assumptions governing sequential execution of statements rigidly determine the sequence in which values for each variable will be available. The further need for control statements to transfer control within sequences of statements in the control flow model is not needed in highly concurrent models, although current understanding of structured techniques within the control flow model reveals that this need has been highly exaggerated in the past.

Control flow models have the property of history sensitivity, or the ability to store and retrieve many data values at will during program execution. Data flow and functional models do not normally have this property (without extending the models). In the control flow model, once a value for a variable named A is defined, it is available in subsequent computations until redefined through a new assignment. Data flow programs require the extremely restrictive single-assignment rule, since they cannot "remember" stored values in this way. Functional programs do not even associate names with ordinary values, except at the highest functional level. The lack of history sensitivity is probably the largest handicap of highly concurrent models as they now exist.

As an example of the power of history sensitivity, imagine a pure data flow or functional program trying to compute a large set of one thousand homogeneous values, which a control flow program could easily store in a memory array. Once stored in a control flow array, the values are individually referencable and retrievable until modified by program assignment. Because of the single assignment rule and the equating of names to arcs, it is very difficult to deal with such arrays in data flow. Research is being conducted in this area [DENN79]. A functional program does very well when using multiple processors to compute a single value, which is just the reverse of the control flow case. Much research is still needed to introduce satisfactory properties of history sensitivity ([BACK78], [MAGO80]).

Highly concurrent models eliminate global mappings of variable names to memory locations. This eliminates complex, hardware-bound concepts of state transitions as contents of memory locations are modified, and it also minimizes such things as subroutine side effects. Thus, simpler, non-hardware associated concepts of state transitions are possible, but only after the important property of history sensitivity has been compromised.

Thus, in terms of comparable execution timings, it's possible to attain very high performance gains over traditional control flow implementations at acceptable costs with LSI/VLSI technology by utilizing highly concurrent models. However, performance includes something more than simple execution timings on scalar structures: it also includes versatility, as exemplified by the history sensitivity property utilized in control flow computing. It's hard to imagine anyone referring to a highly concurrent, business-oriented system with no history sensitivity as being "high performance."

Chapter 3

Implementations of Computing Models

Section 3.1 will introduce data flow models by considering Rumbaugh's [RUMB77] conceptual model from an architectural viewpoint. Dennis' abstract implementation will then be considered [DENN80]. Most of the important concepts of data flow computing are expressed in these works; very good additional readings can be found in ([GOST80], [GUR680], [GUR780], [KELL80], and [TREL79]).

Section 3.2 will consider a functional implementation from [MAGO80], which is based on Backus' work [BACK78]. Section 3.3 will consider parallel taxonomies, and how these will have to be extended for the highly concurrent model.

3.1 Data Flow Models

Proponents of data flow architectures believe data flow models will one day displace control flow models as more important structures. They note common properties of data flow models which seem stronger than their control flow counterparts. Some examples of data flow implementations are discussed in this section.

Rumbaugh [RUMB77] defines a data flow multiprocessor which is defined in terms of a set of activation processors.

Activation processors perform a single invocation of a small data flow procedure held in its local memory. The terms _procedure activation_ and _procedure invocation_ are used interchangeably and refer to the moment when operands arrive and execution of the local procedure is initiated. Each activation processor is defined in terms of a pipeline of other logical units, so concurrency is obtained among and within activation processors.

Data are stored and processed within the system in tree structures. The results are value oriented; identifying names or addresses are not associated with each value. Rather, the data are grouped functionally, according to operation, into _result packets_. Hardware units called _structure controllers_ and _structure memory_ process and store the data structures.

Rumbaugh's model is conceptual: no implementation currently exists. He intended it to be considered as a standalone multiprocessor, but it could be imbedded in a larger system (e.g., a large control flow processor).

Rumbaugh's conceptual model consists of a number of _major modules_ (i.e., hardware units at the same level of implementation as activation processors, which can operate concurrently). The major modules are:

- Activation Processors
  each holds and executes a single procedure activation;

- Scheduler
  coordinates and assigns activations to processors;
- Structure Memory
  holds data structures too large to fit in activation processors;
- Structure Controllers
  operate on structures for the processors;
- Program Memory
  holds procedures which can be called;
- Swap Memory
  holds procedure activations which are temporarily dormant;
- Swap Network
  transfers procedure activations between Swap and Program Memories and Activation Processors;
- Peripheral Processors
  connect the machine to the outside world.

The major modules are further subdivided into the basic modules, where a basic module is an asynchronous finite state machine which executes concurrently with and independently of all other modules. These are pipelined within the major modules. A similarity among all data flow hardware designs reviewed is the fundamental pipeline structure used to interconnect the various processors of the machine. An arbitrary major module (e.g., an Activation Processor) is broken down into a fairly large collection of basic modules (such as Add, Multiply, Copy, Decoder, etc., in an Activation Processor), which are independent, pipelined units. Pipelining at this basic level assures a very high degree of concurrency. Figure 15 is Rumbaugh's conceptual model, and Figure 16 is an example of the pipelined basic modules connected to form a major module (an Activation Processor).

Rumbaugh feels the advantages of such a structure are related to simple, independent construction of the basic modules. Simplicity enables him to prove that the machine correctly implements the associated data flow language. Because the basic modules are simple, finite state, asynchronous, without side effects and interdependencies, and guaranteed by proof [RUMB75] to execute well-formed data flow programs, they can be verified to do so without processor-memory interdependencies, deadlocks, and race conditions.

Rumbaugh's conceptual model is an excellent reference for gaining a high-level view of data flow structures. However, Dennis' [DENN80] tutorial report is better for a novice to data flow computing studying the detailed concepts for the first time. Hence, Dennis' paper will be utilized as a base reference to present the basic details of the data flow model. We will now terminate consideration of Rumbaugh's conceptual model (Figures 15 and 16) and study Dennis' data flow machine (Figure 17).

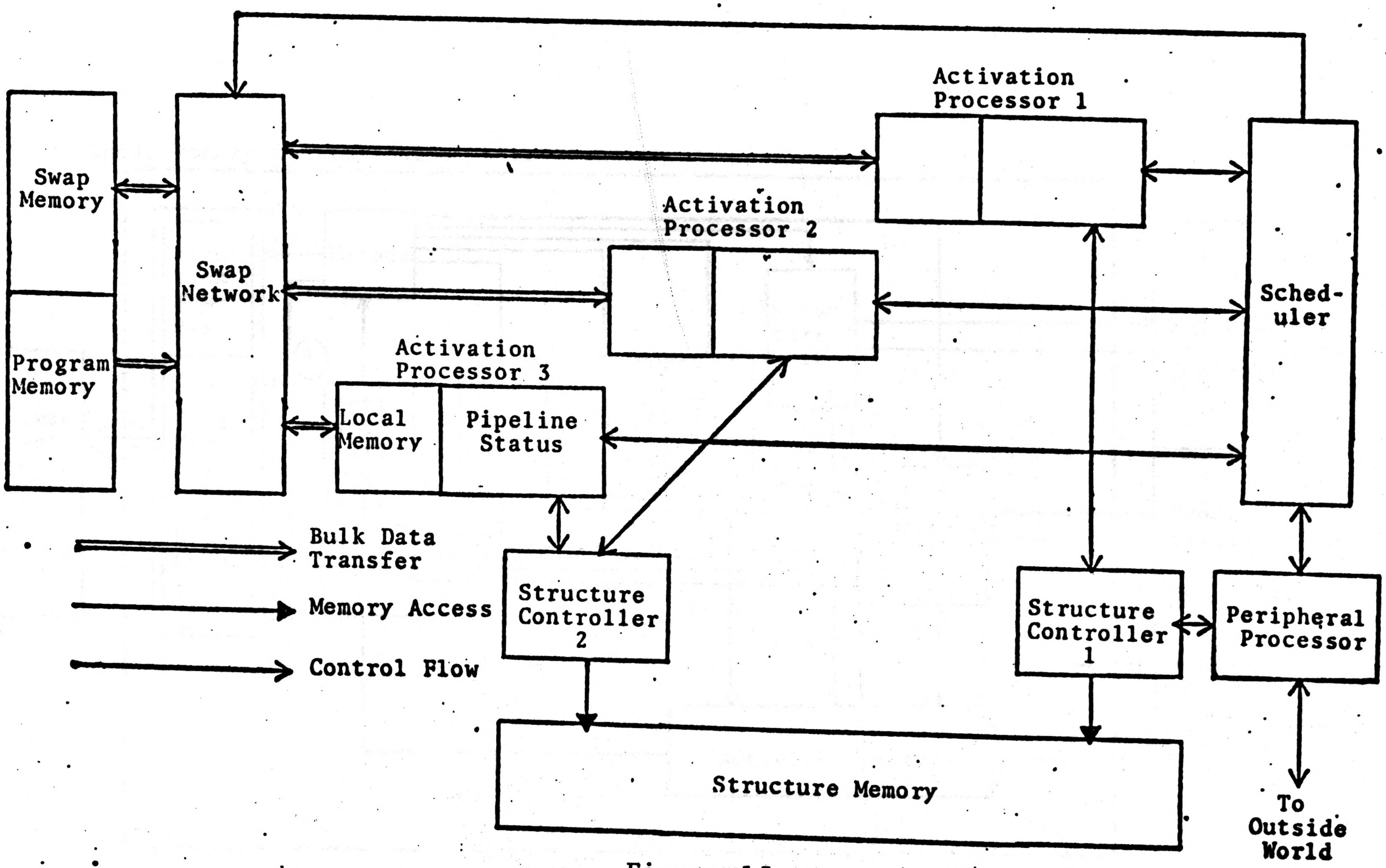


Figure 15

Rumbaugh's Conceptual Data Flow Model [RUMB77]

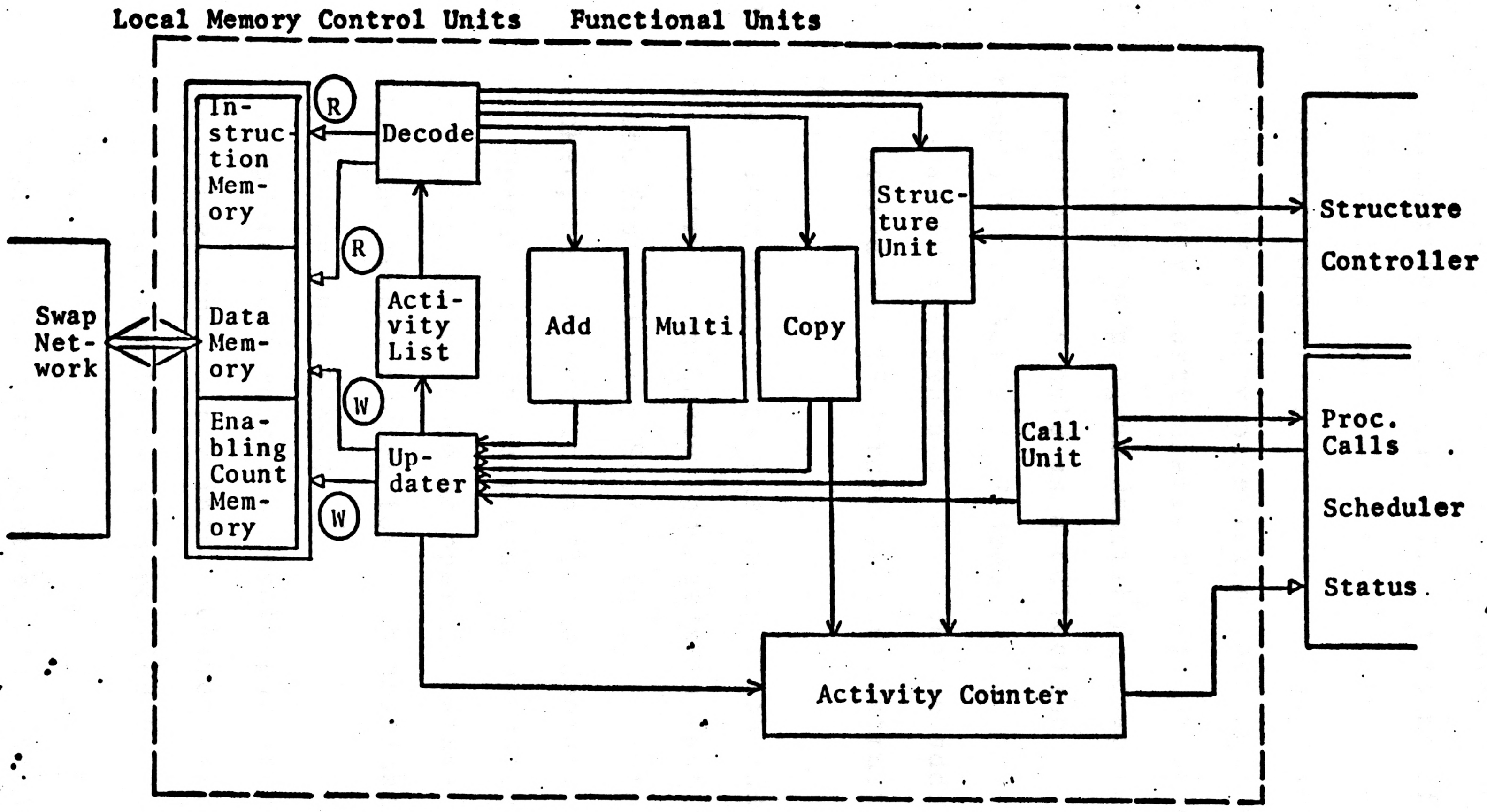


Figure 16

Rumbaugh's Activation Processor [RUMB77]

Dennis' basic instruction execution mechanism is defined as a series of six steps. The structure upon which the instruction cycle operates is a circular pipeline architecture as illustrated in Figure 17. In terms of the structure, the highest level of concurrency is obtained from the circular pipeline connecting the units. Lower levels of concurrency are obtained by pipelining each unit within the structure separately, particularly the operation units.

1) the data flow program describing computation to be performed is held as a collection of activity templates in Activity Store;
2) each activity template has a unique address which is entered in FIFO order in the Instruction Queue Unit;
3) the Fetch Unit takes the instruction address from the Instruction Queue and reads the activity template from Activity Store, forms it into an operation packet, and passes it on to the Operation Unit;
4) the Operation Unit performs the operation specified by the operation code on operand values and generates one result packet for each destination field of the operand packet;
5) the Update Unit receives result packets and enters the values they carry into receiver operand fields of activity templates as specified by destination fields;
6) the Update Unit tests whether all operand and

acknowledge[1] packets required to activate destination instructions have been received; if so, it enters the instruction address into the Instruction Queue.

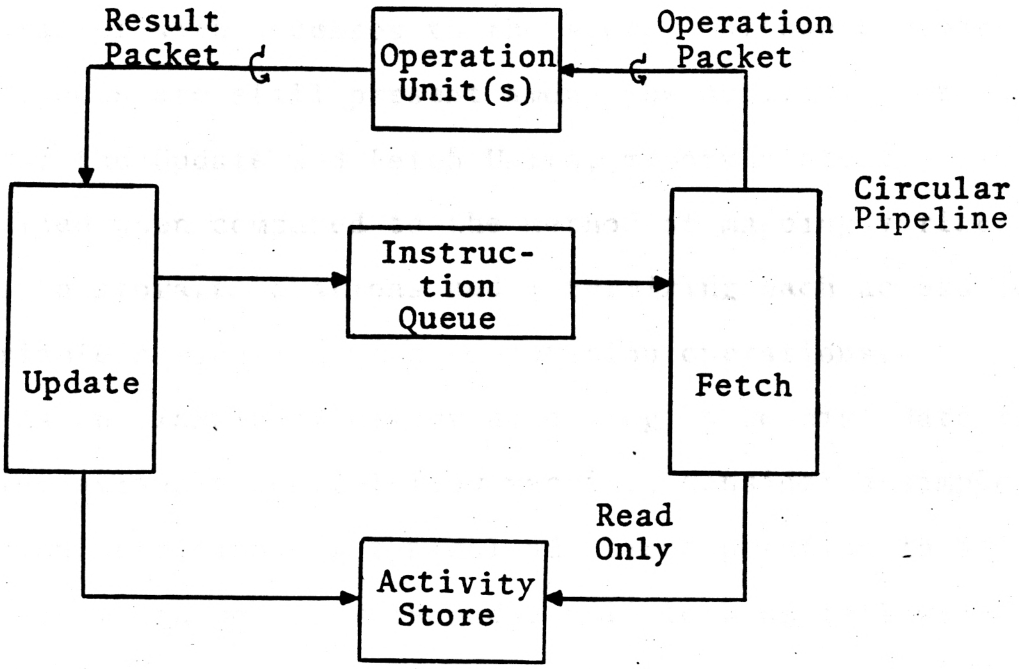


Figure 17

Dennis' Instruction Execution Mechanism

[DENN80]

[1]Acknowledge signals and packets are discussed in [DENN80]. They are required by the need to verify that output arcs of an actor are free of tokens before firing.

The accesses to Activity Store are for the purposes of retrieving and updating activity templates, each of which holds all information required for a given computation. Each activity template is addressable as a unit by address from the Instruction Queue Unit. All computation is performed within the operation units, separately and in parallel with accesses to the store. Although memory bottlenecks are still present among the Activity Store and each of the Update and Fetch Units, memory contention is minimized when compared to the method of mapping variable names to storage locations and intermixing each access to a variable's location with computation operations.

As an example of memory accessing in Dennis' data flow machine versus a control flow machine, consider a simple addition operation. A typical addition operation in a high level control flow language will look as follows:

$$A := B + C$$

The following memory accesses will be required to calculate, store, and use this value across the connecting tube between the store and the CPU:

- the address of B;
- the value of B;
- the address of C;
- the value of C;
- the address of A;
- the value of A (the result).

Additionally, each succeeding operation that requires the result will have to access A (two memory accesses per reference). This totals 6+2*n memory references to perform the operation and supply the result value to n succeeding operations. In Dennis' architecture, the Fetch Unit will access the Activity Store to obtain a packet of information including the operation code, the B-value, the C-value, and a field to contain the result of the operation. The add operation unit will perform the operation without further access to memory. The Update Unit will update destination fields in other packets which are waiting for the result of this operation in Activity Store. This totals one access plus a minimum of n accesses, depending upon the way in which result packets are "addressed". Since destination fields are carried by the original packet, a total of 1+n memory references should be accurate. The factor of 2 in the control flow value representing number of references is a direct result of mapping names in that model to memory locations. The linear factor in the corresponding data flow value is due to the value-oriented approach of that model.

Concurrency can be obtained from this structure in many ways. Basically, however, the number of entries in the Instruction Queue measures the degree of concurrency in the program. The basic instruction execution mechanism can exploit concurrency immediately, since an entry may be read from the Instruction Queue without waiting, just after the Fetch Unit has sent an operation packet to the Operation Unit. There is no need to wait until the processing for the instruction previously fetched is complete. A continuous flow of operation packets may flow from fetch unit to operation unit as long as entries remain in the Instruction Queue.

Concurrency is also obtained from the circular pipeline construction of the system. All its units may process concurrently. Here, the degree of concurrency obtainable is limited only by the degree of pipelining within each unit.

Additional concurrency may be obtained by splitting units in the ring into multiple units which can operate concurrently. Eventually, the level of concurrency will be limited by the capacity of data paths between units of the ring.

Finally, the data flow processor itself may be joined in a data flow multiprocessor system with others of its kind. This increases concurrency enormously, [DENN80] discusses a data flow multiprocessor and a supporting communication network system.

3.2 A Functional Implementation

Dennis' data flow structure in Section 3.1 utilized a circular pipeline, or ring communication network structure. Though very popular, rings have the disadvantages that delay grows linearly with size, and capacity is bounded in a fixed way [DENN80].

Magó's functional structure discussed in this section will be a tree-structured network for communication among processors [MAGO80]. Specifically, a binary tree structure will be described. Advantages of such a tree are that the worst-case distance between leaves grows only as $2\log_2 N$, and many pairs of nodes are connected by relatively short data paths. A disadvantage is that traffic density at the root node may be too high [DENN80].

A further advantage of tree-structured networks is ease of extensibility; new processor elements may be absorbed and utilized rather easily into an existing structure. This is discussed in Magó's paper.

For Magó's architecture, the programming language actually preceded and inspired the architecture. The architecture was devised to execute Backus' formal Functional Programming (FFP) language [BACK78]. Here is a case where the language design drove the architectural design.

Backus [BACK78] blames the lack of programming power in conventional systems on current programming languages. He suggests an alternative: functional programming. Magó [MAGO80] notes two reasons for the current difficulty in building high performance computers:

- dominance of von Neumann languages and lack of computing models;

- practice of designing hardware and software separately.

Magó then proposes an approach to the design of a high performance computer by basing it on the following properties:

- highly concurrent, cellular architecture;
- Formal Functional Programming Language (FFP) of Backus [BACK78];
- direct FFP-execution hardware.

By directly executing the FFP source language, complex software, such as compilers and schedulers, are eliminated. These can be exceedingly complex for parallel computers, since the scheduling of parallel resources is a very complex task at the relatively high software level. In the functional implementation, the responsibility for scheduling concurrent operation is designed into the lowest hardware levels, and the responsibility for resource scheduling at software levels is eliminated. Eliminating these scheduler and compiler resource responsibilities from the software level and designing them into the hardware enormously increases chances for a successful highly concurrent operation [MAGO80]. In order to maximize concurrency, it should be implicit to the model and should be designed in the hardware level.

Magó's machine [MAGO80] is a binary tree of cells (Figure 18). Leaf cells are called L cells (Leaf, or Linear), and collectively are called the L array. Non-leaf cells are called T cells (Tree). All L cells are identical structures. All T cells are identical except for those connected as I/O ports.

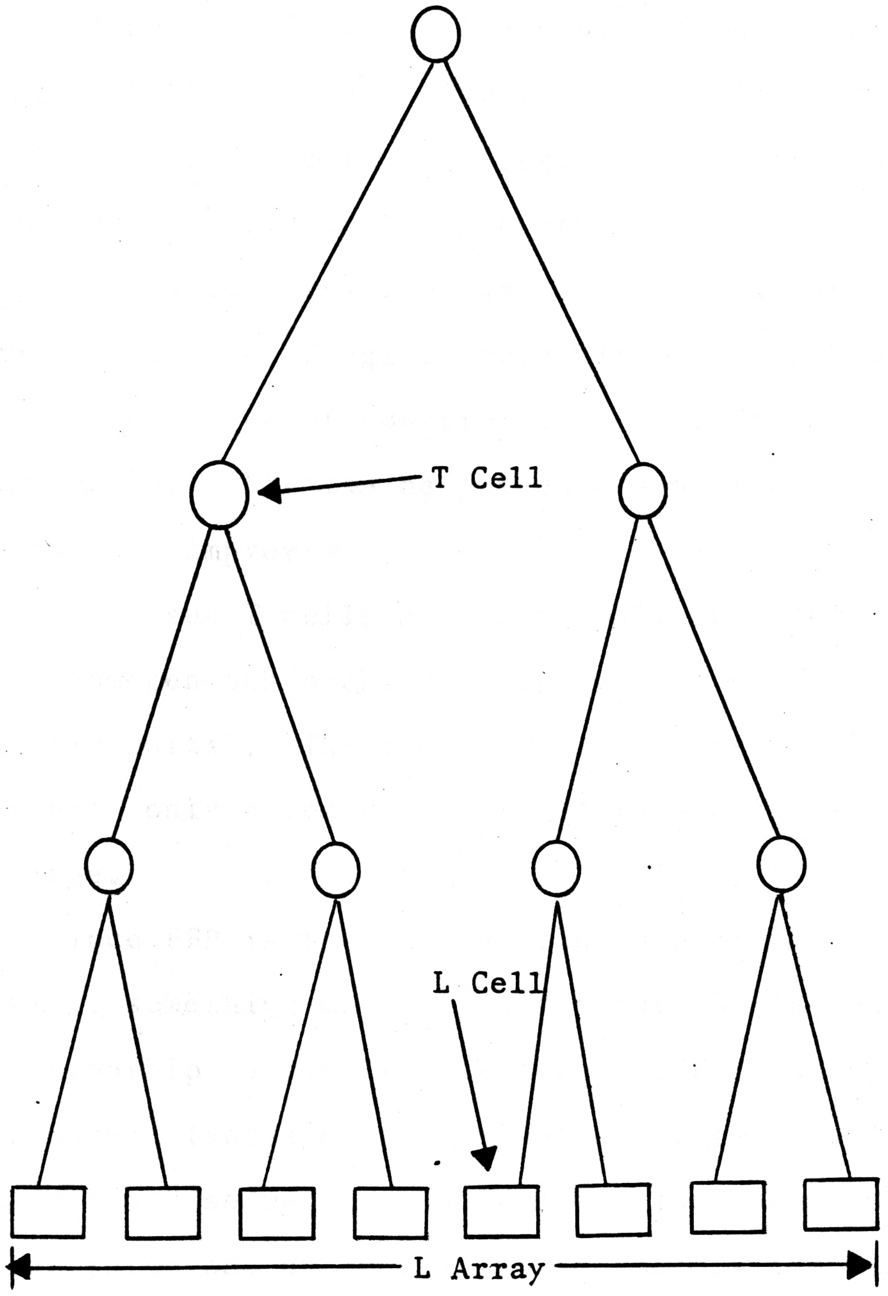


Figure 18

Magó's Binary Tree Structure

[MAGO80]

The number of cells is a linear function of the length of the L array. There is approximately one T cell for each L cell. The regularity of the construction reduces hardware complexity and cost. The network can be expanded easily by adding new cells and enlarging the L array accordingly. Advancing VLSI technology favors this type of construction: larger and larger subtrees of cells can be put on a single chip as the technology improves.

The L and T cells are kept small and simple. L cells have homogeneous architectures, and so do T cells (except for I/O ports). The architectural needs of each cell are meager: only a few dozen registers are required for local storage.

Since FFP is the machine language of the conglomerate device, something must be said about the language and its relationship to the architecture. FFP is an applicative language: language expressions consist of nested applications and sequences. An application is composed of an operator and an operand which specify computations to be performed. For example,

```
<5,(*:<7,3>)>
```

is a sequence consisting of two elements:

- number 5;
- nested application: *:<7,3>.

The nested application is called the innermost application, since no other applications are contained within it. the application consists of:

- an operator, * ;
- a sequence, <7,3>.

In FFP, innermost applications are eligible for execution, and the execution of an innermost application is called a reduction, or a reducible application (RA). To execute an application, it is evaluated according to its operator and operands and replaced with a result expression. In the example:

*:<7,3> is replaced by 7*3=21.

Thus, the original sequence is reduced from <5,(*:<7,3>)> to <5,21>.

FFP languages possess an important property which enhances their ability to incorporate concurrency: the final result of computation is independent of the order in which innermost applications are executed. This is called the Church-Rosser Property.

An FFP program is a linear string of symbols which are mapped onto the L array from left to right. One symbol is assigned to each L cell, and empty cells can be interspersed. Expression separators (parentheses, etc.) can be omitted, since that function is satisfied by cell boundaries, integers are stored in place of closing application and sequence brackets to indicate nesting levels of symbols.

Since the FFP program is mapped onto the L array only, the L array serves as a store (without address registers, etc.). The T cells serve as a set of processing elements. Their rules are somewhat interchangeable, since L cells have processing capabilities, and T cells may hold symbols temporarily during processing. Several consequences follow from the capability to place at most one FFP symbol in an L cell:

- L and T cells may be small and simple;
- a network of practical size comprises many cells;
- sequences and applications are held by collections of cells, and reducing an RA involves the cooperation of several cells;
- parallelism can be exploited at both the FFP language level (among different RA's), and below the language level at the level of language primitives (such as operations).

Appendix A discusses the execution mechanisms and shows an example of a mapping into the L and T cells. The partitioning of the machine for RA's is illustrated. An example of the apply-to-all (AA) operator is also shown.

Placing FFP symbols together in their natural order groups all symbols in the L array into advantageous leaves on binary subtrees for processing. Operator, operand, and any two different elements of a sequence occupy disjoint segments of the L array. This distribution allows the processors to locate subexpressions easily, without the need for complicated addressing schemes or software descriptors. Since two different RA's occupy disjoint segments of the L array, independent execution of each is enhanced.

How is the concept of "machine state" related to this structure? Certainly, the concept of states cannot be applied at the language statement and expression level as it is with control flow languages. Cells are coordinated by granting each cell finite-state control of its own operations. The state of the cell is then determined by its communication events with its immediate neighbors. Since the state of the cell changes whenever its parent or both its children change states, the entire network is controlled by state changes which sweep up and down the tree structure based upon problem events during execution. A cell change-of-state is represented by the completion of its operation (e.g., add, multiply, etc.), and the subsequent signal sent to its parent (or child) that the result is ready. As one scans down the tree, the L cells will seem quite out of step with each other. But as the operations progress, upsweeps in the tree will introduce higher and higher levels of synchronization. When the last change reaches the root node, the entire network is fully synchronized. However, even when fully synchronized, individual cells could be in any of a number of possible states, and a global state of the network cannot be defined.

Magó discusses many additional properties of the structure. He includes one example which depends on a specific FFP microprogram for the "Apply to All" operator. Since this example includes both a microprogram and some examples of copying operations to bring operators and operands together, which will increase concurrency, it seems worthwhile to include it as Appendix A, along with the description of some details of operation in which the example is embedded. No other examples (nor definitions) of microprogram operation were available. The remainder of this section merely summarizes the detail of Appendix A.

Some of the additional properties discussed by Magó are:

- understandability in terms of the FFP language alone, without reference to machine detail;
- tradeoffs of simpler operation versus faster execution speeds when electing whether to divide the machine operation into well-defined cycles (see Appendix A and Magó's paper);
- comparative ease of debugging FFP programs;
- dynamic repartitioning of the network of T cells;

- microprogramming language (Appendix A);
- communication during processing;
- resource and storage management;
- fundamental issues of program efficiency;
- problems remaining before the structure could be implemented into a full, history-sensitive computing system.

Only a few highlights of these areas are discussed in the remainder of this section. Magó's paper [MAGO80] should be referenced for complete details.

Dynamically repartitioning the network for optimum usage of L and T cells is an interesting problem. Yet, Magó points out that the entire process of repartitioning is unnecessary and in general may not be worth the effort. At the initiation of execution, each RA has a subtree of the entire network automatically allocated to it. The subtree is defined by the L subarray containing the RA. Efforts to repartition the T part of the network to obtain "optimal" subtrees can never diminish the initial allocation of resources available to RA's, and a set of subtrees with "optimal" properties for problem solution should exist. However, actually performing this repartitioning does not seem possible at the present time. Moreover, the natural partitioning process of the entire network (itself a "tree machine") into a set of disjoint subtrees is easily accomplished:

- it is automatic: it's completely determined by the FFP expression and its position in the L array;
- it is dynamic: it's done once in each machine cycle and marks the changes in FFP program text;
- it is fast: only one upsweep and one downsweep is required.

The mode of communication among L cells during processing is based on the tree structure. Information "climbs" the tree limbs to the roots of the subtrees of the RA's. From these roots information is broadcast to other L cells of each RA. L cells need only specify what to send and what must be received; the rest of the communication process is automatic.

Communication among L cells is also related to the logarithmic distance properties of the tree, since communication eventually is accomplished at the root nodes of RA's. Queuing occurs at higher and higher levels, so that movement through the root nodes is eventually sequential.

The only kind of resource management needed in the system is a form of "storage" management. This occurs only when additional L cells may be needed for an RA result. The L cells are obtained by moving L cell contents around to reposition empty cells, since the whole machine participates, this is a global process. The T network functions as an agent with global perspective. Storage management is highly concurrent, dynamic, automatic, and integrated; it is exclusively a function of the hardware.

Efficient parallelism is aided by the representation of the FFP expression in the L array. Representation provides the opportunity for parallelism both at and below the FFP level. Parallelism is maintained during execution by copying expressions, a process which is not advantageous on control flow machines. In this case, copying optimizes parallelism, which will then regain all the lost copying time many times over, or, at least, so Magó claims in Appendix A.

Many problems remain to be solved for a functional architecture such as Magó's. Among those are the following:

- suitable I/O and file systems are needed;
- a method for transparently using auxiliary memory is needed;
- suitable parallel algorithms need to be found.

3.3 Parallel Taxonomies

Parallel taxonomies in control flow models classify computing structures and provide a rough gauge to measure concurrent operation. These taxonomies will have to be extended to encompass highly concurrent models. An extended taxonomy will need to retain the property of serving as a measure of concurrency.

3.3.1 A Control Flow Taxonomy

Both [FLYN72] and [KUCK78] discuss parallel taxonomies for the control flow model. [KUCK78] is more useful, since it was originally derived from [FLYN72] and more carefully defined the control element and its input and output streams. The taxonomy discussed here is Kuck's [KUCK78].

Kuck defines an abstract processing unit called a "global control unit" (GCU): This is a hardware structure used to prepare instructions for sequencing the system. The GCU inputs an arbitrary number of undecoded instruction streams and outputs an arbitrary number (independent of the number of inputs) of decoded execution streams. Only "instantaneous descriptions" of the GCU are considered, or time intervals of just a very few clocks. Input and output lines refer to (practically) physically simultaneous events. Figure 19 graphically illustrates a GCU.

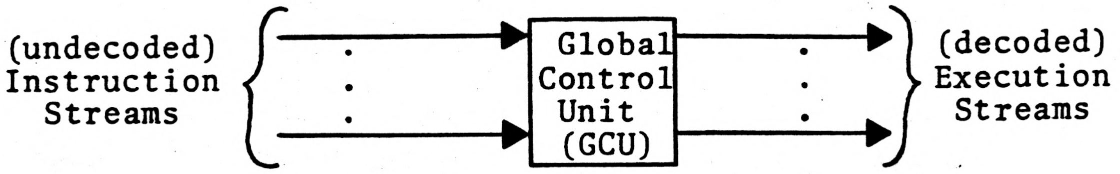

(undecoded) Instruction Streams
Global Control Unit (GCU)
(decoded) Execution Streams

Figure 19

Global Control Unit [KUCK78]

Kuck categorizes GCU's into four types as shown in Table 3. He then extends these four types to sixteen by considering combinations of scalar and array inputs and outputs. Table 4 lists a few of the scalar/array classifications. All are ultimately based on the control flow model. Kuck states the point of such a categorization is two-fold:

- it is useful to categorize machines based on GCU organizations;
- system capacity is strongly related to taxonomical categories.

3.3.2 An Extended Taxonomy

Kuck's taxonomy assumes instruction and execution streams in the conventional sense of multi-threaded instruction streams and lock-step data streams. Each computer in the system is assumed to be some form of control flow processor. Eventually, each computer is assumed to operate sequentially on a conventional control flow instruction stream using conventional control flow

GCU Type	Meaning	Example(s)
SISE	Single Instruction Single Execution	conventional uniprocessor
SIME	Single Instruction Multiple Execution	CDC 6600 CPU (multifunction processor)
MISE	Multiple Instruction Single Execution	CDC 6600 PPU's (uniprocessor with instruction-level multiprocessing)
MIME	Multiple Instruction Multiple Execution	conventional multiprocessor system

Table 3

Basic Types of Global Control Units (Taxonomies)

[KUCK78]

GCU Type	Meaning	Example(s)
SISSES	Single Instruction, Scalar Single Execution, Scalar	uniprocessor (same as SISE)
SISSEA	Single Instruction, Scalar Single Execution, Array	ILLIAC IV
SIASEA	Single Instruction, Array Single Execution, Array	Burroughs BSP TI ASC CDC STAR

Table 4

Some Types of Array GCU's

[KUCK78]

techniques. Concurrency is usually primarily derived from the whole of the computer grouping, as opposed to the actual groupings of processors within each computer of the external network. Each atomic machine usually doesn't contribute too much concurrency, unless a relatively expensive CPU is involved that pipelines parts of the control system and/or possibly operates on multiple data streams as an array processor does.

In highly concurrent models no atomic computer in any machine configuration can be assumed to be configured as a conventional control flow machine. Any single computer can be expected to consist of multiple processing elements, usually a comparatively large number when contrasted with conventional control flow machines. In some sense, each computer could itself be considered to be an MIME machine in Kuck's sense, but the ideas of "multiple instruction streams" and "multiple data streams" are much different in highly concurrent systems, where the concept of machine states are not at all the same. Whereas Kuck's taxonomy dealt primarily with two variables (instruction streams and execution streams) in a relatively limited sense, highly concurrent taxonomies will have to deal with a very large number of variables.

Kuck's taxonomy also contents itself with vague categorizations, differentiating only between "single" and "multiple" configurations. For the type of architectures it classifies, this is wholly adequate. But for highly concurrent architectures, even a rough classification is often going to require a more enumerative approach. For example, there can be a wide difference between a "multiple binary tree processor network" containing three processors and another containing fifteen.

An extended taxonomy embracing highly concurrent architectures should divide the implicitly sequential and implicitly concurrent single computer configurations into disjoint sets. Conventional control flow systems are adequately described by a scheme such as Kuck's. A more embracing scheme is needed for highly concurrent machine configurations. In the highly concurrent category, the problem then reduces to one of identifying performance parameters and classifying configurations using these parameters.

The remainder of this section will discuss a few parameters that might prove important in determining the performances of highly concurrent machines. A simple classification will be suggested based upon the parameters. The set of parameters is in no way implied to be complete. It is clear from the literature that much work needs to be done in this area, and it could well be found that a "complete," or even a "preferred," set of parameters cannot be identified. For our purposes, we will assume the five parameters chosen are somehow "best" in the sense of identifying an optimum set of parameters.

Following is a list of parameters that seem important in classifying performance on a single highly concurrent computer:

- a simple statistical enumeration of the number of processors in the internal computer network;
- the type of internal network organization utilized (assume pipelined or binary tree organizations for our purposes);
- the degree of internal processor interconnectedness;
- whether the internal network consists of homogeneous or nonhomogeneous processors in terms of instruction rates, etc.;
- some roughly quantitative measure of local to global memory in the internal network.

A simple statistical enumeration of the number of processors in a network reveals something about the processing power of a network. Adding processors to a network will usually increase processing power up to some point, depending upon the nature of the network.

The type of network organization utilized in the computer will be important. Binary tree organizations have properties not shared by pipelined organizations, for example. Depending on the computing situation, the choice of network organization could be very important. For example, binary tree organizations experience increased queuing and processor contention problems for processors higher in the tree, near the root node. An important set of subparameters for binary tree organizations would include such things as depth of the tree and the way in which the application would be implemented to use the tree.

The degree of processor interconnectedness is a measure of the number of other processors in a network with which a typical processor can directly communicate. In a binary tree network, a typical processor can directly communicate with three others, its parent and two children. In a pipelined network, a typical processor can communicate with two others. Of course, the root node in a binary tree network can only communicate with its two children, and certain processors in a pipelined network will only be able to communicate with one other processor; however, the degree of interconnectedness will measure statistical mode values and ignore the exceptions.

Measurements of interconnectedness will also have to account for increased complexity caused by adding interconnections. This effect can often negate any gains from increased interconnectedness.

Whether the network processors are all of equal types, with equivalent processing power in equivalent networks, may be a parameter of importance. The effects of varying such things as differing processor levels within a network configuration will need to be understood.

Local versus global memory accesses will be an important measure, since memory contention on such a computer will need to be understood. Perhaps one measure could be something as simple as a ratio of local to total memory words, where the total number of memory words in the computer is the sum of local and global memory word counts:

$$r = \frac{\ell}{\ell+g};$$

where ℓ = count of local words;

g = count of global words

Then, a machine with only local processor memory would have:

$$r = \frac{\ell}{\ell+g};$$

$$g = 0;$$

$$r = 1.$$

A computer with only global memory would have:

$$r = 0.$$

There could be many different levels of "global" memory in a system (i.e., memory accessible to all processors in the network versus memory accessible to more than one but less than all processors).

Finally, in a highly concurrent system, multiple highly concurrent computers can be connected to an external multiprocessing network. A whole set of new parameters can be determined for this second network. Many authors discuss this possibility of extensible machines and networks (i.e., [GUR780]). To simplify this section, the example to follow will consider only a single highly concurrent computer.

As an example, one possible type of taxonomy might consist of strings of text identifying combinations of the five parameters discussed previously. Suppose a series of highly concurrent computers were available with combinations of the parameters as follows:

- three, seven, or fifteen processors;
- binary tree organization;
- degree of interconnectedness = 3;
- processors of two levels will be available, with Level 2 "more powerful" than Level 1 (assume, however, that a given internal computer network will be composed of processors of either Level 1 or Level 2 types);
- no global memory in the computer, so

$$r = 1.$$

Table 5 lists the kind of rough taxonomy that would result from these considerations.

# Processors	Processor Level 1	Processor Level 2
3	3T311	3T321
7	7T311	7T321
15	15T311	15T321

Table 5

Example of (Partial) Highly Current Taxonomy

In Table 5, each taxonomical string in the table at the intersection of a row and column identifies a highly concurrent computer configuration, based on the variance of just two parameters (number of processors and processor level). A 7T311 configuration, for example, consists of 7 processors, binary tree configuration, degree of interconnectedness = 3, Level 1 processors, and no global memory (r=1).

In highly concurrent computers and networks there are going to be many interacting factors which will determine performance classifications. Taxonomies will probably be covered by statistical tables in book-sized publications. It will require many years of research with these networks to be able to make meaningful analytical generalizations about the performance within a given taxonomical family when the external network and internal network parameters are varied. In fact, just the determination of a relevant set of parameters will be a very difficult task.

3.4 Comparing Highly Concurrent and Control Flow Computing Implementations

This report previously compared highly concurrent and control flow models in terms of concurrency and history sensitivity. This section will compare the models in a few implementation areas. A little reflection in each area will convince the reader that each area relates in some way to the properties of concurrency and history sensitivity in the underlying computing models. Some important differences between the implementations which will be briefly reviewed in this section are:

- naming conventions;
- flow of data values;
- side effects;
- parallel taxonomies.

A control flow implementation equates variable names with storage locations. This is quite different from methods used in any highly concurrent implementation. A data flow machine assigns variable names to arcs on the program graph, and the functional machine avoids names completely by assigning values to processors in the L-Array, initially, and allowing subsequent subtree partitioning operations to keep intermediate and final values separately identified.

A control flow machine has a distinct disadvantage when compared to highly concurrent machines since its method of naming variables necessitates constant processing of two types of values: names (or addresses) and data values. This becomes quite costly and complex since each of these types must be processed through the von Neumann bottleneck. None of the highly concurrent machines suffers such redundancy; they process all data values directly.

However, none is history sensitive, either. For a data flow machine the naming convention is highly restrictive, since a (receiving) variable name can be used at only one place in a source program (the single-assignment rule).

In a control flow machine named data variables "stand still" in static memory locations, while a sequence of operations are performed upon these names and the associated data values. In highly concurrent machines the variables "move" through the computing networks and processors dynamically, and no static memory mapping is done to establish named locations for values. This is not as different from control flow computing as it may appear at first glance. Values also "move" from memory, to registers, to arithmetic processors, etc., during computation in a control flow machine. During computations, there are sequences of time periods when values associated with named variables in assigned memory locations are undefined, as computation proceeds with associated variables in (remote) processors. But control flow computing demands that final values be stored in memory; no highly concurrent machine requires this.

Only a control flow machine experiences the phenomenon known as "side effects". Since naming and storage of data values is distinctly separate from associated processing, shared and multiply-mapped memory locations can be changed unexpectedly from the perspective of one machine routine by the processing of another. This occurs in parallel processing of storage locations shared among routines, in subroutine parameter-name mappings to variables in locations common to both calling and called routines, and in other types of common storage mappings. Highly concurrent machines do not display these types of side effects.

Taxonomies in control flow computing classify the quantity of concurrency in parallel operations. The types of hardware structures, data streams, data flow, etc., are all important in these classifications. Control flow computing is implicitly sequential, so a parallel taxonomy of this sort is extremely important. In implicitly concurrent machines, however, many more parameters are involved, and highly concurrent taxonomies will probably ordinarily consist of multiple tables of statistical values and enumerations.

CHAPTER 4

Conclusion

Most authors feel as Treleaven [TREL79] does: "data flow systems are a fundamentally new style of (tightly coupled, distributed) computer which could eventually supersede the conventional general purpose (von Neumann) computer." Yet, in the conclusion of that report, a painfully obvious note is taken of the current state of the art in highly concurrent computing: "most research has concentrated on the programming and evaluation of numerical algorithms. Little study has been made of how activities such as I/O (or) semi-permanent storage (file storage) should be controlled or programmed in a data flow computer, using a data flow language. It is unclear whether it will be possible to practically widen applicability of data flow computers. The data flow approach may be restricted to parallel (numerical) algorithms, or it may prove possible to find a suitable synthesis of the data and control flow approaches."

The history sensitivity property of control flow computing is like a two-edged sword. The undesirable properties of side effects, memory accessing bottlenecks, etc., are there largely because of the control flow implementation of this property. Yet, commercial business computing, text handling applications, efficient file handling, etc., would not be possible today without this property. It would seem that much research remains to be done on models, properties of models, and data operations in models in general before any conclusion about the property can be reached. Perhaps a satisfactory mix of history sensitivity, concurrency, and non-numerical data operations has yet to be combined in the right kind of model.

The very lack of history sensitivity may be the primary reason for the strengths of the properties of algebraic representation and concurrency in the data flow and applicative models of this report. Certainly in the data flow model, the assigning of a variable name to an arc aids concurrency and representation at the cost of sensitivity. In numerical processing it may be worth the price to trade these properties off in this way, but in non-numerical processing history sensitivity seems essential.

The Functional Programming Language of Backus illustrates the strength of the algebraic representation property very well. This kind of power for numerical processing certainly can't be obtained with traditional approaches. However, the need for this kind of symbology in more pragmatic commercial areas of computing is questionable. If the language were too mathematical for file processing, for example, it might discourage a large proportion of users.

Table 6 [MEAD80] combined with memory locality and improved parallel algorithms for highly concurrent structures illustrates concurrency and its obvious advantages. The figures were derived from Mead's model for memory access time (Section 1.2.1). Gostelow and Thomas [GOST80] present a performance study of data flow architectures. Figure 20 summarizes their findings by plotting number of processing elements versus time.

Technique	Typical Speedup Factor
Memory Hierarchy	10
Pipelining	
Instruction Overlap	2
Special-Purpose	n
Multiprocessors	<n

Table 6

Speedup Factors (n Processors) [MEAD80]

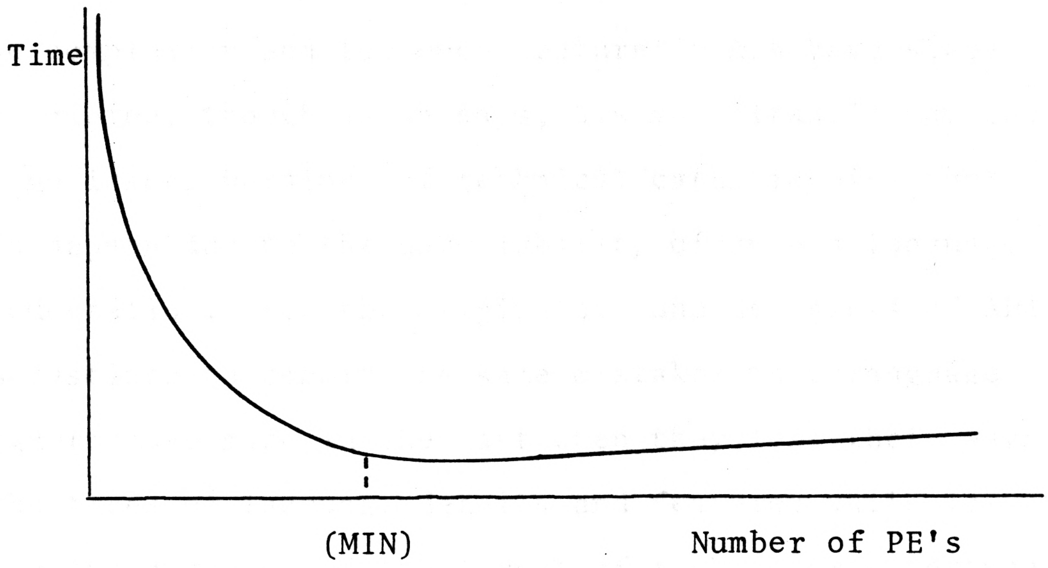


Figure 20

Speedup Curve for Data Flow Speedup Experiments of [GOST80]

If there is one central message the author of this paper would like to convey, it is that research of computing models for all areas of computing (i.e., numerical, non-numerical) is going to be necessary in the VLSI era. The problems of language definition and development in this era will not be restricted to control flow models, and language developers will have to assume a more innovative niche in the total computer design process than they have in the past. This problem is discussed in Chapter 1.

The 1977 Turing Award Lecture of Backus [BACK77] provides an important framework for this report. It seems fitting to end the report by referring to another such lecture, the 1980 Turing Award Lecture of C.A.R. Hoare [HOAR81]. Professor Hoare was hired by one of the most powerful and influential organizations in the world, the United States Department of Defense, in 1975 to provide consultation on their ADA language. His warnings of immense complexity and too much feature in ADA have since gone ignored, though as he says, his consultant's pay goes on. He issues warnings of technical catastrophies that could happen due to the unreliability of such a language implementation. But the originators and designers of ADA seem destined to commit the same mistakes that language designers have made in the past when they lost their ways in the trees of language details and features while ignoring the advances of the forest of machine architecture and language representation. In frustration and protest he ends his lecture with the allegory which will end this report. The following section is quoted from [HOAR81].

4.1 The Emperor's Old Clothes [HOAR81]

Many years ago, there was an emperor who was so excessively fond of clothes that he spent all his money on dress. He did not trouble himself with soldiers, attend banquets, or give judgment in court. Of any other king or emperor one might say, "he is sitting in council," but it was always said of him, "the emperor is sitting in his wardrobe." And so he was. On one unfortunate occasion, he had been tricked into going forth naked to his chagrin and the glee of his subjects. He resolved never to leave his throne, and to avoid nakedness, and he ordered that each of his many new suits of clothes should be simply draped on top of the old.

Time passed away merrily in the large town that was his capital. Ministers and courtiers, weavers and tailors, visitors and subjects, seamstresses and embroiderers, went in and out of the throne room about their various tasks, and they all exclaimed, "how magnificent is the attire of our emperor."

One day the emperor's oldest and most faithful minister heard tell of a most distinguished tailor who taught at an ancient institute of higher stitchcraft, and who had developed a new art of abstract embroidery using stitches so refined that no one could tell whether they were actually there at all. "These must indeed be spendid stitches," thought the minister. "If we can but engage this tailor to advise us, we will bring the adornment of our emperor to such heights of ostentation that all the world will acknowledge him as the greatest emperor there has ever been."

So the honest old minister engaged the master tailor at vast expense. The tailor was brought to the throne room where he made obeisance to the heap of fine clothes which now completely covered the throne. All the courtiers waited eagerly for his advice. Imagine their astonishment when his advice was not to add sophistication and more intricate embroidery to that which already existed, but rather to remove layers of finery, and strive for simplicity and elegance in place of extravagant elaboration. "This tailor is not the expert that he claims," they muttered. "His wits have been addled by long contemplation in his ivory tower and he no longer understands the sartorial needs of a modern emperor." The tailor argued loud and long for the good sense of his advice but could not make himself heard. Finally, he accepted his fee and returned to his ivory tower.

Never to this very day has the full truth of this story been told: that one fine morning, when the emperor felt hot and bored, he extricated himself carefully from under his mountain of clothes and is now living happily as a swineherd in another story. The tailor is canonized as the patron saint of all consultants, because in spite of the enormous fees that he extracted, he was never able to convince his clients of his dawning realization that their clothes have no emperor.

REFERENCES

ALLA76 Allan, S. and Oldehoeft, A., "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language", Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pages 26-34.

BACK78 Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", CACM, Vol. 21, No. 8, August 1978, pages 613-641.

BROC79 Brock, J., and Montz, L., "Translation and Optimisation of Data Flow Programs", Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pages 46-54.

COMT74 Comtre Corp., Multiprocessors and Parallel Processing, Philip H. Enslow, Ed., John Wiley and Sons, New York, 1974.

DENN79 Dennis, J., and Weng, K., "An Abstract Implementation for Concurrent Computation with Streams", Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pages 35-45.

DENN80 Dennis, J., "Data Flow Supercomputers", Computer, Vol. 13, No. 11, November 1980, pages 48-56.

DIJ168 Dijkstra, E.W., "Cooperating Sequential Processes", Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, pages 43-112.

DIJ268 Dijkstra, E.W., "The Structure of THE Multiprogramming System", CACM, Vol. 11, No. 5, May 1968, pages 341-346.

DIJK71 Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes", ACTA INFORMATICA, Vol. 1, 1971, pages 115-138.

FENN77 Fennell, R.D., and Lesser, V.R., "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II", IEEE TOC, Vol. C-26, No. 2, February 1977, pages 98-111.

FINN77 Finnila, C.A., and Love, H.H., Jr., "The Associative Linear Array Processor", IEEE TOC, Vol. C-26, No. 2, February 1977, pages 112-125.

FLYN72	Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE TOC, Vol. C-21, No. 9, September 1972, pages 948-960.
GOST80	Gostelow, K.P., and Thomas, R.E., "Performance of a Simulated Dataflow Computer", IEEE TOC, Vol. C-29, No. 10, October 1980, pages 905-919.
GUR680	Gurd, J., and Watson, I., "Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution", Computer Design, June 1980, pages 91-100.
GUR780	Gurd, J., and Watson, I., "Data Driven System for High Speed Parallel Computing - Part 2: Hardware Design", Computer Design, July 1980, pages 97-106.
HANS77	Hansen, P.B., The Architecture of Concurrent Programs, Prentice-Hall, Englewood Cliffs, N.J., 1977.
HANS79	Hansen, P.B., "A Keynote Address on Concurrent Programming", Computer, Vol. 12, No. 5, May 1979, pages 50-56.
HOAR81	Hoare, C.A.R., "The Emperor's Old Clothes", CACM, Vol. 24, No. 2, February 1981, pages 75-83.
JOHN80	Johnson, D., et al, "Automatic Partitioning of Programs in Multiprocessor Systems", VLSI: New Architectural Horizons, IEEE COMCON, Spring 1980, pages 175-178.
KAMI79	Kaminsky, W.J., and Davidson, E.S., "Developing a Multiple-Instruction-Stream Single-Chip Processor", Computer, Vol. 12, No. 12, December 1979, pages 66-76.
KELL80	Keller, R.M., Linstrom, G., and Patil, S., "Dataflow Concepts for Hardware Design", VLSI: New Architectural Horizons, IEEE COMCON, Spring 1980, pages 105-111.
KUCK78	Kuck, D.J., The Structure of Computers and Computations, John Wiley and Sons, New York, 1978.
KUCK79	Kuck, D.J., and Padua, D.A., "High Speed Multiprocessors and Their Compilers", Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pages 5-16.
KUNG80	Kung, H.T., "The Structure of Parallel Algorithms", Advances in Computers, Marshall C. Yovitts, Ed., Academic Press, New York, Vol. 19, 1980, pages 65-112.

LIPO77	Lipovski, G.J., "On a Varistructured Array of Microprocessors", IEEE TOC, Vol. C-26, No. 2 February 1977, pages 125-138.
MAGO80	Magó, D., "A Cellular Computer Architecture for Functional Programming", VLSI: New Architectural Horizons, IEEE COMCON, Spring 1980, pages 179-185.
MEAD80	Mead, C., and Conway, L., "Highly Concurrent Systems", Chapter 8, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980, pages 263-332.
NECH79	Neches, P.M., "Conference Report: VLSI Architecture, Design, and Fabrication", Computer, Vol. 12, No. 5, May 1979, pages 76-78.
RALS76	Ralston, A., and Meek, C.L., Eds., Encyclopedia of Computer Science, Petrocelli/Charter, New York, 1976.
RUMB75	Rumbaugh, J., "A Parallel Asynchronous Computer Architecture for Data Flow Programs", MIT Project MAC, TR-150, May 1975.
RUMB77	Rumbaugh, J., "A Data Flow Multiprocessor", IEEE TOC, Vol. C-26, No. 2, February 1977, pages 138-146.
SCHN79	Schneck, P., "Issues in Parallel Computing: A Non-Euclidean Examination", Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pages 1-4.
SCHW80	Schwartz, J.T., "Ultracomputers", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980, pages 484-521.
SLEE80	Sleep, M.R., "Applicative Languages, Dataflow, and Pure Combinatory Code", VLSI: New Architectural Horizons, IEEE COMCON, Spring 1980, pages 112-115.
STON73	Stone, H.S., "Problems of Parallel Computation", Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub, Ed., Academic Press, New York, 1973, pages 1-6.
SUGA80	Sugarman, R., "VLSI Computing: A Tough Nut to Crack", IEEE SPECTRUM, January 1980, pages 34-35.
TREL79	Treleaven, P.C., "Exploiting Program Concurrency in Computing Systems", Computer, Vol. 12, No. 1, January 1979, pages 42-50.

APPENDIX A

Some Details of Operation and an Example of Magó's Functional Architecture

This appendix reproduces two sections of [MAGO80] in their entirety: "Some Details of Operation," and "Efficiency of Program Execution: Fundamental Issues." The first section includes an example microprogram (Apply to All). It's very difficult to summarize these sections, and the example cannot be replaced without further knowledge of the microprogram architecture, which was not available to the author of this report. For those desiring better detail than the sketchy summary in Section 3.2, this appendix will provide full explanations from the source document. The remainder of this appendix is taken directly from [MAGO80].

Some Details of Operation

Decomposition of FFP Programs.

As the computation unfolds, each RA produces changes, often large ones, in the FFP expression. Consequently, it is imperative that the machine be able to decompose anew in each machine cycle this ever-changing FFP text. The need for decomposition arises in two different situations. First, at the beginning of each machine cycle the whole FFP expression held by the L array is considered, and all RAs in it must be located. Later, in the process of executing RAs, certain subexpressions of these applications, such as their operators, operands, or subexpressions thereof, must be located.

Partitioning the Network.

Once an FFP expression is placed in the L array, L cells (or collections of L cells) may be thought of as being dedicated to FFP symbols (or FFP subexpressions), at least for the duration of one machine cycle. The idea of also dedicating entire T cells to computations is quite an obvious next step, but setting up a correspondence between L and T cells with just the right properties does not seem possible.

The example in Figure A.2 shows how the machine dedicates the resources of T cells to computations by breaking each T cell into at most four parts, and allocating these parts to computations. The example reveals two properties of the partitioning: (1) different RAs "own" disjoint sets of resources (L cells and parts of T cells); and (2) these resources are always connected to form binary trees, with L cells as their leaves. The first property makes a practical possibility out of a theoretical one: now all RAs <u>have</u> the necessary resources to begin their execution simultaneously, as permitted by the Church-Rosser property of FFP languages. The second property means that each RA has a small "tree machine" all to itself (with all the advantages this implies), just as if it were alone in the original processor--partitioning the original network never diminishes the quality of resources made available to the RAs.

The process of partitioning the original network (itself a "tree machine") into a collection of disjoint, smaller "tree machines" is (1) automatic--it is completely determined by the FFP expression and its placement in the L array; (2) dynamic--it is done once in each machine cycle, to keep up with the changing FFP program text; (3) fast--it takes one upsweep and one downsweep.

Programming a Collection of Cells.

Having been located in the L array and given all the resources it needs, the RA is now ready to begin execution. The definition of the FFP language gives little guidance here: it only specifies what the result expression should be, given the operator and the operand expressions. The problem is to devise a way to cause a collection of cells (more precisely: L cells and parts of T cells) to transform the RA into the result expression. This collection of cells, allocated to reduce the RA, is itself a cellular computer: its processing resources are evenly distributed over the cells, and no cell in it can ever have complete information about what is going on during execution.

What is needed is a suitable programming language. The programmer, writing programs in this language, would prepare a plan for all the cells involved to act in concert. When executing such a program, the elementary actions of the cells (each cell using local information only) would combine harmoniously and effectively to bring about the desired (global) transformation of the RA.

A programming language capable of serving such a purpose, and able to define a large class of transformations of FFP expressions, has been described. It is referred to as the microprogramming language partly because it is below the level of the FFP language (which is the "machine language" of the network), and partly because it does resemble conventional microprogramming languages. The following are important characteristics of this microprogramming language:

1. Microprograms normally reside outside the network of cells, and are brought in only on demand. This helps

keep both L and T cells small. It also provides for flexibility: FFP language primitives are easier to change, different users may have different sets of primitives, and so on.

2. Once a microprogram is brought into the processor, it is placed in the L cells holding the RA. Each L cell receives only a fraction of the microprogram: just what is necessary to make its own contribution to the total computation. (Subexpressions of an RA are found by the relevant parts of the microprogram through, again, a form of program decomposition.)
3. The purpose of the microprogram is to transform the RA into the result expression. Therefore, the microprogram is aimed explicitly at changing the contents of L cells, and uses the T cells (or parts thereof) only implicitly, mostly for purposes of communicating among L cells. For example, if one of the L cells wants to broadcast some information to all other L cells involved in reducing the same RA, it executes a SEND instruction, explicitly identifying the information item to be broadcast. As a result, the information item is moved automatically to the root of the RA's tree, and from there it is broadcast to all L cells of the RA, again automatically.
4. The microprogramming language is able to exploit the potentials for low-level parallelism offered by the fact that there is at most one FFP symbol per L cell.

When writing a microprogram, one decomposes the required transformation into elementary computations, many of which can then be executed concurrently by different cells. As an example, consider the execution of an FFP primitive whose purpose is to normalize a vector of numbers by dividing each component of the vector by the Euclidean length of that vector. Assuming that the vector is represented as an FFP sequence of numeric atoms, a microprogram can prescribe the following execution sequence: (a) for each i the cell holding xi computes (xi)**2--these computations are done simultaneously for every i; (b) for each i the L cell holding xi sends (xi)**2 up into the tree--these are done simultaneously for every i; (c) in one upsweep the sum of squares is produced in the root cell of the RA's tree (whenever a T cell receives two numbers from its children, it performs an addition, and sends the sum to its father); (d) the sum of squares is broadcast to every L cell of the RA, and each L cell holding xi for some i accepts this sum; (e) each L cell holding xi for some i computes the square root of the sum just received, and finally divides xi by this number. These computations can again be carried out simultaneously, producing the desired normalized vector.

5. The microprogram is written before execution begins (the FFP language does not allow changing the set of primitives during execution), and consequently it must be able to deal with aspects of the computation that

become known only at run-time. For example, the primitive may want to copy a subexpression of the operand whose size becomes known only at run-time, or it may want to select the ith element of a sequence where i is a parameter supplied at run-time.

As an example, Figure A.3 shows the innermost application (<AA,+>:<<1,11>,<2,12>,<3,13>,<4,14>>), which produces, as its result expression, <(+:<1,11>),(+:<2,12>),(+:<3,13>),(+:<4,14>)>. (AA stands for "Apply to All.") It also shows, in an informal manner, how the microprogram specifies the result of reducing this application. The microprogram is written in five separate parts. Parts 1 and 2 (received by cells 3 and 5, respectively) rewrite the FFP symbol and leave the nesting level number unchanged. Part 3 (received by cell 8) keeps the contents of the cell unchanged. In addition, the FFP symbol contained in this cell is marked with a symbol chosen by the writer of the microprogram (in this case with "x"). With the help of "x", Part 5 will be able to refer to the contents of this cell.

Part 5 is received by all occupied cells between 9 and 23, inclusive. These cells hold the operand of the innermost application in question. First, the whole expression is marked with the symbol "y" (this symbol must be different from the one used in Part 4, which was "x"). Among the effects of marking (executing a

MARK statement in the microprogram) is placing the number i in all L cells holding the ith element expression of the marked sequence. Thus, although every occupied cell between 9 and 23 receives exactly the same microprogram, the microprogram can test the value of the integer generated by marking, and can thereby ascertain what part of the operand expression it is working on. Hence the microprogram can do different things to different parts of the operand expression--again an example of program decomposition. In this particular case, the results of marking are used to pinpoint cells 14, 18 and 21, and execute in each a so-called INSERT statement of the microprogramming language, the effect of which is a declaration of _what_ should be inserted on the left or right of the FFP symbol held by the cell in question. In our example, we want to insert an application symbol with level number 1, followed by the parameter of AA. Since only at run-time will it be known what the parameter of AA is (in our example it is "+"), we mark this parameter with "x" so that the INSERT statement can refer to it symbolically. The INSERT statement simply initiates a sequence of events, which then take place automatically: getting the length of the expression to be inserted (which is determined by the MARK statement) to the place of insertion, requesting that number of empty cells, producing the required number of empty cells by moving the contents of L cells, and finally moving the expression to its final destination.

Communication during Processing.

The pattern of communication among L cells during processing is simple and always the same: information items are sent to the root of the RA's tree, and from there they are broadcast, one after another, to every L cell of the RA. The L cells have to specify only *what* they want to send and *what* they want to accept, and the rest of the machinery operates automatically. Sending every information item through the root node of the tree means that the logarithmic distance characteristics of the tree are well utilized, especially when L cells far from each other have to communicate. It also means that the time taken to move a large number of items is proportional to the number of items moved through the root node. The tree used this way is a very simple routing network: the upward moving items queue up throughout the tree, waiting to move through the root node sequentially. Investigations have been done into ways of using cross connections in the tree network to speed up communication in this kind of machine (i.e., without the use of addresses).

Resource Management.

It often happens that the result expression cannot be produced in the L cells that held the initial RA, because, for example, the result expression is too long. In such cases execution can continue only if sufficiently many empty L cells are made available to the RA in question.

If the required number of empty cells is available somewhere in the L array, they can be made available to the RA in question by moving the contents of occupied L cells, thereby repositioning the empty cells. This process is called <u>storage management</u>. This is the only kind of resource management needed in the processor because whenever an RA has all the L cells it needs, it is guaranteed to receive, with the help of the partitioning mechanism, all the T cells (or parts thereof) it needs.

Storage management in the machine is <u>global</u>, meaning that the whole machine participates in it, so that as many requests for insertions can be satisfied as possible, and all empty cells in L can be utilized to satisfy these requests. The T network is used to determine how far and in what direction each FFP symbol should be moved in L to position the required number of empty cells in the right places relative to the FFP symbols. Although each T cell works with local information only, on this occasion the T network as a whole acts as an agent with a "global understanding" of the situation in L.

Storage management in the machine is <u>highly concurrent</u>: all FFP symbols move simultaneously, under local control, to their destinations in L. (If the connections between L cells are used, the process of repositioning the FFP symbols is similar to, although more general than, the operation of a shift-register: different FFP symbols may move in different directions and by differing amounts before coming to a halt.)

Storage management in the machine is *dynamic*: it is done once in each machine cycle. Thus, the L cells released in one machine cycle can immediately be reallocated to other subcomputations for the next cycle, and the processor can immediately attempt to satisfy requests for empty L cells made during the current machine cycle.

Storage management in the machine is *automatic*: initiating it requires no action on the part of the FFP programmer, only on the part of the writer of the microprograms. Moreover, no system software is involved: storage management is exclusively the function of the hardware.

Finally, storage management in the machine is *integrated*: being the only resource management mechanism in the machine, it manages storage at once among different user programs, among different subcomputations (RAs) of the same user program, and also on the lowest level, among subexpressions and individual symbols of a single RA.

Efficiency of Program Execution: Fundamental Issues

In trying to grasp the peculiar qualities that set this machine apart from all others proposed to date, one is led to consider two issues, both of which seem to have a decisive influence on the operational characteristics of the machine.

The first peculiarity is the representation of the FFP expression in the L array. It almost inevitably leads to the patterns of communication employed in the machine, and most of these communications may be viewed as efforts to maintain the representation. For example, some of the most time-consuming aspects of executing an RA are the rearranging of the FFP expression (e.g., copying a subexpression from one place to another) and the often accompanying storage management. These are always aimed at bringing the operator and operand expressions together, or producing operand expressions in the syntactic form required by some operator to be applied later. (There is never any need to explicitly communicate the result of an RA--it is just left in L wherever it is produced.) The primitive operator AA, used in Figure A.3 illustrates one means of forming new applications by bringing operator and operand expressions together. Of course, the machine needs no special planning to accomplish this (other than faithfully executing RAs): the FFP programmer simply composes FFP operators in such a way that the intended expressions are brought together.

The representation also plays a crucial role by providing opportunity for parallelism both on and below the FFP level. The connection seems inherent: parallelism is made possible by the representation, which, in turn, is maintained by copying expressions. Therefore literal copying, eschewed on the von Neumann computer, is tolerated here: it unlocks parallelism, which can be used to regain, often many times over, the time "lost" in copying. This remark is not based on vague hopes of being eventually justified by some future implementation. Credible statements about the complex interaction between the positive forces of parallelism and the negative forces of literal copying--pitted against each other in every machine cycle--can be substantiated by detailed quantitative reasoning about programs executing on the machine.

The second peculiarity of the machine is its <u>ability to handle complex operands (i.e., data structures)</u> within innermost applications. (In this respect, the machine appears to differ greatly even from the data flow computers recently surveyed by Dennis.) The FFP language places no limitations on what a language primitive can do to its operand. The machine, on the other hand, does have some inherent limitations. Because of the finiteness of its cells, for example, it cannot "see" the details of subexpressions nested too deeply in the operand and operator. Despite such limitations, the machine can efficiently implement, and the microprogramming language can express as FFP primitive operations, a large class of transformations on operand expressions. This class includes transposing a square matrix of atoms, performing an n-point Fourier transformation, finding the kth largest element of a set, and determining whether two arbitrary expressions are the same. The key to this ability of the machine is that RAs, regardless of their size, are handled by the same cellular machinery: a sufficiently large assembly of cells (L cells and parts of T cells) is organized, and this assembly, under the control of the applicable microprogram, brings about the required transformation of the operand (and possibly also of the operator) expression.

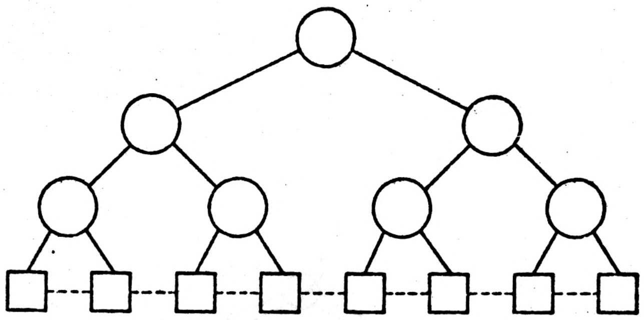


Figure A.1a

Interconnection of Cells

[MAGO80]

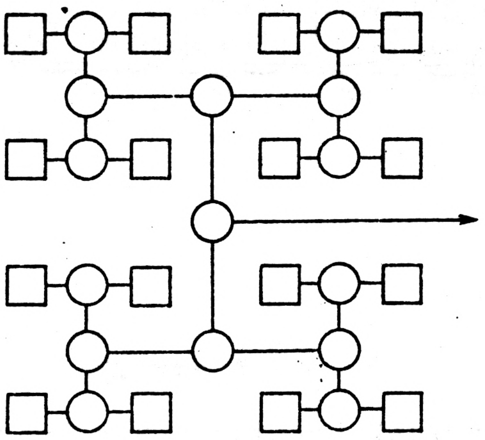


Figure A.1b

A Possible Layout Scheme

[MAGO80]

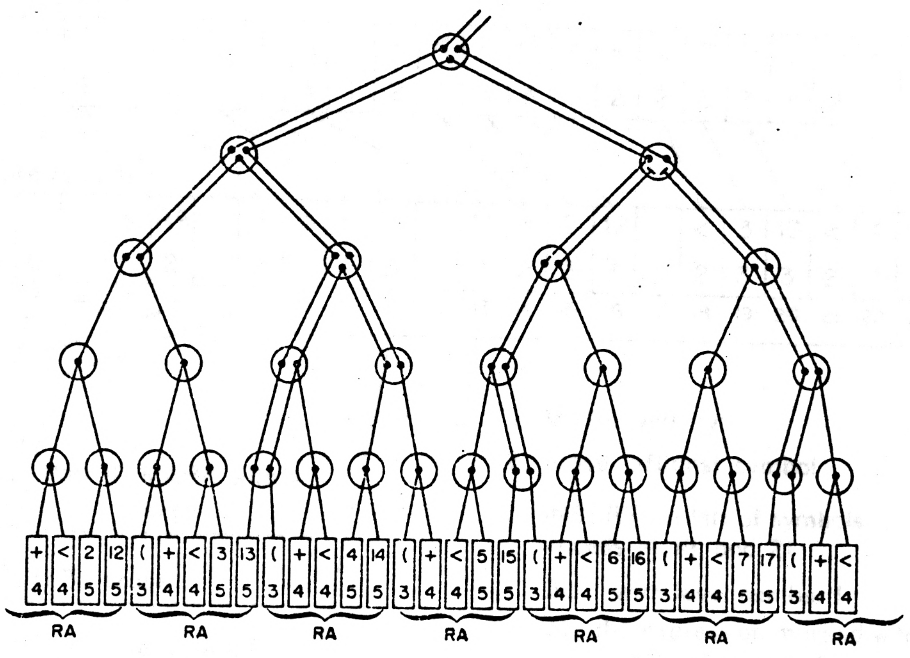
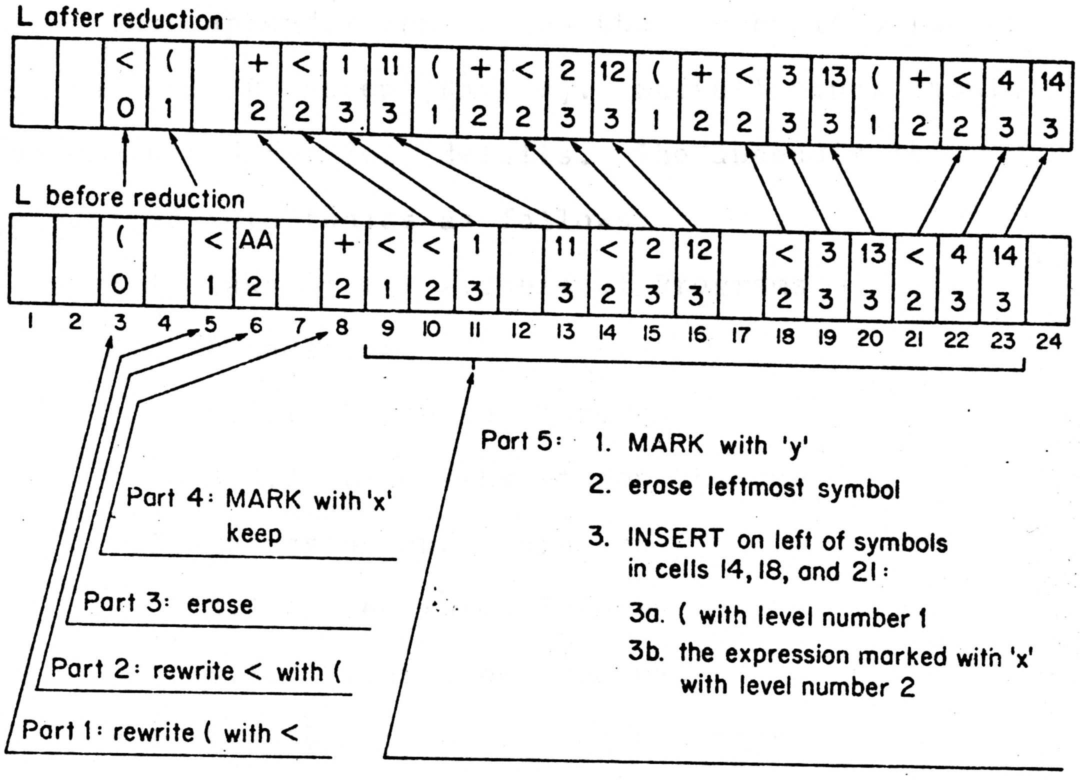


Figure A.2

Fragment of a Partitioned Network

[MAGO80]



L after reduction
L before reduction
Part 1: rewrite (with <
Part 2: rewrite < with (
Part 3: erase
Part 4: MARK with 'x' keep
Part 5: 1. MARK with 'y'
2. erase leftmost symbol
3. INSERT on left of symbols in cells 14, 18, and 21:
3a. (with level number 1
3b. the expression marked with 'x' with level number 2

Figure A.3

Microprogram for AA (Apply to ALL)

[MAGO80]

APPENDIX B

The Algebra of Functional Programs

This appendix summarizes the algebraic structure of Backus' FP system [BACK78]. Backus' paper should be consulted for full details. The appendix is organized into topics as follows:

B.1 Laws of the Algebra of Programs

Backus presents some definitions and a list of algebraic laws for the algebra of programs. These definitions and laws are listed, here, so that they can be used later to help illustrate examples and proofs.

Definition. "defined"

The "defined" definition is used to define the domain of a function. Many laws have a domain that is only a proper subset of the domain of all objects. For example, $1\circ[f,g]\equiv f$ is true only when g is properly defined. If $g:x=\bot$, then the law does not hold. The notation

$$\text{defined}\circ g \twoheadrightarrow 1\circ[f,g]\equiv f$$

indicates the law (or theorem) on the right holds only within the domain of objects x for which $\text{defined}\circ g:x=T$.

A qualified functional equation is written

$$p \twoheadrightarrow f\equiv g$$

and means that, for any object x, whenever $p:x=T$, then $f:x=g:x$.

The following definitions specify ordering on functions and functional equivalence in terms of the ordering.

Definition. $f\leq g$ iff for all objects x, either $f:x=\bot$, or $f:x=g:x$.

Definition. $f\equiv g$ iff $f\leq g$ and $g\leq f$.

The list of algebraic laws is organized by the two principal functional forms involved. This list follows and is copied verbatim from Backus.

I Composition and construction

I.1 $[f_1, \ldots, f_n] \circ g \equiv [f_1 \circ g, \ldots, f_n \circ g]$

I.2 $\alpha f \circ [g_1, \ldots, g_n] \equiv [f \circ g_1, \ldots, f \circ g_n]$

I.3 $/f \circ [g_1, \ldots, g_n]$

$\equiv f \circ [g_1, /f \circ [g_2, \ldots, g_n]]$ when $n \geq 2$

$\equiv f \circ [g_1, f \circ [g_2, \ldots, f \circ [g_{n-1}, g_n] \ldots]]$

$/f \circ [g] \equiv g$

I.4 $f \circ [\overline{x}, g] \equiv (\text{bu } f\ x) \circ g$

I.5 $1 \circ [f_1, \ldots, f_n] \leq f_1$

$s \circ [f_1, \ldots, f_s, \ldots, f_n] \leq f_s$ for any selector s, $s \leq n$

$\text{defined} \circ f_i$ (for all $i \neq s$, $1 \leq i \leq n$) $\rightarrow\!\!\rightarrow s \circ [f_1, \ldots, f_n] \equiv f_s$

I.5.1 $[f_1 \circ 1, \ldots, f_n \circ n] \circ [g_1, \ldots, g_n] \equiv [f_1 \circ g_1, \ldots, f_n \circ g_n]$

I.6 $\text{tl} \circ [f_1] \leq \Phi$ and $\text{tl} \circ [f_1, \ldots, f_n] \leq [f_2, \ldots, f_n]$ for $n \geq 2$

$\text{defined} \circ f_1 \rightarrow\!\!\rightarrow \text{tl} \circ [f_1] \equiv \bar{\phi}$

and $\text{tl} \circ [f_1, \ldots, f_n] \equiv [f_2, \ldots, f_n]$ for $n \geq 2$

I.7 $\text{distl} \circ [f, [g_1, \ldots, g_n]] \equiv [[f, g_1], \ldots, [f, g_n]]$

$\text{defined} \circ f \rightarrow\!\!\rightarrow \text{distl} \circ [f, \Phi] \equiv \Phi$

The analogous law holds for distr.

I.8 $\text{apndl} \circ [f, [g_1, \ldots, g_n]] \equiv [f, g_1, \ldots, g_n]$

$\text{null} \circ g \rightarrow\!\!\rightarrow \text{apndl} \circ [f, g] \equiv [f]$

And so on for apndr, reverse, rotl, etc.

I.9 $[\ldots, \bar{\bot}, \ldots] \equiv \bar{\bot}$

I.10 $\text{apndl} \circ [f \circ g, \alpha f \circ h] \equiv \alpha f \circ \text{apndl} \circ [g, h]$

I.11 pair & $\text{not} \circ \text{null} \circ 1 \rightarrow\!\!\rightarrow \text{apndl} \circ [[1 \circ 1, 2], \text{distr} \circ [\text{tl} \circ 1, 2]]$

$\equiv$ distr

Where $f\&g \equiv \text{and}\circ[f,g]$; $\text{pair} \equiv \text{atom} \rightarrow \bar{F};\text{eq}\circ[\text{length},\bar{2}]$

II Composition and condition (right associated parentheses omitted).

II.1 $(p\rightarrow f;g)\circ h \equiv p\circ h \rightarrow f\circ h;\ g\circ h$

II.2 $h\circ(p\rightarrow f;g) \equiv p \rightarrow h\circ f;\ h\circ g$

II.3 $\text{or}\circ[q,\text{not}\circ q] \twoheadrightarrow\ \text{and}\circ[p,q] \rightarrow f;$

$\text{and}\circ[p,\text{not}\circ q] \rightarrow g;\ h \equiv p \rightarrow (q\rightarrow f;g);h$

II.3.1 $p \rightarrow (p\rightarrow f;g);\ h \equiv p \rightarrow f;h$

III Composition and miscellaneous

III.1 $\bar{x}\circ f \leq \bar{x}$

$\text{defined}\circ f \twoheadrightarrow\ \bar{x}\circ f \equiv \bar{x}$

III.1.1 $\bar{\bot}\circ f \equiv f\circ\bar{\bot} \equiv \bar{\bot}$

III.2 $f\circ \text{id} \equiv \text{id}\circ f \equiv f$

III.3 $\text{pair} \twoheadrightarrow 1\circ\text{distr} \equiv [1\circ 1,2]$ also:

$\text{pair} \twoheadrightarrow 1\circ\text{tl} \equiv 2$ etc.

III.4 $\alpha(f\circ g) \equiv \alpha f \circ \alpha g$

III.5 $\text{null}\circ g \twoheadrightarrow \alpha f\circ g \equiv \bar{\phi}$

IV Condition and construction

IV.1 $[f_1, \ldots, (p \rightarrow g;\ h), \ldots, f_n]$

$\equiv p \rightarrow [f_1, \ldots, g, \ldots, f_n];[f_1, \ldots, h, \ldots, f_n]$

IV.1.1 $[f_1, \ldots, (p_1 \rightarrow g_1; \ldots ;p_n\rightarrow g_n;\ h), \ldots, f_m]$

$\equiv p \rightarrow [f_1, \ldots, g_1, \ldots, f_m];$

$\ldots ;p_n \rightarrow [f_1, \ldots, g_n, \ldots, f_m];\ [f_1, \ldots, h, \ldots, f_m]$

This concludes the present list of algebraic laws: it is by no means exhaustive. there are many others.

B.2 Foundations

Backus' goal is to develop a foundation for the algebra of programs that is based on a sufficient theoretical base to allow the programmer to use simple algebraic laws plus some theorems from the foundations to solve problems and prove functions (programs). The proofs will be algebraically mechanical and will be written directly in the programming language. The latter point is very important: the logical system used by program proofs is identical to that used for writing the program.

An expansion theorem, a linear expansion theorem, and a corollary to the latter are stated and proved as part of the foundations. These results are used later in conjunction with the algebraic laws to establish recursion and iteration theorems. Recursion and iteration theorems are stated in section B.3; they allow looping and iteration in the language.

The Expansion Theorem also provides a method to prove "termination". In the statement of the theorem and its associated definition, there is the following stipulation:

f:x is defined if and only if there is an n such that, for every i less than n, p_i:x=F, P_n:x=T, and

q_n:x is defined.

This stipulation is sufficient to establish termination.

The following sections (B.2.1 and B.2.2) state the definitions for "expansive" and "linearly expansive," and the Expansion and Linear Expansion Theorems, plus the corollary. Proofs can be found in Backus [BACK78].

B.2.1 Expansion Theorem

Definition. Expansion. Suppose we have an equation of the form

$$f \equiv E(f)$$

where E(f) is an expression involving f. Suppose further that there is an infinite sequence of functions f_i for i=0,1,2,..., each having the following form:

$$f_o \equiv \bar{\bot}$$

$$f_{i+1} \equiv p_o \rightarrow q_o \ ; \ \ldots \ ; \ p_i \rightarrow q_i \ ; \ \bar{\bot}$$

where the p_i's and q_i's are particular functions, so that E has the property:

$$E(f_i) \equiv f_{i+1} \text{ for } i=0,1,2, \ldots$$

Then we say that E is expansive and has the f_i's as approximating functions.

Expansion Theorem. Let E(f) be expansive with approximating functions as given in the definition of expansion. Let f be the least function satisfying

$$f \equiv E(f).$$

Then

$$f \equiv p_o \rightarrow q_o \; ; \; \ldots \; ; \; p_n \rightarrow q_n \; ; \; \ldots$$

B.2.2 Linear Expansion Theorem

Definition. Linear Expansion. Let $E(f)$ be a function expression satisfying the following:

$$E(h) \equiv p_o \rightarrow q_o \; ; \; E_1(h) \text{ for all } h \varepsilon F$$

where $p_i \varepsilon F$ and $q_i \varepsilon F$ exist such that

$$E_1(p_i \rightarrow q_i \; ; \; h) \equiv p_{i+1} \rightarrow q_{i+1} \; ; \; E_1(h)$$
$$\text{for all } h \varepsilon F \text{ and } i=0,1,2,\ldots$$

and

$$E(\bar{\bot}) \equiv \bar{\bot}.$$

Then E is said to be linearly expansive with respect to these p_i's and q_i's.

Linear Expansion Theorem. Let E be linearly expansive with respect to p_i and q_i, $i=0,1,2,\ldots$. Then E is expansive with approximating functions

$$f_o \equiv \bar{\bot}$$
$$f_{i+1} \equiv p_o \rightarrow q_o \; ; \; \ldots \; ; \; p_i \rightarrow q_i \; ; \; \bar{\bot}.$$

Corollary. If E is linearly expansive with respect to p_i and q_i, $i=0,1,\ldots$, and f is the least function satisfying $f \equiv E(f)$, then

$$f \equiv p_o \rightarrow q_o \; ; \; \ldots \; ; \; p_n \rightarrow q_n \; ; \; \ldots$$

B.3 Recursion and Iteration

Backus uses three laws and the definition of linear expansion to prove a recursion theorem. A simple expansion is thus made available for many recursively defined functions. A corollary to The Recursion Theorem is then stated and proved as The Iteration Theorem. The Iteration Theorem gives an expansion for many iterative programs. Sections B.3.1 and B.3.2 state the Recursion and Iteration Theorems, respectively.

B.3.1 Recursion Theorem

Let f be a solution of

$$f \equiv p \rightarrow g;Q(f)$$

where

$Q(k) \equiv h\circ[i,k\circ j]$ for any function k

and p,g,h,i,j are any given functions. Then

$f \equiv p \rightarrow g;\ p\circ j \rightarrow Q(g);\ \ldots\ ;\ p\circ j^n \rightarrow Q^n(g);\ \ldots$

(where $Q^n(g)$ is $h\circ[i,Q^{n-1}(g)\circ j]$, and j^n is $j\circ j^{n-1}$ for $n \geq 2$) and

$Q^n(g) \equiv /h\circ[i,i\circ j,\ \ldots\ ,\ i\circ j^{n-1},g\circ j^n]$.

B.3.2 Iteration Theorem

Let f be the least solution of

$$f \equiv p \rightarrow g\ ;\ h\circ f\circ k$$

Then

$$f \equiv p \rightarrow g\ ;\ p\circ k \rightarrow h\circ g\circ k\ ;\ \ldots\ ;\ p\circ k^n \rightarrow h^n\circ g\circ k^n\ ;\ \ldots$$

B.4 Proofs for Functional Programs

The definitions and theorems stated in Sections B.2 and B.3 plus the laws stated in Section B.1 are used to prove functional programs correct. An example is given in Section B.5.

B.5 Example of a Recursive Program and its Proof

Section B.5.1 gives a detailed example of a recursive factorial function and its step-by-step application to an object. Section B.5.2 lists the correctness proof for this program. This example is taken from Backus [BACK78].

B.5.1 Recursive Factorial Function

<u>Def</u> $! \equiv eq0 \rightarrow \bar{1}\ ;\ X \circ [id, ! \circ s]$

<u>Def</u> $eq0 \equiv eq \circ [id, \bar{0}]$

<u>Def</u> $s \equiv - \circ [id, \bar{1}]$ (i.e., subtract 1)

As an example of the application and reduction of the function "!", consider the step-by-step application and reduction of the function when applied initially to the object "2". This is detailed in Table B.1. The "Justification" column lists laws and primitive operations that justify the reduction from the previous line. Let $f = !$, $p = eq0$, $q = \bar{1}$, $E(f) = X \circ [id, ! \circ s]$. Then

$$f \equiv p \rightarrow q\ ;\ E(f)$$

is the abstract form of the program.

Step Number	Function Expression	Justification
1	!:2	Apply f
2	(eq0→$\bar{1}$;X∘[id,!∘s]):2	Substitute right side
3	X∘[id,!∘s]:2	Condition when p:x=F
4	X∘<id:2,!∘s:2>	Construction
5	X:<id:2,!∘s:2>	Composition
6	X:<id:2,!:1>	s:2⇒2-$\bar{1}$=1
7	X:<2,!:1>	Apply id
8	X:<2,X∘[id,!∘s]:1>	Apply f, Substitute right side, Condition when p:x=F
9	X:<2,X:<id:1,!∘s:1>>	Construction, Composition
10	X:<2,X:<1,!∘0>>	Apply id;s
11	X:<2,X:<1,$\bar{1}$:0>>	Condition when p:x=T
12	X:<2,X:<1,1>>	Apply constant
13	X:<2,1>	Apply X
14	2	Apply X

Table B.1

Example of Application of Recursive Factorial Function

B.5.2 Proof for Recursive Factorial Function

Let f be a solution of

$$f \equiv eq0 \rightarrow \bar{1};\ X\circ[id, f\circ s]$$

where eq0 and s are defined in Section B.5.1. Then f satisfies the hypothesis of the Recursion Theorem with $p \equiv eq0$, $g\equiv\bar{1}$, $h\equiv X$, $i\equiv id$, and $j\equiv s$. Therefore f can be written

$$f\equiv eq0 \rightarrow \bar{1};\ \ldots\ ;\ eq0\circ s^{n}\rightarrow Q^{n}(\bar{1});\ \ldots$$

and

$$Q^{n}(\bar{1})\equiv /X\circ[id,\ id\circ s,\ \ldots\ ,\ id\circ s^{n-1},\ \bar{1}\circ s^{n}].$$

By III.2 and III.1 from Section B.1, respectively,

$$id\circ s^{k}\equiv s^{k}$$

and $\qquad eq0\circ s^{n} \rightarrow\rightarrow\ \bar{1}\circ s^{n}\equiv\bar{1}$

since $\qquad eq0\circ s^{n}:x \Rightarrow\ defined\circ s^{n}:x$

and $\qquad eq0\circ s^{n}:x\ \equiv\ eq0:(x-n)\equiv\ \ x=n.$

Thus, if $eq0\circ s^{n}:\ x=T$, then $x=n$ and

$$\begin{aligned} Q^{n}(\bar{1}):n &= (/X\circ[id,\ id\circ s,\ \ldots\ ,id\circ s^{n-1},\ \bar{1}\circ s^{n}]):n \\ &= /X:<n,\ n\circ s,\ \ldots\ ,\ n\circ s^{n-1},\ \bar{1}\circ s^{n}> \\ &= nX(n-1)X\ldots X(n-(n-1))\ X\ (\bar{1}:(n-n)) \\ &= n! \end{aligned}$$

Using these results for $\bar{1}\circ s^{n}$, $eq0\circ s^{n}$, and $Q^{n}(\bar{1})$ in the expansion for f, we obtain

$$f:x\ \equiv\ X=0\ \rightarrow\ 1;\ \ldots\ ;\ x=n\ \rightarrow\ nX(n-1)X\ldots X1X1;\ \ldots$$

This proves that f terminates on precisely the set of non-negative integers and represents the factorial function upon them.

HIGHLY CONCURRENT VS. CONTROL FLOW COMPUTING MODELS

by

ROBERT CLARENCE MARSHALL

B.S., University of Rochester, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

ABSTRACT

This report reviews the properties of two highly concurrent (data flow and functional) computing models, and compares them to the control flow (von Neumann) model. A highly concurrent model is one in which concurrency is designed into the model at the primitive hardware implementation level. A highly concurrent model is also implicitly concurrent, since no explicit concurrency primitives need be coded by the programmer of an implementation in order to allow concurrency. Conversely, implicitly sequential model implementations require the coding of such concurrency primitives to unlock concurrency. The properties of implicitly concurrent models are contrasted with the implicitly sequential control flow model.

The impact of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) on computing models and subsequent levels of concurrent operation is discussed. VLSI and LSI technologies are seen to be the catalysts which make highly concurrent computing systems practical. The impact on computer design is reviewed: VLSI and LSI are found to be changing the conventional views in which hardware design activities drive software and algorithm design.

The most important distinguishing property among the models presented is found to be the relative level of concurrency which the model can exhibit. The models are compared on the basis of potential (or actually exhibited) concurrency. Taxonomies are discussed as presented in the literature for the control flow model. The form for an extended taxonomy to embrace the highly concurrent models is suggested.

After concurrency, the most important property of the models is seen to be history sensitivity, or the ability to store data values internally during processing. In the control flow model, a very high level of history sensitivity is built into the model, but a very low level of concurrency is available. In the data flow and functional models, the reverse is true. History sensitivity seems to be a key property: the degree to which it is present in highly concurrent models is proportional to the applicabilities of these models. Presently, the highly concurrent models are applicable primarily only to numerical processing implementations, due to the lack of extensive internal storage capabilities.

Implementations of the highly concurrent models are reviewed, and some relevant properties of control flow implementations are discussed. Pipeline hardware structures are found to be common in data flow implementations; the single functional implementation reviewed is a binary tree structure.

In the concluding chapter an attempt is made to identify some of the potential weaknesses of the newer highly concurrent models. A common language design fallacy, which has manifested itself in recent years, is discussed, and an allegory is presented from the literature to dramatically highlight this fallacy.