Data-Driven and Demand-Driven Computer Architecture

PHILIP C. TRELEAVEN, DAVID R. BROWNBRIDGE, AND RICHARD P. HOPKINS

Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, England

Novel data-driven and demand-driven computer architectures are under development in a large number of laboratories in the United States, Japan, and Europe. These computers are not based on the traditional von Neumann organization; instead, they are attempts to identify the next generation of computer. Basically, in data-driven (e.g., data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g , reduction) computers the requirement for a result triggers the operation that will generate it.

Although there are these two distinct areas of research, each laboratory has developed its own individual model of computation, stored program representation, and machine organization. Across this spectrum of designs there is, however, a significant sharing of concepts. The aim of this paper is to identify the concepts and relationships that exist both within and between the two areas of research. It does this by examining data-driven and demand-driven architecture at three levels. computation organization, (stored) program organization, and machine organization. Finally, a survey of various novel computer architectures under development is given.

Categories and Subject Descriptors: C.0 **[Computer Systems Organization]**: General— *hardware/software interfaces; system architectures;* C.1.2 **[Processor Architecture]**: Multiple Data Stream Architectures (Multiprocessors); C.1.3 **[Processor Architecture]** Other Architecture Styles—*data-flow architectures; high-level language architectures,* D 3 2 **[Programming Languages]** Language Classifications—*data-flow languages; macro and assembly languages; very high-level languages*

General Terms: Design

Additional Key Words and Phrases Demand = driven architecture, data = driven architecture

INTRODUCTION

For more than thirty years the principles of computer architecture design have largely remained static [ORGA79], based on the von Neumann organization. These von Neumann principles include

(1) a single computing element incorporating processor, communications, and memory;
(2) linear organization of fixed-size memory cells;
(3) one-level address space of cells;
(4) low-level machine language (instructions perform simple operations on elementary operands);
(5) sequential, centralized control of computation.

Over the last few years, however, a number of novel computer architectures based on new "naturally" parallel organizations for computation have been proposed and some computers have even been built. The principal stimuli for these novel architectures have come from the pioneering work on data flow by Jack Dennis [DENN74a, DENN74b], and on reduction languages and machines by John Backus [BACK72, BACK73] and Klaus Berkling [BERK71, BERK75]. The resulting computer architecture research can be broadly classified as either data driven or demand driven. In data-driven (e.g., data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g., re-

CONTENTS

INTRODUCTION
1 BASIC CONCEPTS
1 1 Control Flow
1 2 Data Flow
1 3 Reduction
2. COMPUTATION ORGANIZATION
2 1 Classification
2 2 Control Flow
2 3 Data Flow
2 4 Reduction
2 5 Implications
3 PROGRAM ORGANIZATION
3.1 Classification
3 2 Control Flow
3 3 Data Flow
3 4 Reduction
3 5 Implications
4 MACHINE ORGANIZATION
4 1 Classification
4 2 Control Flow
4 3 Data Flow
4.4 Reduction
4 5 Implications
5 DATA-FLOW COMPUTERS
5 1 M I.T Data-Flow Computer
5 2 Texas Instruments Distributed Data Processor
5 3 Utah Data-Driven Machine (DDM1)
5 4 Irvine Data-Flow Machine
5 5 Manchester Data-Flow Computer
5 6 Toulouse LAU System
5 7 Newcastle Data-Control Flow Computer
5.8 Other Projects
6 REDUCTION COMPUTERS
6 1 GMD Reduction Machine
6 2 Newcastle Reduction Machine
6 3 North Carolina Cellular Tree Machine
6 4 Utah Applicative Multiprocessing System
6.5 S-K Reduction Machine
6 6 Cambridge SKIM Machine
6.7 Other Projects
7 FUTURE DIRECTIONS
ACKNOWLEDGMENTS
REFERENCES
BIBLIOGRAPHY

duction) computers the requirement for a result triggers the operation that will generate it.

Although the motivations and emphasis of individual research groups vary, there are basically three interacting driving forces. First, there is the desire to utilize concurrency to increase computer performance. This is based on the continuing demand from areas such as weather forecasting and wind tunnel simulation for computers with a higher performance. The natural physical laws place fundamental limitations on the performance increases obtainable from advances in technology alone. And conventional high-speed computers like CRAY 1 and ILLIAC IV seem unable to meet these demands [TREL79]. Second, there is the desire to exploit very large scale integration (VLSI) in the design of computers [SEIT79, MEAD80, TREL80b]. One effective means of employing VLSI would be parallel architectures composed of identical computing elements, each containing integral capabilities for processing, communication, and memory. Unfortunately "general-purpose" organizations for interconnecting and programming such architectures based on the von Neumann principles have not been forthcoming. Third, there is the growing interest in new classes of very high level programming languages. The most well-developed such class of languages comprises the functional languages such as LISP [MCCA62], FP [BACK78], LUCID [ASHC77], SASL [TURN79a], Id [ARVI78], and VAL [ACKE79b]. Because of the mismatch between the various principles on which these languages are based, and those of the von Neumann computer, conventional implementations tend to be inefficient.

There is growing agreement, particularly in Japan and the United Kingdom, that the next generation of computers will be based on non-von Neumann architecture. (A report [JIPD81a] by Japan's Ministry of International Trade and Industry contains a good summary of the criteria for these fifth-generation computers.) Both data-driven and demand-driven computer architecture are possible fifth-generation architectures. The question then becomes, which architectural principles and features from the various research projects will contribute to this new generation of computers?

Work on data-driven and demand-driven architecture falls into two principal research areas, namely, data flow [DENN79b, GOST79a] and reduction [BERK75]. These areas are distinguished by the way computation, stored programs, and machine re-

sources are organized. Although research groups in each area share a basic set of concepts, each group has augmented the concepts often by introducing ideas from other areas (including traditional control-flow architectures) to overcome difficulties. The aim of this paper is to identify the concepts and relationships that exist both within and between these areas of research. We start by presenting simple operational models for control flow, data flow, and reduction. Next we classify and analyze the way computation, stored programs, and machine resources are organized across the three groups. Finally, a survey of various novel computer architectures under development is given in terms of these classifications.

1. BASIC CONCEPTS

Here we present simple operational models of control flow, data flow, and reduction. In order to compare these three models we discuss each in terms of a simple machine code representation. These representations are viewed as instructions consisting of sequences of arguments—operators, literal operands, references—delimited by parentheses:

(arg0 arg1 arg2 arg3 ··· arg$n$ − 1 arg$n$).

However, the terms "instruction" and "reference" are given a considerably more general meaning than their counterparts in conventional computers. To facilitate comparisons of control flow, data flow, and reduction, simple program representations for the statement a = (b + 1) * (b − c) are used. Although this statement consists of simple operators and operands, the concepts illustrated are equally applicable to more complex operations and data structures.

1.1 Control Flow

We start by examining control flow, the most familiar model. In the control-flow program representations shown in Figure 1, the statement a = (b + 1) * (b − c) is specified by a series of instructions each consisting of an operator followed by one or more operands, which are literals or references. For instance, a dyadic operation such as + is followed by three operands; the first two, b and 1, provide the input data and the last, t1, is the reference to the shared memory cell for the result. Shared memory cells are the means by which data are passed between instructions. Each reference in Figure 1 is also shown as a unidirectional arc. Solid arcs show the access to stored data, while dotted arcs define the flow of control.

In traditional sequential (von Neumann) control flow there is a single thread of control, as in Figure 1a, which is passed from instruction to instruction. When control reaches an instruction, the operator is initially examined to determine the number and usage of the following operands. Next the input addresses are dereferenced, the operator is executed, the result is stored in a memory cell, and control is passed implicitly to the next instruction in sequence. Explicit control transfers are caused by operators such as GOTO.

There are also parallel forms of control flow [FARR79, HOPK79]. In the parallel form of control flow, shown in Figure 1b, the implicit sequential control-flow model is augmented by parallel control operators. These parallel operators allow more than one thread of control to be active at an instance and also provide means for synchronizing these threads. For example, in Figure 1b the FORK operator activates the subtraction instruction at address i2 and passes an implicit flow of control on to the addition instruction. The addition and subtraction may then be executed in parallel. When the addition finishes execution, control is passed via the GOTO i3 instruction to the JOIN instruction. The task of the JOIN is to synchronize the two threads of control that are released by the addition and subtraction instruction, and release a single thread to activate the multiply instruction.

In the second parallel form of control flow, shown in Figure 1c, each instruction explicitly specifies its successor instructions. Such a reference, i1/0, defines the specific instruction and argument position for the control signal, or *control token*. Argument positions, one for each control signal required, are represented by empty bracket symbols ( ), and an instruction is

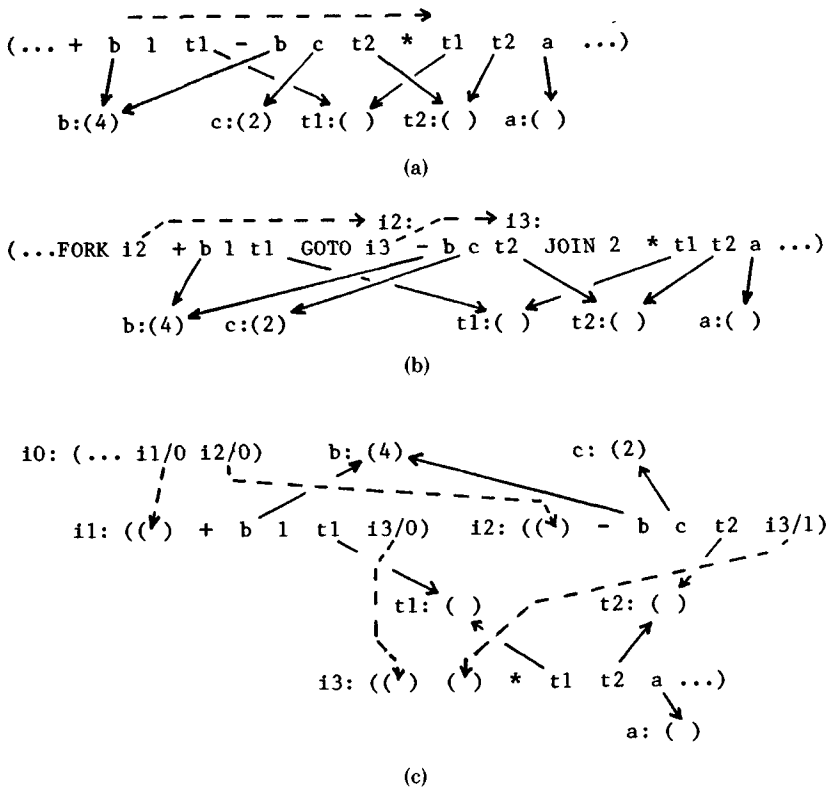

Figure 1. Control-flow programs for a = (b + 1) * (b − c): (a) sequential, (b) parallel "FORK-JOIN"; (c) parallel "control tokens."

executed when it has received the required control tokens. The two parallel forms of control flow, illustrated by Figures 1b and 1c, are semantically equivalent; FORKS are equivalent to multiple successor instruction addresses and JOINs are equivalent to multiple empty bracket arguments.

The sequential and parallel control-flow models have a number of common features: (1) data are passed indirectly between instructions via references to shared memory cells; (2) literals may be stored in instructions, which can be viewed as an optimization of using a reference to access the literal; (3) flow of control is implicitly sequential, but explicit control operators can be used for parallelism, etc.; and (4) because the flows of data and control are separate, they can be made identical or distinct.

1.2 Data Flow

Data flow is very similar to the second form of parallel control flow with instructions activated by tokens and the requirement for tokens being the indicated ( ) symbols. Data-flows programs are usually described in terms of directed graphs, used to illustrate the flow of data between instructions. In the data-flow program representation shown in Figure 2, each instruction consists of an operator, two inputs which are either literal operands or "unknown" operands defined by empty bracket ( ) symbols, and a reference, i3/1, defining the specific instruction and argument position for the result. A reference, also shown as a unidirectional arc, is used by the producer instruction to store a *data token* (i.e., result) into the consumer. Thus data are passed directly between instructions.

An instruction is enabled for execution when all arguments are known, that is, when all unknowns have been replaced by partial results made available by other instructions. The operator then executes, removing the inputs from storage, processing them according to the specified operation,

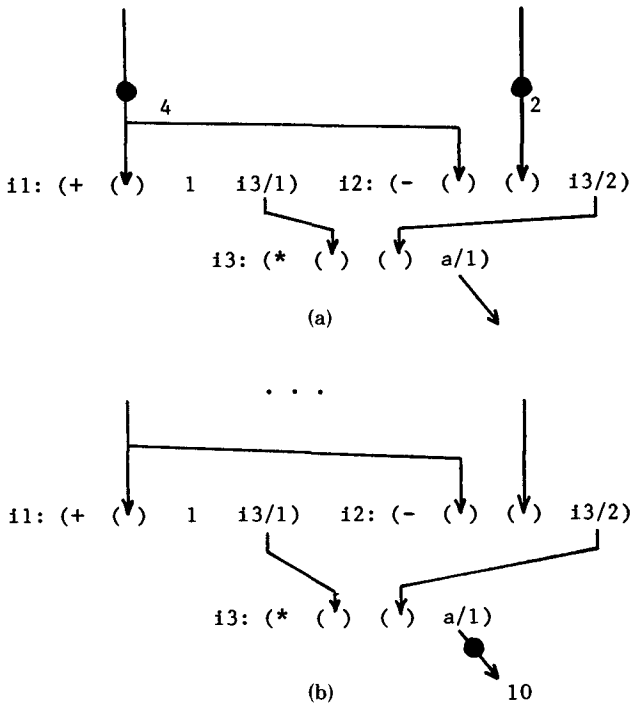

Figure 2. Data-flow program for a = (b + 1) * (b − c) (a) Stage 1; (b) Stage 4.

and using the embedded reference to store the result at an unknown operand in a successor instruction. In terms of directed graphs, an instruction is enabled when a data token is present on each of its input arcs. During execution the operator removes one data token from each input arc and releases a set of result tokens onto the output arcs.

Figure 2 illustrates the sequence of execution for the program fragment a = (b + 1) * (b − c), using a black dot on an arc to indicate the presence of a data token. The two black dots at Stage 1 in Figure 2 indicate that the data tokens corresponding to the values of b and c have been generated by predecessor instructions. Since b is required as input for two subsequent instructions, two copies of the token are generated and stored into the respective locations in each instruction. The availability of these inputs completes both the addition and the subtraction instruction, and enables their operators for execution. Executing completely independently, each operator consumes its input tokens and stores its result into the multiplication instruction "i3." This enables the multiplication, which executes and stores its result corresponding to the identifier "a," shown at Stage 4.

In the data-flow model there are a number of interesting features: (1) partial results are passed directly as data tokens between instructions; (2) literals may be embedded in an instruction that can be viewed as an optimization of the data token mechanism; (3) execution uses up data tokens—the values are no longer available as inputs to this or any other instruction; (4) there is no concept of shared data storage as embodied in the traditional notion of a variable; and (5) sequencing constraints—flows of control—are tied to the flow of data.

1.3 Reduction

Control-flow and data-flow programs are built from fixed-size instructions whose arguments are primitive operators and operands. Higher level program structures are built from linear sequences of these primitive instructions.

In contrast, reduction programs are built from nested expressions. The nearest analogy to an "instruction" in reduction is a function application, consisting of ⟨function⟩ ⟨argument⟩, which returns its result in place. Here a ⟨function⟩ or ⟨argument⟩ is recursively defined to be either an atom, such as + or 1, or an expression. Likewise, a reference may access, and function application may return, either an atom or an expression. Higher level program structures are reflected in this machine representation, being themselves function applications built from more primitive functions. In reduction, a program is mathematically equivalent to its result in the same way that the expression 3 + 3 is equivalent to the number 6. Demanding the result of the *definition* "a," where a = (b + 1) * (b − c), means that the embedded reference to "a" is to be rewritten in a simpler form. (It may be helpful for the reader to view this evaluation of a reference as calling the corresponding definition, giving reduction a CALL-RETURN pattern of control.) Because of these attributes, only one definition of "a" may occur in a program, and all references to it give the same value, a property known as *referential transparency*. There are two forms of reduction, differentiated in the way that arguments in a program are manipulated, called string reduction and graph reduction.

The basis of string reduction is that each instruction that accesses a particular definition will take and manipulate a separate copy of the definition. Figure 3 illustrates string manipulation for a reduction execution sequence involving the definition a = (b + 1) * (b − c). Each instruction consists of an operator followed by literals or embedded references, which are used to demand its input operands. At Stage 1 in Figure 3 some instruction, containing the reference "a," demands the value corresponding to the definition "a." This causes a copy of the definition to be loaded into the instruction overwriting the reference "a," as also shown in Figure 3. Next the multiplication operator demands the values corresponding to i1 and i2, causing them to be overwritten by copies of their definitions. The multiplication then suspends and the addition and subtraction operators demand the values of b and c. The substitution of the values 4 and 2 is shown at Stage 3 in Figure 3. The reducible subexpressions (+ 4 1) and (− 4 2) are then rewritten, causing the multiplication to be reenabled. Finally at Stage 5 the multiplication is replaced by the constant 10, which is the value of "a."

The basis of graph reduction is that each instruction that accesses a particular definition will manipulate references to the definition. That is, graph manipulation is based on the sharing of arguments using pointers. Figure 4 illustrates graph reduction using the same program definition a = (b + 1) * (b − c) as above. At Stage 1 in Figure 4 an instruction demands the value corresponding to "a," but instead of a copy of the definition being taken, the reference is traversed in order to reduce the definition and return with the actual value. One of the ways of identifying the original source of the demand for "a," and thus supporting the return, is to reverse the arcs (as shown in Figure 4) by inserting a source reference in the definition.

This traversal of the definition and the reversal of the references is continued until constant arguments, such as b and c in Figure 4, are encountered. In Figure 4, reduction of the subexpressions in the definition starts with the rewriting of the addition and the subtraction as shown at Stage 4. This proceeds until the value of "a" is calculated and a copy is returned to the instruction originally demanding "a." (If there are no further references to b, c, i1, and i2, then they can be "garbage collected.") Any subsequent requests for the value of "a" will immediately receive the constant 10—one of the major benefits of graph reduction over string reduction.

In reduction the main points to note are that: (1) program structures, instructions, and arguments are all expressions; (2) there is no concept of updatable storage such as a variable; (3) there are no additional sequencing constraints over and above those implied by demands for operands; and (4) demands may return both simple or complex arguments such as a function (as input to a higher order function).

Control flow, data flow, and reduction clearly have fundamental differences,

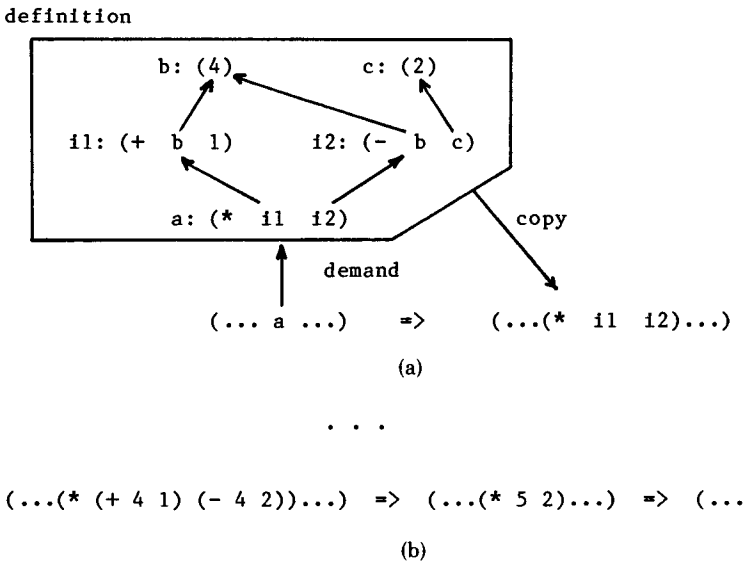

Figure 3. String reduction program for a = (b + 1) * (b − c) (a) Stages 1 and 3, (b) Stages 3–5.

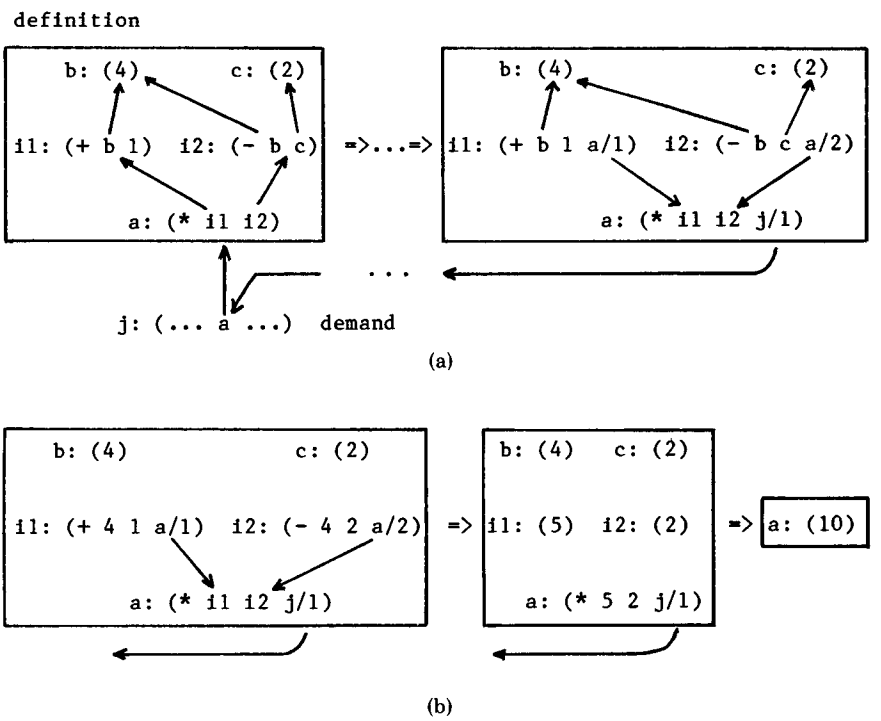

Figure 4. Graph reduction program for a = (b + 1) * (b − c): (a) Stages 1 and 3, (b) Stages 4–6.

which relate to their advantages and disadvantages for representing programs. However, they also have interesting underlying similarities. In the next two sections on computation organization and program organization we attempt to identify and classify these underlying concepts.

2. COMPUTATION ORGANIZATION

In this section we examine, at an abstract level, the way computation progresses. This progress takes the form of successive changes in the state of the computation brought about by executing instructions. Computation organization describes how these state changes come to take place by describing the sequencing and the effect of instructions. We describe the rules determining which instructions are selected for execution and how far reaching the effects of their execution may be.

This abstract level of classification enables a clear distinction to be drawn between the terms: control driven, data driven, and demand driven. These three classes are often identified with, respectively, the operational models control flow, data flow, and reduction. Here we define the notions of control-driven, data-driven, and demand-driven computation organizations and identify their relationships to the three operational models.

2.1 Classification

Computation organizations may be classified by considering computation to be a continuous repetition of three phases: select, examine, and execute. It needs to be emphasized that this description does not necessarily reflect the way in which particular computer implementations operate but rather that it is a logical description of the affects achieved.

(1) *Select.* At the select phase a set of instructions is chosen for possible execution. The rule for making this choice is called a *computation rule.* The computation rule selects a subset of instructions in the program. Only instructions chosen by the select phase may be executed, but selection does not guarantee execution. Three of the computational rules used in this classification are imperative, innermost, and outermost. The imperative computation rule selects the instructions indicated by, for example, a special ("program counter") register or the presence of control tokens. This selection is made regardless of the position of the instruction in the program structure. Innermost and outermost computation rules select, respectively, the instructions most deeply nested and least deeply nested in the program structure. An innermost instruction has no instructions as arguments to it (only values). An outermost instruction is not nested as an argument of any other instruction. The instructions selected by the three rules are illustrated in Figure 5.

(2) *Examine.* At the examine phase, each of the instructions previously chosen in the select phase is examined to see if it is executable. The decision is based on examination of each instruction's actual arguments. The rule for making this decision is called a *firing rule.* For instance, the firing rule may require all operands to be data values, or it may require only one operand to be a value as, for example, in a conditional. If an instruction is executable, it is passed on to the next phase for execution; otherwise, the examine phase may take some action, such as delaying the instruction or attempting to coerce arguments so as to allow execution.

(3) *Execute.* At the execute or "target" phase, which is broadly similar in all computation organizations, instructions are actually executed. The result of execution is to change the state of the computer. Results are made available and are passed to other parts of the program. Execution may produce globally perceived changes, perhaps by changing the state of a globally shared memory, or it may produce localized changes as when an expression is replaced by its value.

2.2 Control Flow

The select phase of control-flow computation corresponds to the fetch part of the fetch–execute control cycle. Each control-flow computing element has a program counter naming the next instruction to execute. In the select phase, the program

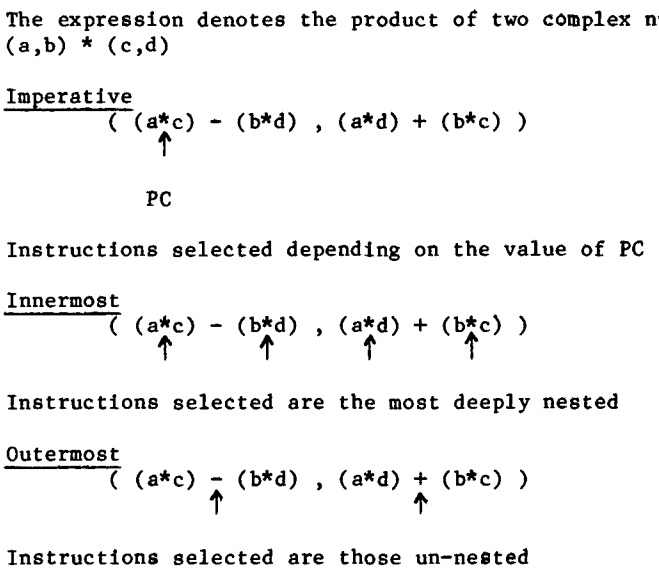

Figure 5. Three computation rules applied to an expression.

counter is used to choose the instructions to be used. Once chosen by select, instructions are not checked by an examine phase, but are automatically passed on to execution. The execute phase of control-flow instructions is allowed to change any part of the state. Control flow uses a shared memory to communicate results. The state of computation is represented by the contents of this shared memory and of the program counter register(s). A program counter is updated at the end of each cycle either implicitly or, in the case of GOTOs, explicitly.

We define the term *control driven* to denote computation organizations in which instructions are executed as soon as they are selected. The select phase alone determines which instructions are to be executed. For all computation organizations in this class the examine phase is redundant, and instruction sequencing is independent of program structure.

2.3 Data Flow

There are many varieties of data-flow computers; here we restrict ourselves to "pure" data-flow computation as described in Section 1.2. In pure data flow, instructions are executed as soon as *all* their arguments are available. Logically at least, each instruction has a computing element allocated to it continuously, just waiting for arguments to arrive. So the select phase of data-flow computation may be viewed as logically allocating a computing element to every instruction. The examine phase implements the data-flow firing rule, which requires all arguments to be available before execution can take place. Arguments must be data items, not unevaluated expressions. If values are not yet available, the computing element will not try to execute the instruction but will remain dormant during the execute phase. The execute phase in data flow changes a local state consisting of the executing instruction and its set of successor instructions. The instruction consumes its arguments and places a result in each successor instruction.

We define the term *data driven* to denote computation organizations where instructions passively wait for some combination of their arguments to become available. This implies a select phase, which (logically) allocates computing elements to all instructions, and an examine phase, which suspends nonexecutable instructions. In data-driven computation organizations, the key factor governing execution is the availability of data. For this reason "data driven" is the same as "availability driven."

2.4 Reduction

Reduction computers each have different rules embodied in their select phase. The choice of computation rule is a design choice for a particular reduction computer. The commonest rules used are *innermost* and *outermost* (see Figure 5), and in fact the discussion of reduction in Section 1 was restricted to outermost reduction. The computation rule in a reduction computer determines the allocation of computing elements at the beginning of each computation cycle. In the examine phase the arguments are examined to see whether execution is possible. If it is, the instruction is executed. Otherwise, the computing element tries to coerce the arguments into the required pattern. This coercion demands the evaluation of argument(s) until sufficient are available for execution. Logically, this demand consists of spawning one or more subcomputations to evaluate operands and waiting for them to return with a value. The instruction set of a reduction computer may contain many different firing rules, each instruction having the rule most suited to it. For example, all arithmetic operations will have a firing rule that forces their arguments to be values. The execute phase in a reduction machine involves rewriting an instruction in situ. The instruction is replaced by its result where it stands. Only the local state consisting of the instruction itself and those instructions that use its results are changed. Execution may thus also enable another instruction.

We define the term *demand driven* to denote a computation organization where instructions are only selected when the value they produce is needed by another, already selected instruction. All outermost reduction architectures fall into this category but innermost reduction architectures do not. The essence of a demand-driven computation organization is that an instruction is executed only when its result is demanded by some other instruction and the arguments may be recursively evaluated where necessary. In reduction computers with an innermost computation rule, instructions are never chosen by select until their arguments are available. This restriction means that all arguments reaching the examine stage are preevaluated and hence no coercions need ever take place. It also means that all instructions have all their arguments evaluated whether or not this is necessary, exactly as occurs in data flow. Thus we believe innermost computation organizations are data driven.

2.5 Implications

The implications of the computation organization classification can now be summarized. Control-flow computers have a control-driven computation organization; instructions are arbitrarily selected, and once selected they are immediately executed. Data-flow computers have a data-driven computation organization; all instructions are in principle active, but only execute when their arguments become available. Some reduction computers are demand driven and some are data driven.

Control-flow computers all have a control-driven computation organization. The control-driven organization is characterized by the lack of an examine stage, and by a computation rule that selects instructions independently of their place in the program's structure. This implies that the program has complete control over instruction sequencing. Once selected, instructions will always be executed regardless of the state of their operands. There is no wait for arguments, or demand for arguments, apart from the dereferencing of an address. It is up to the programmer to ensure that arguments are set up before control reaches an instruction. The advantage of control-driven computation is full control over sequencing. The corresponding disadvantage is the programming discipline needed to avoid run-time errors. These errors are harder to prevent and detect than exceptions (overflow, etc.), which occur at the execute phase in all computation organizations. A typical example of the twin generalities and dangers of control-driven computation organization is the ability to execute data as a program.

Data-flow computers have a data-driven computation organization that is characterized by a passive examine stage. Instructions are examined, and if they do not pass the firing rule, no action is taken to force

them to become executable. The data-flow firing rule requires all arguments to arrive before an instruction will execute. However, some data-flow implementations have found this too restrictive and have added non-data-driven instructions to provide some degree of explicit control. The advantage of data-driven computation is that instructions are executed as soon as their arguments are available, giving a high degree of implicit parallelism. The disadvantages are that instructions may waste time waiting for unneeded arguments. This becomes increasingly apparent when the implementation of data-flow procedures is considered. In addition, operators such as an if-then-else operator, which will use only two of its three arguments, discarding the other, will always be forced to wait for all three. In the worst case this can lead to nontermination through waiting for an unneeded argument, which is, for example, an infinite iteration.

A reduction computer having a demand-driven organization is characterized by an outermost computation rule coupled with the ability to coerce arguments at the examine stage. Instruction sequencing is driven by the need to produce a result at the outermost level, rather than to insist on following a set pattern. Each instruction chosen by the outermost select can decide to demand further instructions. Instructions actively coerce their arguments to the required form if they are not already in it. Reduction computers not possessing (1) an outermost select and (2) a coercing examine phase cannot be classified as demand driven. The advantage of the demand-driven computation organization is that only instructions whose result is needed are executed. A procedure-calling mechanism is built in, by allowing the operator of an instruction to be defined as a block of instructions. The disadvantage of demand-driven computation is in processing, say, arithmetic expressions, where every instruction (+, *, etc.) always contributes to the final result. Propagating demand from outermost to innermost is wasted effort; only operator precedence will determine sequencing, and every instruction must be activated. In these cases, data-driven computation organization is better since the sequencing is determined solely by operator priorities. Demand driven is superior only for "nonstrict" operators such as "if-then-else," which do not require all their arguments.

Last, the execute phase of any computation organization has important consequences for the underlying implementation. Global changes may have far-reaching effects, visible throughout the computer. Local changes can only alter the state of a small part of the computation. To support a computation organization allowing global state changes, some form of global communications between instructions is required. On the other hand, if only local changes are to be supported, this locality can be exploited in a distributed architecture. In general, data-flow and reduction programs are free from side effects, another feature making them suitable for distributed implementation.

3. PROGRAM ORGANIZATION

We use the term program organization to cover the way machine code programs are represented and executed in a computer architecture. This section starts by classifying the underlying mechanisms of program organization for control-flow, data-flow, and reduction models.

3.1 Classification

Two computational mechanisms, which we refer to as the data mechanism and the control mechanism, seem fundamental to these three groups of models. The *data mechanism* defines the way a particular argument is used by a number of instructions. There are three subclasses:

(1) *by literal*—where an argument is known at compile time and a separate copy is placed in each accessing instruction (found in all the operational models);
(2) *by value*—where an argument, generated at run time, is shared by replicating it and giving a separate copy to each accessing instruction, this copy being stored as a value in the instruction (as seen in data flow and string reduction);

Control Mechanisms	Data Mechanisms: by value (& literal)	Data Mechanisms: by reference (& literal)
sequential		von Neumann control flow
parallel	data flow	parallel control flow
recursive	string reduction	graph reduction

Figure 6. Computational models: control and data mechanisms.

(3) *by reference*—where an argument is shared by having a reference to it stored in each accessing instruction (as seen in control flow and graph reduction).

The *control mechanism* defines how one instruction causes the execution of another instruction, and thus the pattern of control within the total program. There are again three subclasses:

(1) *sequential*—where a single thread of control signals an instruction to execute and passes from one instruction to another (as seen in traditional sequential control flow);
(2) *parallel*—where control signals the availability of arguments and an instruction is executed when all its arguments (e.g., input data) are available (as seen in data flow and parallel control flow);
(3) *recursive*—where control signals the need for arguments and an instruction is executed when one of the output arguments it generates is required by the invoking instruction. Having executed, it returns control to the invoking instruction (as seen in string reduction and graph reduction).

The relationship of these data and control mechanisms to the three groups of operational model is summarized in Figure 6. Using this classification as a basis, we now examine the advantages and disadvantages for program representation and execution of control flow, data flow, and reduction.

3.2 Control Flow

Control flow is based on a "sequential" or "parallel" control mechanism. Flow of control is implicitly sequential with explicit sequential and parallel patterns of control being obtained from, respectively, GOTO and FORK-JOIN style control operators. The basic data mechanism of control flow is a by-reference mechanism, with references embedded in instructions being used to access shared memory cells. This form of data sharing is *shared update*, in which the effects of changing the contents of a memory cell are immediately available to other users.

In computers based on parallel program organizations such as parallel control flow, special precautions must be taken in a program's representation (style of machine code generated) to ensure that the natural asynchronous execution does not lead to unwanted indeterminacy. This is basically a problem of synchronizing the usage of shared resources, such as a memory cell containing an instruction or data. It is appropriate to examine the support in parallel control-flow computers of two important programming mechanisms—iteration and procedures—because they illustrate how these synchronization problems are controlled and also the style of program representation used.

Iteration becomes a potential problem for parallel control flow because program fragments with loops may lead to logically cyclic graphs in which each successive it-

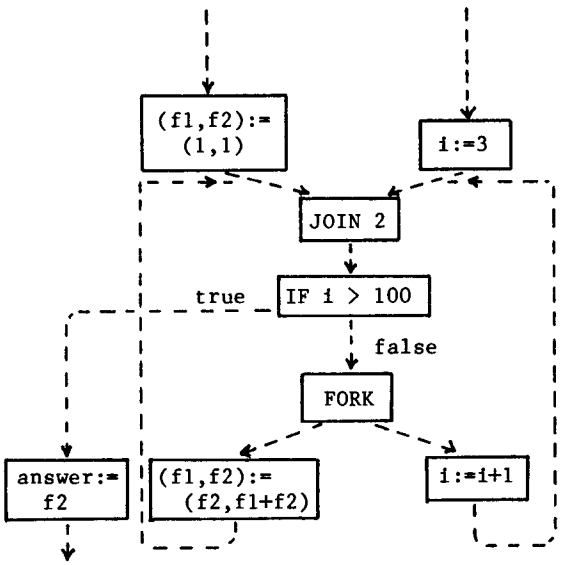

Figure 7. Control-flow iteration using feedback.

eration of a loop could execute concurrently, giving the possibility, for instance, of multiple-data items being stored in the same memory cell. Two possible schemes may, in general, be used to control potentially concurrent iteration. The first uses the feedback of control to synchronize reference usage and the second represents iteration by the equivalent recursion, thereby creating unique contexts for references.

To illustrate these two schemes for representing iteration, we use as an example a program fragment that calculates the one-hundredth number in the Fibonacci series:

```
(f1, f2) := (1, 1);
FOR i = 3 TO 100 DO
  (f1, f2) := (f2, f1 + f2) OD;
answer := f2;
```

This fragment, using concurrent assignment, consists of two calculations, one producing the Fibonacci series as successive values of f2, and the other incrementing the iteration count i. Since i is not used within the DO . . . OD, these two calculations may execute in parallel.

The first scheme for supporting iteration based on the feedback of control to synchronize resource usage is shown in Figure 7. This ensures that only a single copy of an instruction can be active or that a single data item may occupy a memory cell, at an instant. This synchronization is achieved by the JOIN instruction. Next the IF instruction, if false, performs a new iteration or, if true, transfers the value of f2 to memory cell "answer." Since memory cells are continually updated in this iteration scheme, it may be necessary in specific implementations to execute the concurrent assignment (f1, f2) := (f2, f1 + f2) sequentially to exclude indeterminacy. The second iteration scheme makes use of the procedure mechanism to provide separate contexts for each iteration, by transforming the iterative program into the equivalent recursion:

```
fib(f1, f2, i) := IF i > 100
                  THEN f2
                  ELSE fib(f2, f1 + f2, i + 1) FI;
answer := fib(1, 1, 3);
```

Each time a new call of the function fib is made, a new process, with a separate context, is created.

At a logical level there are two instructions involved in procedure invocation. (Figure 8 illustrates this procedure mechanism.) In a calling process P1, there is a CALL instruction that first obtains a new (globally unique) process identifier P2 and then changes the context of the input parameters from P1 to the new context P2. At the end of the called procedure, there must be a RETURN instruction that changes the context of the computed results back to

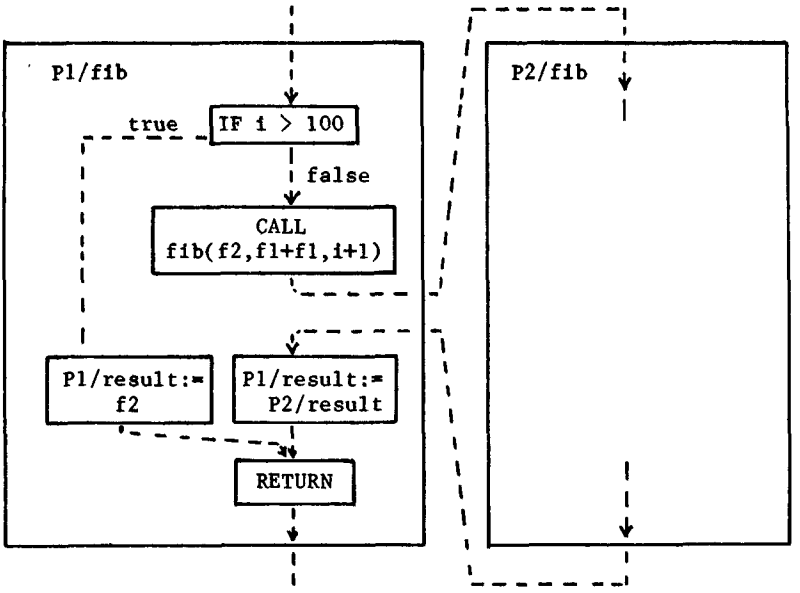

Figure 8. Control-flow iteration using recursion.

the calling context P1. To achieve this, the CALL instruction must pass the caller's process identifier P1 to the RETURN. When all the results have been returned to the calling process, the called process P2 is deleted by the RETURN instruction.

3.3 Data Flow

Data flow is based on a by-value data mechanism and a parallel control mechanism, supported by data tokens. Thus flows of data and control are identical in data flow. A data token is used to pass a copy of a partial result directly from the producer to the consumer instruction. This form of data sharing is that of *independent copies*, in which the effect of a consumer instruction accessing the contents of a received data token is hidden from other instructions.

When an instruction is executed, the role of an embedded reference is to specify the consumer instruction and argument position for a data token. In terms of directed graphs, the role of the reference is to provide a "name" that identifies uniquely a particular data token generated by a program at an instant, by specifying the arc on which it is traveling and the node to which it is destined. Unfortunately the two-field (instruction/argument position) name format does not provide such uniqueness. For instance, more than one copy of a particular instruction may be executing in parallel. Thus tokens are no longer uniquely named, leading to the possibility of tokens being inserted in the wrong instruction. To distinguish the separate contexts of instances of a procedure, an additional "process" field is logically appended to a reference. In summary, the basic format of a reference is [ARVI77a, TREL78]

• ⟨P / N / A⟩
process
instruction (node)
argument (arc)

and the fields are used for the following:

(1) The process (P) field distinguishes separate instances of an instruction N that may be executing in parallel, either within a single program or within distinct programs.
(2) The instruction (N) field identifies the consuming instruction to which the data token is being passed.
(3) The argument (A) field identifies in which argument position in the instruction N the token is to be stored.

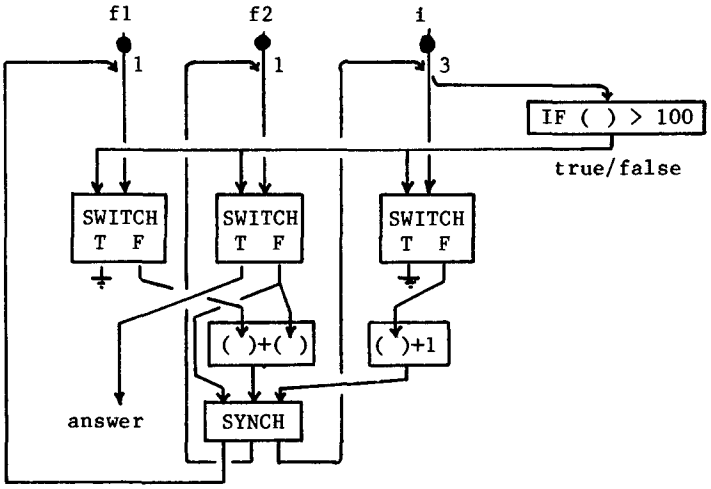

Figure 9. Data-flow iteration using feedback of tokens.

In the machine code of a data-flow computer, the values of the N and the A fields are usually statically embedded in the code at compile time, whereas the value of the P is dynamically generated at run time by the system.

Recall in our discussion of parallel control flow that special precautions need to be taken in the style of machine code generated for a program to exclude unwanted indeterminacy. Similar precautions must be taken for data flow. Here we examine data-flow program representations for iteration and procedures, using the two schemes for iteration previously discussed for control flow. Again, the program fragment that calculates the one-hundredth number in the Fibonacci series is used for examples. It is interesting to compare these examples with those previously given for control flow.

The first scheme for supporting iteration is illustrated by Figure 9. Here the feedback of data tokens synchronizes the usage of references, thereby ensuring that only a single data token can ever be on a logical arc at an instant. At the start of each iteration the IF instruction releases a true/false data token, a copy of which is passed to each SWITCH. A SWITCH takes two types of inputs: one being a true/false token, which selects either the true or the false outputs, and the other the data token to be switched. If the token is false, the other tokens are fed into the iteration, whereas a true token causes the data token corresponding to f2 to be routed to answer and the other tokens to be discarded, as shown by the "earth" symbols. To ensure that all calculations within the loop have terminated before feeding back the tokens into the next iteration, a SYNCHronizer instruction is used. The SYNCH fires when all inputs are present and releases them onto the corresponding output arcs.

The second scheme for supporting iteration is based on a data-flow procedure mechanism [ARVI78, MIRA77, HOPK79], which allows concurrent invocations of a single procedure through the use of the process (P) field. This mechanism is essential to provide distinct naming contexts for each procedure invocation, thus isolating the data tokens from those belonging to any other invocation. The second iteration scheme that represents iteration by the equivalent recursion is shown in Figure 10. In this example parallelism is only obtained when a new invocation of the procedure fib is to be used, by calculating the value parameters (f2, f1 + f2, i + 1) concurrently. As in the control-flow procedure mechanism, discussed above, the CALL instruction creates a new process and inserts the parameters, and the RETURN changes back the context of the results and deletes the invoked process.

There is, in fact, a third scheme for supporting iteration in use in data-flow com-

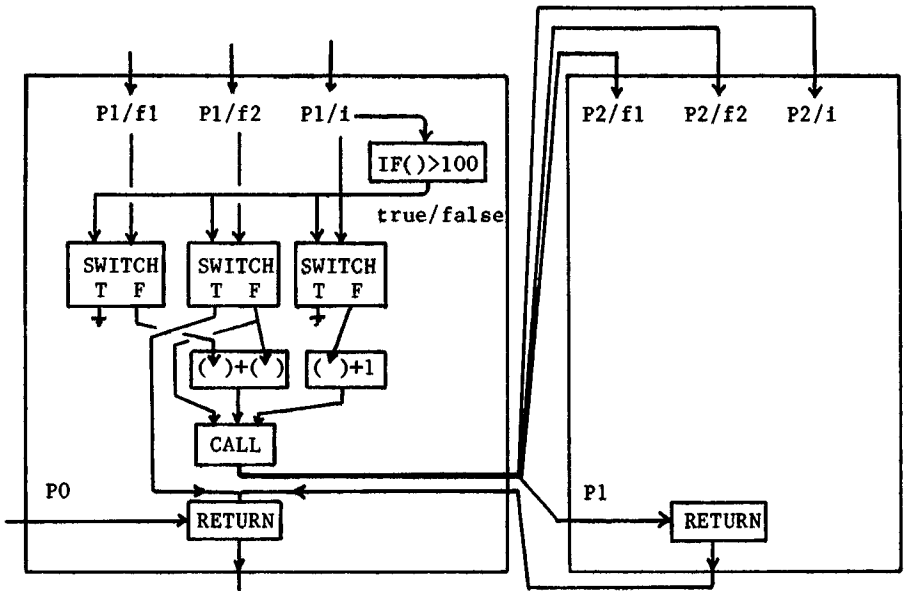

Figure 10. Data-flow iteration using recursion.

puters, based on an additional *iteration number* field [ARVI77a, TREL78] in each reference, for example, P/N/A/I. This iteration number field distinguishes individual data tokens, logically flowing on a particular arc, by giving each token a unique I value, for example, 1, 2, 3, ... Using this third scheme for the Fibonacci example, basically three sequences of data tokens would be generated: f2/1, f2/2, ...; f1/1, f1/2, ...; and i/1, i/2, ... Some of the data-flow computer designs [ARVI77a, WATS79] support this concept of an iteration number field, but the field is only directly applicable for a single level of iteration. Using only iteration numbers for nested iterations such as

```
FOR x = 1 TO 3 DO
  FOR y = 1 TO 3 DO
    FOR z = 1 TO 3 DO
      ...N...
    OD
  OD
OD
```

it would be necessary to provide three iteration number fields in a reference to give unique names for the 27 tokens for, say, argument A of N, that is, P/N/A/1/1/1 ... P/N/A/1/2/3 ... P/N/A/3/3/3. (In fact, this case can be avoided by treating each FOR ... OD as a procedure with a unique process number and using a single iteration number for its one level of internal iteration.)

3.4 Reduction

Reduction is based on a recursive control mechanism and either a by-value or a by-reference data mechanism. String reduction has a by-value data mechanism, and graph reduction has a by-reference data mechanism. Reduction programs are essentially expressions that are rewritten in the course of execution. They are built (as described in Section 1.3) from functions applied to arguments. Recall that both functions and arguments can be simple values or subexpressions. In string reduction, copies of expressions are reduced. In graph reduction, subexpressions are shared using references. Referential transparency (see Section 1.3) means that a reduction program will give the same result whether data are copied or shared. Below, string reduction and graph reduction are described in more detail. Note that because reduction is inherently recursive, we only show a recursive version of the Fibonacci program. In reduction programs iteration is represented as tail recursion.

```
Initial Expression:

  ( answer ) WHERE
                    answer = fib (1, 1, 3);
          fib (f1, f2, i) = IF i > 100 THEN f2
                                       ELSE fib (f2, f1+f2, i+1) FI;
First Reduction:

  ( IF 3 > 100 THEN 1
               ELSE fib (1, 1+1, 3+1) FI )

Next Reductions:

  ( IF FALSE THEN 1
             ELSE fib (1, 1+1, 3+1) FI )

  ( fib (1, 1+1, 3+1) )

  ( fib (1, 2, 4) )

  ( fib (2, 3, 5) ) ... ( fib (3, 5, 6) ) ... ( fib (5, 8, 7) )
```

Figure 11. String reduction of Fibonacci program.

String reduction programs are expressions containing literals and values. They are conveniently represented as a bracketed expression, with parentheses indicating nesting, as shown in Figure 11. In Figure 11 the initial expression (answer) is first reduced using the definition fib(f1, f2, i), with f1, f2, and i replaced by 1, 1, and 3. The next reduction evaluates $3 > 100$, giving (IF FALSE ...) followed by (fib(1, 1 + 1, 3 + 1)) and so on. Execution terminates with the original expression, answer, being replaced by the one-hundredth Fibonacci number. Because the form of this function is tail recursive, its execution behaves like iteration; the final result is passed out directly to the original computation, and no intermediate values are preserved.

In Figure 11, an innermost computation rule (Section 2.1) was used, forcing all arguments to be evaluated before being substituted into definitions. If another rule were chosen, a different sequence of reductions would occur before the same answer was found. In string reduction, because a by-value data mechanism is used, separate copies of actual arguments are generated for each formal parameter occurrence. This may increase parallelism, in the sense that many processors can work simultaneously on their own copies of subexpressions. But most of this work may be needlessly duplicated effort as in the example above. For this reason, we conclude that string manipulation is best suited to innermost computation rules where functions are only applied to already evaluated arguments. In this case work will not be duplicated.

Graph reduction programs are expressions containing literals, values, and references. In graph reduction, parameters are substituted by reference into the body of a defined function. For simplicity we assume that the substitution occurs automatically when a definition is dereferenced. In practice, a special mechanism such as lambda substitution or combinators [TURN79a] is used to achieve this formal-to-actual parameter binding. Because of the by-reference mechanism, in Figure 12, it is more suitable to use a graphic notation to represent the Fibonacci program. Nodes of the graph represent a function and its arguments, while arcs represent references and structuring.

Figure 12 shows a graph reduction program for Fibonacci. This example uses a parallel outermost computation rule. The loss of efficiency that can occur with outermost string reduction does not occur here because graph reduction permits sharing of expressions. If an innermost computation rule had been used, no real use would have been made of the graph reduction's by-ref-

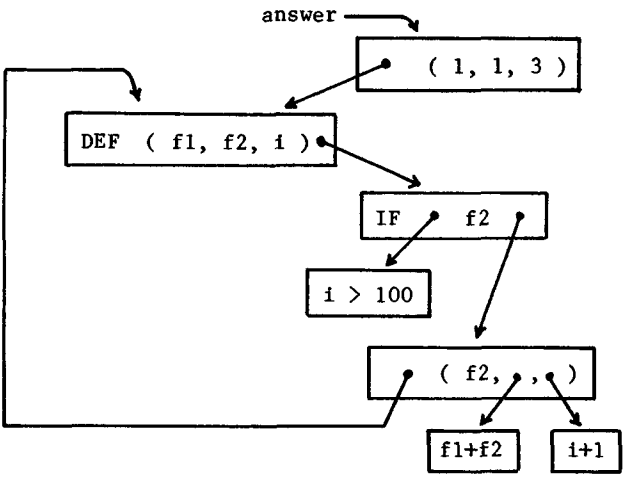

Figure 12. Graph reduction of Fibonacci program.

erence data mechanism. This is because all subexpressions would be reduced before the functions referring to them. Thus the only references in functions would be to values, and there would be no sharing of subexpressions. For this reason, graph reduction is suited to outermost computation rules.

3.5 Implications

Control-flow program organizations, owing to the separation of flows of control from flows of data and the way operands are accessed, tend to be less efficient than, say, data flow when evaluating simple expressions. For example, to pass the partial result of a subexpression to the enclosing expression requires three operations—store result, send control flow, load result—in control flow, but only one operation—send data token—in data flow. However, control flow has advantages when manipulating data structures that are to be manipulated in place, or where a specific pattern of control is required, as, for instance, with conditional evaluation of alternatives. In addition, since instructions have both input and output references, the pattern of data accesses is unconstrained, with execution manipulating a global state composed of all the memory cells of a program. As a general-purpose program organization control flow is surprisingly flexible, particularly with respect to memory changes, interaction, and complex control structures. The criticisms of control-flow organizations have been well documented by Backus [BACK78]. Basically, they lack useful mathematical properties for reasoning about programs, parallelism is in some respect bolted on, they are built on low-level concepts, and there is a major separation between the representation and execution of simple instructions and of procedures and functions.

The major advantage of data flow is the simplicity and the highly parallel nature of its program organization. This results from the data token scheme combining both the by-value data mechanism and the parallel control mechanism. The data-flow program organization is very efficient for the evaluation of simple expressions and the support of procedures and functions with call-by-value parameters. However, where shared data structures are to be manipulated in place or where specific patterns of control are required, such as sequential or conditional, data flow seems at a disadvantage. Implementation of data-flow program organizations often separate the storage for data tokens and instructions, which makes compilation at least conceptually difficult. Thus as a general-purpose program organization pure data flow is questionable, but for more specialist applications like process control or even robotics it may be highly suitable [JIPD81c].

String and graph reduction are both notable for providing efficient support for

functional programming, which is growing in interest. Graph reduction has a by-reference data mechanism that allows sharing and allows manipulation of unevaluated objects. String reduction has a by-value data mechanism and so has minimal addressing overheads. The nature of functional programs makes them suitable for parallel evaluation; referential transparency makes reductions independent of context and sequencing. Graph manipulation allows arbitrary objects to be manipulated without their being evaluated. This means that infinite data structures can conceptually be used as long as only the values of some finite part of them are demanded.

In graph reduction, structures are represented by a reference until their contents are needed; because references are generally smaller than structures, they are more efficient to manipulate. In string reduction, structures are represented by value, and so their contents are duplicated at many points in the program; thus their contents are available locally, without a referenced value being fetched from elsewhere. Last, for reduction program organizations to become candidates for general-purpose computing, it is necessary for functional programming language to become the most widely used style of programming.

4. MACHINE ORGANIZATION

We use the term machine organization to cover the way a machine's resources are configured and allocated to support a program organization. This section starts by classifying the machine organizations being used in data-driven and demand-driven computers.

4.1 Classification

An examination of the data- and demand-driven computer architectures under development reveals three basic classes of machine organization, which we call centralized, packet communication, and expression manipulation.

(1) *Centralized.* Centralized machine organization consists of a single processor, communication, and memory resource, as shown in Figure 13. It views an executing program as having a single active instruction, which passes execution to a specific successor instruction. The state of execution is often held in registers or stacks.

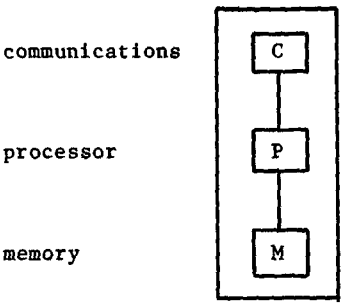

Figure 13. Centralized machine organization.

(2) *Packet communication.* Packet communication machine organization consists of a circular instruction execution pipeline of resources in which processors, communications, and memories are interspersed with "pools of work." This is illustrated by Figure 14. The organization views an executing program as a number of independent information packets, all of which are active, and may split and merge. For a parallel computer, packet communication is a very simple strategy for allocating packets of work to resources. Each packet to be processed is placed with similar packets in one of the pools of work. When a resource becomes idle, it takes a packet from its input pool, processes it, places a modified packet in an output pool, and then returns to the idle state. Parallelism is obtained either by having a number of identical resources between pools, or by replicating the circular pipelines and connecting them by the communications.

(3) *Expression manipulation.* Expression manipulation machine organization consists of identical resources usually organized into a regular structure such as a vector or tree, as shown in Figure 15. Each resource contains a processor, communication, and memory capability. The organization views an executing program as consisting of one large nested program structure, parts of which are active while other parts are temporarily suspended. In an expression manipulation organization the adjacency of items in the program structure is significant, and the memories in this ma-

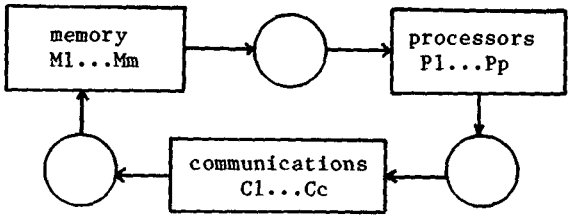

Figure 14. Packet communication machine organization.

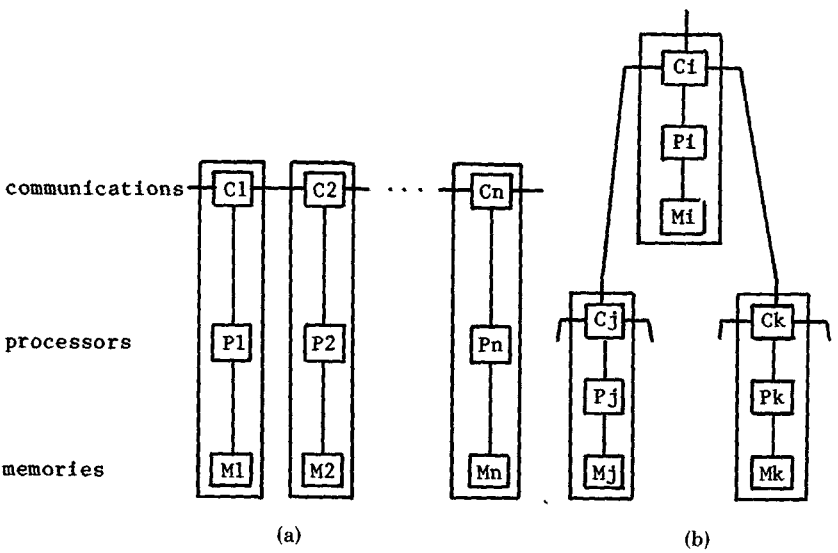

Figure 15. Expression manipulation machine organization: (a) vector; (b) tree

chine structure maintain the adjacency of items in the program structure. Each resource examines its part of the overall program structure looking for work to perform.

Since these machine organizations relate closely to the way programs are represented and executed, the three are often equated and confused with, respectively, control-flow, data-flow, and reduction program organizations. However, as we discuss below, other less obvious pairings of machine and program organizations are possible.

4.2 Control Flow

The most obvious means of supporting control flow is to use a centralized machine organization for sequential forms and either a packet communication or an expression manipulation machine organization for parallel control flow. Sequential control flow supported by a centralized machine organization, where the active instruction is specified by the program counter register, is clearly the basis of all traditional computers. Their familiarity does not warrant further discussion, and we shall concentrate on the support of parallel control flow.

For parallel control flow two basic methods were discussed in Section 1.1 for synchronizing the execution of instructions, namely, FORK-JOIN control operators and the use of control tokens. We start by examining a packet communication machine organization supporting control tokens. In such a machine organization, one of the ways of synchronizing a set of control tokens activating an instruction is to use a *matching* mechanism. This matching mechanism intercepts tokens and groups them into sets with regard to their common consumer instruction. When a set is complete, control is released to activate the instruction, as, for instance, in Figure 1c

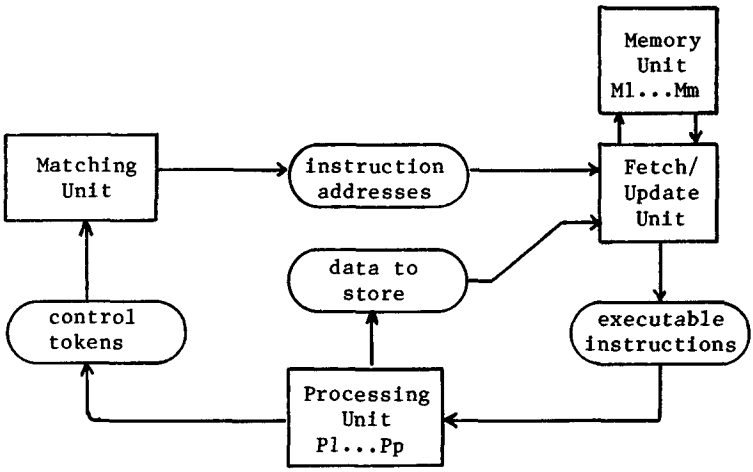

Figure 16. Control-flow packet communications.

with tokens i3/0 and i3/1, which forms a set destined for instruction i3. An example of such a matching scheme is proposed in FARR79 and HOPK79.

Figure 16 illustrates a packet communication machine organization based on token matching. The organization consists of four groups of resources: the matching unit, the fetch/update unit, the memory unit, and processing unit; and four pools of work for instruction addresses, executable instructions, data to store, and control tokens. The task of the matching unit is to group tokens by taking individual tokens from the control tokens pool and storing them in their respective sets in its local memory. When a set of tokens is complete, their common instruction address is placed in the output pool and the set is deleted. The fetch/update unit has two input pools, one containing addresses of instructions to be activated and the other data to be stored. This unit interacts with the memory unit, which stores instructions and data. For each address consumed, the fetch/update unit takes a copy of the corresponding instruction, dereferences its input arguments and replaces them by their corresponding values, and outputs this executable instruction. Last, the processing unit takes executable instructions, processes them, and outputs data to store and control tokens to the respective pools.

For parallel control flow supported by an expression manipulation organization, we consider a control mechanism using FORK–JOIN control operators, as shown in Figure 1b. With this scheme each flow of control is represented by a processor executing instructions sequentially in its local memory. When a processor executes a FORK operator, it activates another processor whose local memory contains the addressed instruction. If this processor is already busy, then the FORK is delayed until the destination processor becomes idle. On completion, the processor issuing the FORK resumes sequential execution. JOIN operators synchronize execution by logically consuming flows of control. The processor executing the JOIN n must be reactivated n times before it resumes sequential execution. The memories of an expression manipulation organization, as shown in Figure 15, maintain the adjacency of instructions in the program structure. Thus a processor sequentially executing instructions may run off the end of its memory. In this case control is passed, in the same way as a FORK operator, to the adjacent processor.

4.3 Data Flow

Since a data-flow computer needs to record the large set of potentially executable instructions, it is difficult to conceive of supporting data flow with a centralized machine organization. We therefore proceed to examine packet communication, the

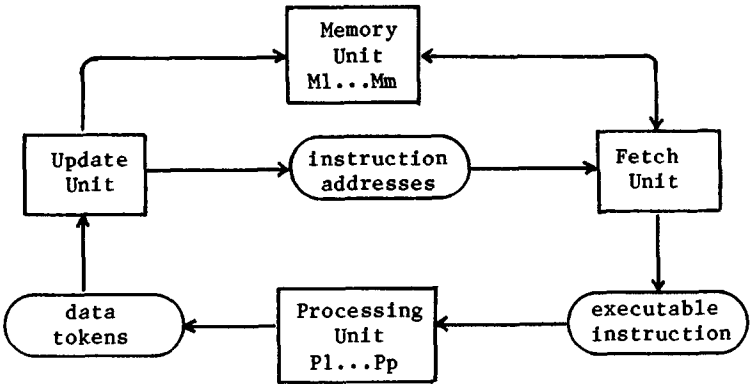

Figure 17. Data-flow packet communication with token storage.

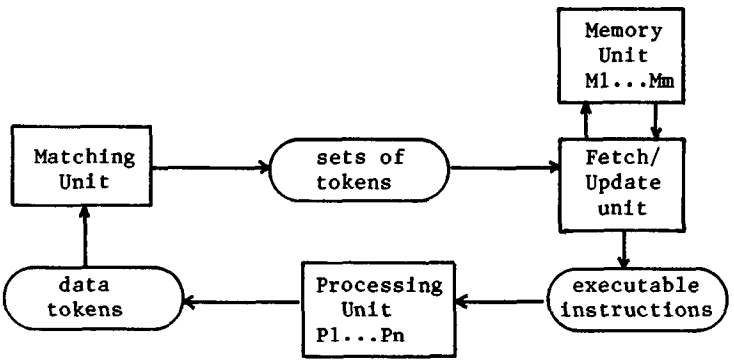

Figure 18. Data-flow packet communication with token matching

most obvious machine organization for supporting data flow.

Instruction execution in data-flow computers is, in general, controlled by either of two synchronization schemes [DENN79b], which we refer to as *token storage* and *token matching*. In the first scheme data tokens are actually stored into an instruction or a copy of the instruction, and an instruction executes when it has received all its inputs. Examples of this scheme include the Massachusetts Institute of Technology [DENN79a] and Texas Instruments [CORN79] data-flow computers. In the second scheme a token-matching mechanism, as described above, is employed. When a set of data tokens is complete, the set is released to activate the consumer instruction—as, for instance, in Figure 2, with i2/1:=4 and i2/2:=2, which form a set of tokens (4,2) for instruction i2. Examples of this scheme include Irvine Data Flow [ARVI80a], the Manchester Data Flow System [WATS79], and the Newcastle Data-Control Flow Computer [HOPK79].

Packet communication organizations based on these two schemes for synchronizing instruction execution are illustrated by Figures 17 and 18. A point of commonality in the two organizations is the processing unit consisting of a number of independent processing elements that asynchronously evaluate the executable instruction packets. Such a packet contains all the information required to process the instruction and distribute the results: the operation code, the input values, and the references for the result tokens.

In Figure 17 the data token packets are in the input pool of the update unit. This unit takes in single data tokens and stores them in the memory unit. Certain of these data tokens may complete the inputs for an instruction, thus enabling it for execution.

For these instructions the update unit places their addresses in its output pool. The fetch unit uses these instruction addresses to retrieve the corresponding instructions and place them in its output pool for execution.

In Figure 18, where synchronization is based on a matching mechanism, data token packets form the input pool of the matching unit. This unit forms them into sets, temporarily storing the set until complete, whereupon the set is released to the fetch/update unit. This unit forms executable instructions by merging the values from a set of tokens with a copy of their consumer instruction.

When a data-flow program organization is supported by an expression manipulation machine organization, each of the identical resources must combine the role of the four units (memory, update, fetch, processing) of the packet communication organization with token storage. When a processing element receives a data token over the communications medium from some other resource, it updates the consumer instruction. The element then inspects the instruction to see if all the inputs are present; if not, it returns to the idle state. If all the inputs are present, the processing element performs the operation and deletes the inputs from its memory. Next it passes the data tokens containing the results to their consumer instructions and returns to the idle state. We place the Utah Data-Driven Machine [DAVI78] in this category.

4.4 Reduction

In reduction computers instruction execution is based on the recognition of reducible expressions and the transformation of these expressions. Execution is by a substitution process, which traverses the program structure and successively replaces reducible expressions by others that have the same meaning, until a constant expression representing the result of the program is reached. There are two basic problems in supporting this reduction on a machine organization: first, managing dynamically the memory of the program structure being transformed and, second, keeping control information about the state of the transformation. Solutions to the memory management problem include (1) representing the program and instructions as strings, for example, "((*) ((+) (b) (1)) ((−) (b) (c)))," which can be expanded and contracted without altering the meaning of the surrounding structure, and (2) representing the program as a graph structure with pointers, and using garbage collection. Solutions to the control problem are (1) to use control stacks, which record, for example, the ancestors of an instruction, that is, those instructions that demanded its execution; and (2) pointer reversal, where the ancestor is defined by a *reversed* pointer stored in the instruction.

Expression manipulation organizations seem most applicable to supporting the reduction form of program organization. However, the computational rules (e.g., innermost and outermost) discussed above provide us with schemes for sequentially executing reduction programs that may be supported by centralized machine organizations. Examples of such centralized organizations includes the GMD reduction machine [KLUG79], which uses seven specialized stacks for manipulating strings, and the Cambridge SKIM machine [CLAR80], which supports graph structures.

Packet communication organizations are also being used to support reduction. An example of such an organization is the Utah Applicative Multiprocessing System [KELL79]. In these organizations, which support demand-driven graph reduction, instruction execution is controlled by two types of token. A consumer instruction dispatches a demand token (containing a return reference) to a producer instruction signaling it to execute and return its results. This producer instruction returns the result in a *result* token, which is basically a data token as in data flow. Two synchronization schemes are required for reduction to be supported by packet communication. The first ensures that only a single demand token for a particular instruction can actually activate the instruction, while the second provides synchronization for result tokens, as was provided for data tokens in data flow.

The final machine organization discussed here is the support of reduction by expression manipulation. Examples of such orga-

Stage	Memories					
	M1	M2	M3	M4	M5	M6
1	(* i1 i2)	-	-	i1:(+b1)	i2:(-bc)	b:(4) c:(2)
2	(* (+	b 1) i2)	-	"	"	" "
3	(* (+	b 1)	(-bc))	"	"	" "
4	(*	(+ 4 1)	(- 4 2))	"	"	" "
5	(* 5 2)	-	-	"	"	" "

Figure 19. Reduction expression manipulation.

nizations are the Newcastle Reduction Machine [TREL80a] and the North Carolina Cellular Tree Machine [MAGO79a]. The example expression manipulation organization we shall examine [WILN80] is one in which the program structure is represented as nested delimited strings and each memory in the machine is connected to its two adjacent memories to form what may be viewed as a large bidirectional shift register. Substitution of an expression into a memory causes the adjacent information to shift apart, which may cause its migration into adjacent memory elements. Figure 19 illustrates this migration of instructions.

To find work each processing element Pi traverses the subexpression in its memory Mi, looking for a reducible expression. Since the "window" of a processing element into the overall expression under evaluation is limited to the contents of its own memory element, it is not possible for two processing elements to attempt simultaneously to reduce the same subexpression—one of the key implementation problems of expression manipulation machines. When a processing element locates a reference to be replaced by its corresponding definition, it sends a request to the communications unit via its communications element Ci. The communications units in such a computer are frequently organized as a tree-structured network on the assumption that the majority of communications will exhibit properties of locality of reference. Concurrency in such reduction computers is related to the number of reducible subexpressions at any instant and also to the number of processing elements to traverse these expressions. Additional concurrency is obtained by increasing the number of Mi-Pi-Ci elements, and also by reducing the size of each memory element, thus increasing the physical distribution of the expressions.

4.5 Implications

From the above discussions of control flow, data flow, and reduction, it is clear that they gravitate toward, respectively, centralized, packet communications and expression manipulation organizations. However, we have also shown, and the fact is being demonstrated by various research groups, that other pairings of program organizations and machine organizations are viable.

Control flow can be efficiently supported by either of the three machine organizations. A centralized organization is most suited to sequential control flow. The advantage of this organization is its simplicity, both for resource allocation and implementation; its disadvantage is the lack of parallelism. A packet communication organization favors a parallel "control token" form of control flow. Although relatively simple, it lacks the concept of an implicit next instruction, thereby incurring additional explicit references in instructions and extra resource allocation. Last, an expression manipulation machine organization is most suited to a parallel FORK-JOIN style of control flow. This organization combines the advantages of the above two by being parallel but also supports the concept of an implicit next instruction. It does, however, incur additional FORK and JOIN style control operators.

For data flow it is difficult to envisage a centralized machine organization because

of the need to record a large number of potentially executable instructions. However, packet communication provides two alternative organizations for the efficient support of data flow. The first packet communication scheme is based on storing data tokens into an instruction and executing the instruction when it is complete. This form of machine organization may be viewed as supporting self-modifying programs and has the advantage of conceptually allowing one data-flow program to generate another for execution. The second packet communication scheme is based on matching data tokens. This form of organization has the advantage of supporting reentrant code, but the disadvantage of being conceptually difficult to generate code for. Data flow may also be supported by expression manipulation, but it is difficult to assess the advantages and disadvantages of this approach.

Finally, we consider machine organization for reduction. Because of the various computational rules for reduction, it can be efficiently supported by any of the three machine organizations. For all these organizations, the two basic problems are, first, managing dynamically the memory and, second, managing the control information. A centralized organization is best suited to a sequential form of reduction. It can implement with reasonable efficiency either string or graph manipulation. A packet communication and expression manipulation organization favor a parallel computational rule.

5. DATA-FLOW COMPUTERS

The number of extremely interesting data-driven and demand-driven computer architectures under investigation has made our task of choosing the set to survey particularly difficult. Since this paper is concerned with identifying related concepts rather than describing implementations, we have chosen to give brief overviews of a number of architecture schemes, described in the open literature, whose concepts seem particularly interesting. Our examination of data-driven computers clearly must start with the Massachusetts Institute of Technology architecture.

5.1 M.I.T. Data-Flow Computer

The contribution of the M.I.T. project to data-flow research has been significant, forming the basis for most other data-flow projects. There are extensive references to this M.I.T. work, which covers data-flow graphs [RODR69, DENN71, DENN72, DENN74a], computer architecture [DENN74b, DENN75b, RUMB77, DENN79a], and the design of high-level programming languages [WENG75, ACKE79a], including the single-assignment language VAL [ACKE79b], based on the abstract-data-type language CLU. (Data-flow languages are in general based on the single-assignment principle [TESL68, CHAM71].) This description of the M.I.T. work concentrates on the computer architecture and is based on a description given in DENN79a.

The program organization used in the M.I.T. computer is clearly data flow; however, only one token may occupy an arc at an instance. This leads to a firing rule which states that an instruction is enabled if a data token is present on each of its input arcs and no token is present on any of its output arcs. Thus the M.I.T. program organization contains control tokens, as well as data tokens, that contribute to the enabling of an instruction but do not contribute any input data. These control tokens act as acknowledge signals when data tokens are removed from output arcs. In the program, organization values from data tokens are stored into locations in an instruction and control tokens signal to a producer instruction when particular locations become unoccupied.

The M.I.T. organization is what we term a packet communication organization with token storage. This organization is shown in Figure 20. It consists of five major units connected by channels through which information packets are sent using an asynchronous transmission protocol. The five units are (1) the Memory Section, consisting of Instruction Cells that hold the instructions and their operands; (2) the Processing Section, consisting of specialist processing elements that perform operations on data values; (3) the Arbitration Network, delivering executable instruction packets from the Memory Section to the Processing Section; (4) the Control Network, deliver-

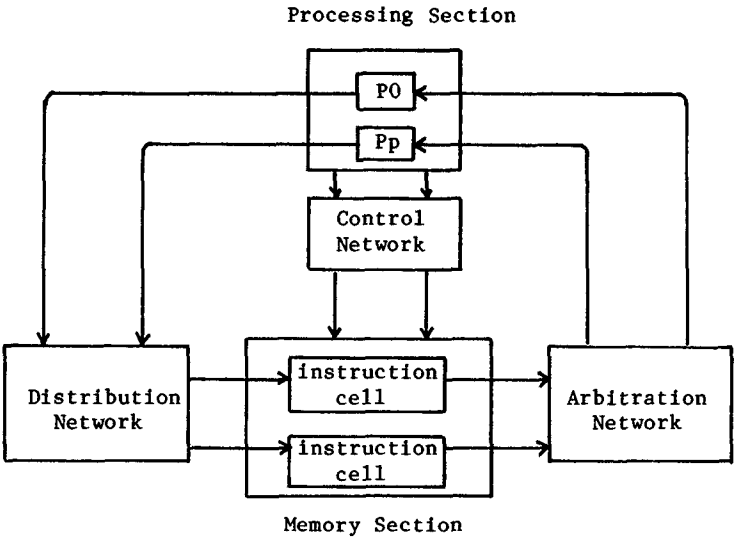

Figure 20. M I.T. data-flow computer.

ing control packets from the Processing Section to the Memory Section; and (5) the Distribution Network, delivering data packets from the Processing Section to the Memory Section.

Instructions held in the Memory Section are enabled for execution by the arrival of their operands in data packets from the Distribution Network and in control packets from the Control Network. Each Instruction Cell in the Memory Section holds one instruction of the data-flow program and is identified by a unique address. When occupied, an Instruction Cell holds an instruction consisting of an operation code and several references (i.e., destination addresses) for results and contains, in addition, three registers, which await the arrival of values for use as operands by the instruction. Once an Instruction Cell has received the necessary operand values and acknowledge signals, the cell becomes enabled.

Enabled instructions together with their operands are sent as operation packets to the Processing Section through the Arbitration Network. This network provides a path from each Instruction Cell to each specialist element in the Processing Unit and sorts the operation packets among its output ports according to the operation codes of the instructions they contain. The results of instruction execution are sent through the Distribution and Control Networks to the Memory Section, where they become operands of other instructions.

Each result packet consists of a result value and a reference derived from the instruction by the processing element. There are two kinds of result packet: (1) control packets containing Boolean values (Boolean data tokens) and acknowledge signals (control tokens), which are sent through the Control Network; and (2) data packets (data tokens) containing integer or complex values, which are sent through the Distribution Network. The two networks deliver result packets to the Instruction Cells specified by their destination field and a cell becomes enabled when all result packets have been received.

The current status of the M.I.T. data-flow project is that hardware for the above computer architecture is under development and a compiler is being written for the VAL programming language. A number of supportive projects on fault tolerance, hardware description languages, etc. are also in progress.

5.2 Texas Instruments Distributed Data Processor

The Distributed Data Processor (DDP) is a system designed by Texas Instruments to investigate the potential of data flow as the basis of a high-performance computer, con-

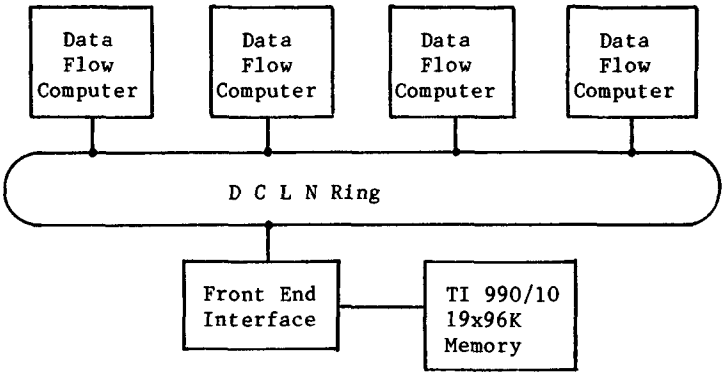

Figure 21. Texas Instruments distributed data processor

structed using only off-the-shelf technology. This project [CORN79, JOHN79] began in mid-1976, and DDP plus its supporting software has been operational since September 1978. A most interesting aspect of the DDP project is that the computer is largely programmed in FORTRAN 66. A cross compiler, based on the Texas Instruments Advanced Scientific Computer's optimizing FORTRAN compiler, translates FORTRAN subprograms separately into directed graph representations and a linkage editor combines them into a single program [JOHN79]. The following description of DDP is largely taken from the paper by Cornish [CORN79].

Conceptually, DDP and the M.I.T. computer discussed above are based on a similar data-flow program organization. An instruction is enabled if a data token is present on each of its input arcs and no token is present on any of its output arcs. Only one token may occupy an arc at an instance. In addition, control tokens are used as acknowledge signals, for instance, to handle FORTRAN language constructs that are resistant to representation by "pure" data-flow code.

A DDP instruction consists of the following fields: (1) an operation code, (2) a so-called predecessor count of the input tokens yet to arrive, (3) a field reserved for a hardware-maintained linked list of instructions ready for execution, (4) an original count of tokens used to restore the predecessor count after the instruction executes, (5) an operand list with space reserved for incoming token operands, and finally (6) a successor list containing the destination instruction addresses for the result tokens. The size of instructions and whether they are of fixed or variable length are unclear from the references.

The DDP machine organization is what we term a packet communication organization with token storage, because operands are stored into unoccupied locations in an instruction. Although this is the same class of machine organization as the M.I.T. computer, the computer architecture of DDP is significantly different. A block diagram of the DDP system is shown in Figure 21. It consists of five independent computing elements: four identical data-flow computers that cooperate in the execution of a computation and a Texas Instruments 990/10 minicomputer, acting as a front-end processor for input/output, providing operating system support, and handling the collection of performance data. (A data-flow program to be executed is statistically partitioned and allocated among the four data-flow computers.) These five computing elements in the DDP are connected together by a variable-length, word-wide, circular shift register known formally as a DCLN ring. This shift register is daisy chained through each element and may therefore carry up to five variable-length packets in parallel.

Each data-flow computer consists of four principle units. These units are (1) the Arithmetic Unit, which processes executable instructions and outputs tokens; (2) the Program Memory, built out of standard random-access-memory (RAM) chips and

holding the data-flow instructions; (3) the Update Controller, which updates instructions with tokens; and (4) the Pending Instruction Queue, which holds executable instructions that have been enabled. Executable instructions are removed from this queue by the Arithmetic Unit and processed. When an instruction completes execution, a series of token packets are released to the Update Controller. Using the address in a packet the Update Controller stores the token operand in the instruction and decrements by one its predecessor count. If this count is zero, the instruction is ready to execute; a copy is placed on the Pending Instruction Queue and the stored version of the instruction is reinitialized. It is unclear whether the Pending Instruction Queue may contain more than one executable instruction. However, when the capacity of the queue is exceeded, the enabled instructions are linked, in memory, to the Pending Instruction Queue via their link field already reserved for this purpose. This method has the advantage that no amount of program parallelism overflows the capacity of the hardware resource.

DDP is implemented in transistor-transistor logic (TTL) on wire-wrap boards and chassis from Texas Instruments 990 minicomputer components. Each data-flow computer contains 32K words of metal-oxide-semiconductor (MOS) memory with each word divided as a 32-bit data field and a 4-bit tag field holding the predecessor count. Each of these computers contains about the same number of components as a minicomputer and provides approximately the same raw processing power. Thus, as remarked by Cornish [CORN79, pp. 19–25], "data flow designs place no particular burden on the implementation other than using more memory for program storage."

5.3 Utah Data-Driven Machine

Data-Driven Machine #1 (DDM1) is a computing element of a recursively structured data-flow architecture designed by Al Davis and his colleagues while working at Burroughs Interactive Research Center in La Jolla, California. DDM1 [DAVI78, DAVI79a, DAVI79b] was completed in July 1976 and now resides at the University of Utah, where the project is continuing under support from Burroughs Corporation. Here we examine the structure of this recursive architecture and the operation of DDM1, descriptions primarily taken from Davis [DAVI78].

The program and machine organization, both based on the concept of recursion, contrasts markedly with the previous data-flow systems we have examined. The computer is composed of a hierarchy of computing elements (processor–memory pairs), where each element is logically recursive and consists of further inferior elements. Physically the computer architecture is tree structured, with each computing element being connected to a superior element (above) and up to eight inferior elements (below), which it supervises. Only recently have other groups come to recognize the fundamental importance of hierarchy for decentralized systems, particularly those exploiting VLSI [SEIT79, MEAD80, TREL80b], since it is able to utilize locality of reference to reduce the critical problems of system-wide communication and control.

In the Utah data-flow program organization, referred to as Data-Driven Nets [DAVI79a], data tokens provide all communication between instructions—there are no control tokens. In addition, the arcs of the directed graph are viewed as first-in/first-out (FIFO) queues, a model that is supported by the architecture. The actual program representation corresponding to these Data-Driven Nets consists of hierarchically nested structure of variable-length character strings. A data-flow program, its subprograms, and their individual instructions are each viewed as a parenthesized string, for example,

"(( ) (( ) . . .) . . .)."

The notion of an arc being a FIFO queue is supported by storing the data tokens that have arrived but have not been consumed with the instruction in the program structure. Each instruction therefore consists of an operation code and a list of destination addresses for the results, together with a variable number of sets of data tokens waiting either for a set to be complete or for consumption by the instruction. An advan-

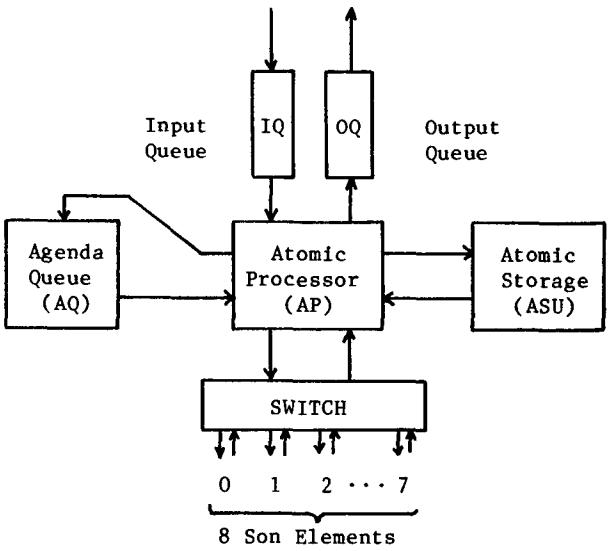

Figure 22. Utah data-driven machine (DDM1).

tage of the parenthesized string form of representation is that it supports dynamic, and localized, variation of the program structure. Because of the nature of this program representation and the method of allocating work to resources, discussed below, we classify the Utah architecture as an expression manipulation machine organization.

A block diagram of the computing element DDM1 is shown in Figure 22. DDM1 consists of six major units: (1) the Atomic Storage Unit (ASU) provides the program memory; (2) the Atomic Processor (AP) executes the instructions; (3) the Agenda Queue (AQ) stores messages for the local Atomic Storage Unit; (4) the Input Queue (IQ) buffers messages from the superior computing element; (5) the Output Queue (OQ) buffers messages to the superior element; and finally (6) the SWITCH connects the computing element with up to eight inferior elements. All paths between these units, except for that between the Atomic Storage Unit and Atomic Processor are six wire paths (a two-wire request–acknowledge control link and the four-wire, character-width data bus). The units communicate asynchronously using a four-phase request–acknowledge protocol.

Work in the form of a program fragment is allocated to a computing element by its superior, being placed as a message in the Input Queue. The action taken by the computing element depends on the structure of the fragment and whether there are further inferior elements. If there exists some set of concurrent subprograms and the computing element has substructure, then it will decompose and allocate the subprograms to its inferior elements. Otherwise, the program fragment is placed in the element's own Atomic Storage Unit. The Atomic Storage Unit of DDM1 contains a 4K × 4-bit character store, using RAM devices, and also performs storage management functions on the variable-length parenthesized strings, such as initialize, read, write, insert, and delete. All target locations in the store are found by an access vector into the tree-organized storage structure. Free space is managed automatically.

When a data token arrives as a message, for example, in the Input Queue, it is either passed on via the appropriate queue to a computing element at some other level, or if the program fragment is in the local Atomic Storage Unit, it is inserted into the instruction. When such an instruction becomes enabled, it is executed immediately by the Atomic Processor and the result tokens distributed. These are placed in the Output Queue or SWITCH, or if the receiving instruction is in the local Atomic Stor-

age Unit, they are placed in the Agenda Queue. After the processor has generated all the result tokens, it will service messages from the SWITCH, the Agenda Queue, and the Input Queue, in descending order of priority.

Current status of the project is that DDM1 is operational and communicates with a DEC-20/40, which is used for software support of compilers, simulators, and performance measurement programs. The current programming language is a statement description of the directed graph; however, an interactive graphical programming language is also under development.

5.4 Irvine Data Flow Machine

The Irvine data-flow (Id) machine is motivated by the desire to exploit the potential of VLSI and to provide a high-level, highly concurrent program organization. This project originated at the University of California at Irvine [ARVI75, ARVI77a, ARVI77b, GOST79a, GOST79b] and now continues at the Massachusetts Institute of Technology [ARVI80a, ARVI80b]. It has made significant contributions to data-flow research, in particular the Id language [ARVI78]. The description given here is principally based on ARVI80a.

The program organization used in the Id machine is pure data flow, with an instruction being enabled when all its input tokens are available. Each instruction may have one or two inputs and any number of outputs. There are a number of interesting features in the Id program organization. The first feature is the sophisticated token identification scheme, similar to the P/N/A/I format discussed in Section 3.3. A token identifier consists of (1) a code block name identifying a particular procedure or loop; (2) a statement number within the code block; (3) an initiation number for the loop; and (4) a context name identifying the activity invoking this procedure or loop. The second interesting feature is its support for data structures, such as arrays, by the inclusion of *I structures* [ARVI80b]. An I structure is a set of components, with each component having a unique selector (for an array the selectors are the indexing integers) and being either a value or an unknown if the value is not yet available. This feature uses the by-reference mechanism of control flow. Two further features are that Id supports the nondeterminism required for implementing resource managers and, by treating procedure definitions as manipulable values, supports higher order functions, abstract data types, and operator extensibility. These features are discussed in detail in ARVI78.

The Id machine has a packet communication organization with token matching. It consists of N processing elements and an $N \times N$ communications network for routing a token from the processing element generating it to the one consuming the token. This machine organization attempts to minimize communications overhead in two ways. First, the matching unit for tokens destined for a particular instruction is in the same processing element as is the storage holding that instruction. Second, there is a short-circuit path from a processing element to itself so that there is no need to use the full $N \times N$ network if a token is destined for the same processing element as generated it. The mapping algorithm, determining in which processing element an instruction is stored, is intended to obtain maximum usage of this short circuit, while still giving good processor utilization.

Figure 23 illustrates a processing element of the proposed Irvine data-flow machine. Each processing element is essentially a complete computer with an instruction set, up to 16K words each of program storage and data structure storage, and certain special elements. These specialist elements include: (1) the input section, which accepts inputs from other processing elements; (2) the waiting-matching section, which forms data tokens into sets for a consumer instruction; (3) the instruction fetch section, which fetches executable instructions from the local program memory; (4) the service section, that is, a floating-point arithmetic-logic unit (ALU) (e.g., Intel 8087); and (5) the output section, which routes data tokens containing results to the destination-processing element.

The current status of the project is that a computer with 64 processing elements is currently being designed at the Massachusetts Institute of Technology and is ex-

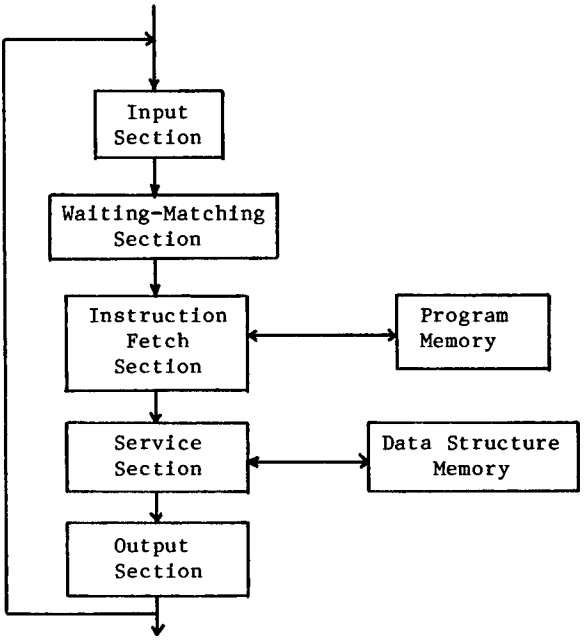

Figure 23. Irvine data-flow processing element

pected to be ready for MOS fabrication by the end of 1982.

5.5 Manchester Data-Flow Computer

The data-flow project at Manchester University, like a number of other projects, is investigating the use of data flow as the basis for a high-performance computer. This project, starting in 1975, has included the design of a high-level, single-assignment programming language LAPSE, the implementation of translators for LAPSE and a subset of PASCAL, and the production of a detailed stimulator for the Manchester computer architecture. Currently the group is implementing a 20-processor data-flow computer prototype. Early ideas on this design are given in TREL78; this description of the computer is based on WATS79.

The program organization used by the Manchester computer is pure data flow, with an instruction being enabled when all its input arcs contain tokens (its output arcs may also contain unconsumed data tokens), and an arc is viewed as a FIFO queue providing storage for tokens. The program representation is based on a two-address format, with an instruction consisting of an operation code, a destination instruction address for a data token, and either a second destination address or an embedded literal. Each instruction consumes either one or two data tokens, and emits either one or two tokens. A token consists of three fields: the value field holding the operand, an instruction address field defining the destination instruction, and last a label field. This label is used for matching tokens into sets and provides three types of information, identifying the process to which the token belongs, the arc on which it is traveling, and also an iteration number specifying which particular token on an arc this is. Thus tokens have a four-field name, as discussed in Section 3.3, serving a number of roles in the architecture, including supporting the notion of arcs being FIFO queues, allowing tokens to be matched into sets, and allowing a program's instructions to be used reentrantly.

The machine organization of the computer is a packet communication organization with token matching. Figure 24 shows a block diagram of the Manchester data-flow computer. It consists of five principal units: (1) the Switch provides input-output for the system; (2) the Token Queue is a FIFO buffer providing temporary storage

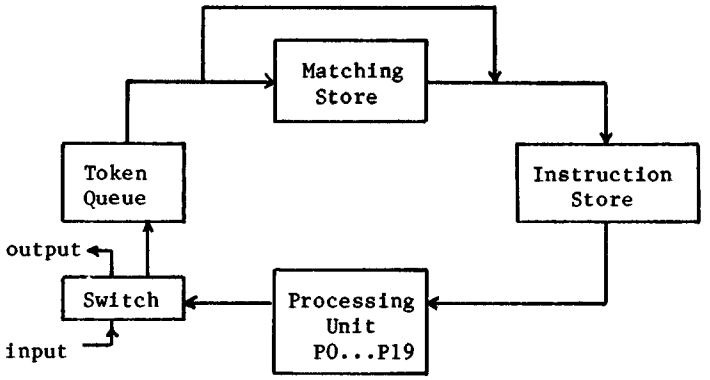

Figure 24. Manchester data-flow computer.

for tokens; (3) the Matching Store matches pairs of tokens; (4) the Instruction Store is the memory holding the data-flow programs; and (5) the Processing Unit, consisting of a number of identical processing elements, executes the instructions. The Switch is used for passing data tokens into or out of the computer, either communicating with peripherals or other, possibly data-flow, computers. To start execution of a program fragment, initialization tokens are inserted at the Switch and directed by their labels to the starting instructions of the computation. A special destination address in the final instructions of the program fragment allows tokens to be output.

A token on reaching the front of the Token Queue can access (if one of a pair) or bypass (if a single input) the Matching Store, depending on information in the token label. An access to the Matching Store will cause a search of the store. The Matching Store is associative in nature, although it is implemented using RAM with hardware hashing techniques, and is based on the work of Goto and Ida [GOTO77]. If a token is found with the same label and instruction address, it is removed to form a token pair. If no match is found, the incoming token is written to the store. Token pairs from the Matching Store, or single tokens that have bypassed it, are routed to the Instruction Store. At this store, which is a RAM addressed by the contents of the instruction address field, the tokens are combined with a copy of the destination instruction to form an executable instruction that is released to the Processing Unit. This unit consists of a distribution and arbitration system, and a group of microprogrammed microprocessors. The distribution system, on receipt of an executable instruction, will select any processor that is free and allocate the instruction. After execution, the arbitration system controls the output of result tokens from the processing elements.

The current status of the project is that a 20-processing-element computer is under construction. Each processing element is built from Schottky bit-slice microprocessors and is estimated to give an average instruction execution time of 3 microseconds for the data-flow arithmetic operations. If all 20 processing elements can be utilized fully, this will give an approximately 6-million-instruction-per-second rate for the computer as a whole. To support this rate, the following operation times [WATS79] are required: (1) Token Queue read 202 nanoseconds; (2) Matching Store access 303 nanoseconds; (3) Instruction Store read 303 nanoseconds; (4) SWITCH operation 202 nanoseconds; and (5) Token Queue write 202 nanoseconds. These speeds require a storage access time of the order of 200 nanoseconds, which is achievable with low-cost MOS storage devices.

5.6 Toulouse LAU System

"*Language à assignation unique*" is the French translation for the phrase "single-assignment language." The LAU system [COMT76, GELL76, PLAS76, SYRE77,

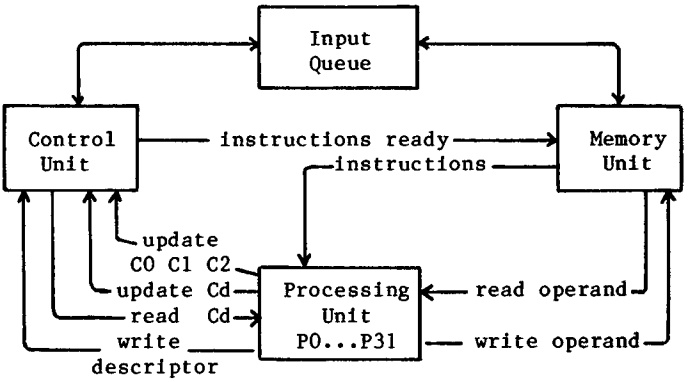

Figure 25. LAU system.

COMT79b] is a data-driven computer designed to execute such languages. The LAU project is based at the CERT Laboratory in Toulouse. Notably this extensive project, starting in 1976, initially designed the LAU high-level language, which was used to program a large number of problems. Subsequently, the group implemented a compiler for the language and a detailed simulator, which yielded a large number of simulation data [PLAS76]. This led to the design and current construction of a powerful 32-processor data-driven computer. The description of the LAU computer given here is based on the paper by Comte and Hifdi [COMT79b].

The LAU programming language has a data-flow model, but the computer's program organization is in fact based on control-flow concepts. In the computer data are passed via sharable memory cells that are accessed through addresses embedded in instructions, and separate control signals are used to enable instructions. However, it should be stressed that, as in data flow, the flow of control is tied to the flow of data (i.e., the control graph and the data graph are identical).

Program representation is based on three logical types of memory, for instructions, for data, and for control information. An instruction (66 bits in length) has a three-address format and consists of an operation code, two data memory addresses for input operands, and a data memory address for the result operand. Following conventional practice, if an input operand is a literal, it replaces the address in the instruction. Each cell in the data memory consists of a value field providing storage for the operand and of two link fields that contain instruction memory addresses of instructions using the operand as an input.

Corresponding to each instruction and data operand are sets of control bits which synchronize execution. Three control bits referred to as C0, C1, and C2 denote the state of an instruction. C1 and C2 define whether the two corresponding input operands are available, while C0 provides environment control, as, for instance, for instructions within loops. An instruction is enabled when C0 C1 C2 match the value 111. A final control bit, referred to as Cd, is associated with each data operand and specifies if the operand is available. Execution of an enabled instruction consists of fetching the two input operands from the data memory using the embedded operand addresses, and performing the specified operation. Next, the result operand is written to the data memory using the result address, which causes the corresponding link addresses to be returned to the processor, and is used to update the corresponding C1 and C2 of instructions using the result as inputs.

The LAU machine organization is a packet communication organization with token storage, owing to the form of program organization, notably the association of C0 C1 C2 control bits with each instruction. Figure 25 illustrates the system organization of the LAU computer. It comprises the

memory unit providing storage for instructions and data, the control unit maintaining the control memory, and the processing unit consisting of 32 identical processing elements. Each element is a 16-bit microprogrammed processor built around the AMD 2900 bit-slice microprocessor. Perhaps the most interesting part is the control unit, where the von Neumann program counter is replaced by two memories: the Instruction Control Memory (ICM) and the Data Control Memory (DCM). ICM handles the three control bits C0 C1 C2 associated with each instruction and DCM manages the Cd bit associated with each data operand.

As an illustration of the operation of the LAU computer let us consider the processing of an enabled instruction. Processing starts in the control unit at the Instruction Control Memory. ICM is composed of 32K three-bit-wide words, with the control bits in word i corresponding to the instruction in word i in the memory unit. Two processors scan this memory: the Update Processor sets particular bits of C0 C1 C2, and the Instruction Fetch Processor associatively accesses the memory for 111 patterns. When an enabled instruction is found, its address is sent to the memory unit and the control bits are reset to 011.

The address of the enabled instruction is queued, if necessary, in a 16-bit × 64-word FIFO queue, which is a pool of work for the memory unit. This unit consumes the address and places the corresponding instruction on the instruction bus, which is also a 64-bit × 128-word FIFO queue, where it is eventually accessed by an idle processing element. Once in a processing element, the instruction is decoded and the input addresses are dispatched to the memory unit to access the data operands. When the inputs return, the operation is performed and the result generated. Next the processing element issues a write–read request to the memory unit giving the result and its address. The result will be stored in the value field and the contents of the two link fields will be returned to the element. Once the link fields have been returned, the processing element sends the links to the Update Processor, which uses them to set the corresponding C1 or C2 bits in the instruction control memory. In parallel to the storing of the result, the processing element sends the result address to the data control memory where the Cd bit is set. This memory is n 1-bit words. Like the ICM, the DCM is served by two processors, one that updates the Cd bits and the other that checks that accesses to operands in the memory unit are in fact available (i.e., the Cd bit is set).

Regarding the status of the LAU project, the first of the 32 processors became operational in September 1979, and the remainder have been constructed since then. Predicted performance figures for this hardware are given in COMT79b.

5.7 Newcastle Data-Control Flow Computer

Most of the data-driven projects discussed above are based on a single program organization and are concerned, specifically, with studying its embodiment in a suitable machine organization. In contrast, the group at the University of Newcastle upon Tyne are interested in the actual program organizations, their suitability for a general-purpose decentralized computer, and the possibilities for combining them. In this respect the group has investigated, using software and hardware simulators, data flow [TREL78], "multithread" control flow [FARR79], and reduction [TREL80a] organizations, and also combinations of more than one organization in a single computer. Here we describe the JUMBO computer architecture [HOPK79, TREL82] built to study the integration of data-flow and control-flow computation.

The program organization has both data tokens and control tokens, and some specific combination of tokens causes the enabling of a particular instruction. In the organization there are two ways in which an instruction may obtain its input operands, namely, (1) by receiving data tokens, which may carry a value or an address of the stored value; or (2) by means of embedded inputs stored in the instruction, which, like the contents of data tokens, may be literal values or addresses. When an instruction is enabled, the token inputs and embedded inputs are merged to produce a set of values and addresses. The addresses of inputs are then dereferenced and re-

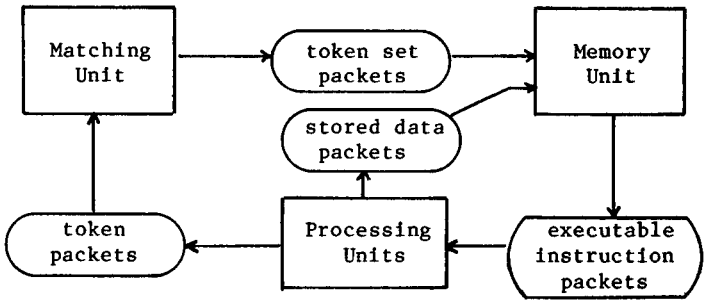

Figure 26. Newcastle data-control flow computer.

placed by their corresponding values from memory. The resulting executable instruction then has a complete set of value arguments on which to compute.

An instruction consists of an operation code and up to eight arguments, certain arguments being embedded in the stored instruction and others being supplied by data tokens at run time. Each operation code uses arguments in specific positions for inputs and places its results in other positions. A stored instruction therefore consists of (1) an operation code, (2) up to eight embedded arguments, (3) a position field defining those arguments that are present, (4) an input mode field defining which of the merged token and embedded arguments are to be dereferenced, and (5) an output mode field specifying which arguments and results are to be combined to produce the outputs of the instruction after execution. Three types of output may be produced by an instruction, namely, data to store in memory, data tokens, and control tokens. Each consists of a reference and a value. For data to store, the name gives the address of the memory cell; for tokens it gives the address of the destination instruction and information to control the token's matching with other tokens in the set, such as the count of tokens. In the computer up to four tokens may be grouped together in a set.

The machine organization of the JUMBO computer is a packet communciation organization with token matching. A block diagram of the computer, as shown in Figure 26, consists of three principal units interconnected by FIFO buffers. The Matching Unit controls the enabling of instructions by matching sets of tokens, which are released to the Memory Unit when complete. The Memory Unit provides storage for data and instructions. It places the contents of stored data packets in the appropriate memory cell, and for token set packets it constructs executable instructions, which are released to the Processing Unit. Finally, the Processing Unit supports instruction execution and the distribution of results.

When a token set packet is released by the Matching Unit, it contains between zero and four input arguments supplied by data tokens. Using the destination instruction address in the packet, the Memory Unit takes a copy of the target instruction and merges the token arguments with those already embedded in the instruction. The copy of the instruction now has a complete set of arguments. Next, the input mode field, which is an 8×1-bit vector, is extracted, and for each bit set the corresponding argument is assumed to be a memory address and is dereferenced and replaced by its corresponding value to give an executable instruction.

Each of the three units of the JUMBO computer is built from a Motorola M6800 microcomputer system. Storage in the JUMBO computer is divided into 1-kbyte pages. Each process executing in the computer has three pages, one for its tokens in the Matching Unit, and one each for its code and data in the Memory Unit. Processes can be dynamically created and killed, and the token page can be reallocated, implicitly deleting residual tokens so that graphs do not have to be self-cleaning as on other data-driven computers.

5.8 Other Projects

Research into data flow is a rapidly expanding area in the United States, Japan, and Europe. Besides the projects briefly described above, there are a number of other interesting data-flow projects worthy of description in this survey. These include: the MAUD single-assignment system at the University of Lille, France [LECO79]; work at the Mathematical Center, Amsterdam on compiling conventional languages for data-flow machines [VEEN80]; the PLEXUS project at the University of Tampere, Finland [ERKI80]; the FLO project at the University of Manchester, England [EGAN79]; work on a hierarchical data-flow system at the Clarkson College of Technology, New York [SHRO77]; and a number of machines that have been built or are under development in Japan [JIPD81b] such as a high-speed, data-flow machine being developed at Nippon Telegraph and Telephone [AMAM80, JIPD81b].

6. REDUCTION COMPUTERS

Apart from the pioneering work of Klaus Berkling, the stage of development of reduction computers somewhat lags behind that of data-flow computers. This is probably due to reduction semantics being an unfamiliar form of program execution for most computer architects.

6.1 GMD Reduction Machine

The reduction machine project based on the GMD (Gesellschaft für Mathematik und Datenverarbeitung) Laboratory in Bonn, West Germany, aimed to demonstrate that reduction machines are a practical alternative to conventional architectures. In particular, the aim was to build a computer easy to program directly in a high-level, functional language based on the lambda calculus. Early ideas on this theme are given in BERK71 and consolidated in BERK75. This description of the GMD reduction machine is based on the account in KLUG79, supplemented by information from HOMM79 and KLUG80.

The GMD machine's program organization is string reduction. A design objective was the elimination of addresses entirely, and this is achieved by always using substitution copies of code and data instead of sharing by using addresses. In the machine a program is represented as a prefix expression, the binary tree structuring being uniquely exhibited by a string of symbols. These expressions may be atoms—single symbols or values—or may themselves be strings. Each subtree consists of three parts, namely, a *constructor*, a *function*, and its *argument*.

One task of the constructor is to indicate which of its offspring in the tree is the function and which the argument. Since the reduction machine is designed to traverse expression trees in preorder (i.e., left subtree before the right), it is necessary to know whether the function or the argument should be reduced first, and the order in which they occur in the expression. This is provided by two types of constructor represented by the symbols ":" and "←." The constructor ":", used in the format ": argument function", evaluates the argument expression, by reduction to a constant expression, before the function is applied to it. The constructor "←", used in the form "← function argument", applies (reduces) the function expression before the argument is evaluated.

Expressions may in general be built from either constructor and identical constant expressions obtained. For instance, the arithmetic expression 4 + 2 can be represented either as :2:4+ or as ←←+42. Differences arise when constructors are applied to function bodies as they give rise to by-value and by-name parameter substitution. Special symbols for function-argument binding are also provided in the form of a pair of constructors lambda and alpha. The former implements standard lambda substitution, while the latter is used to implement recursion. Lambda simply causes the actual parameter to be substituted for a formal parameter in an expression (the operation being known in lambda calculus as a beta reduction). Alpha is used to bind function bodies to occurrences of the function name in recursive expressions, with occurrences of the name being replaced by a new application of alpha, for example,

ALPHA.f(. . . f . . .)
    reduces to f(. . . ALPHA.f . . .)

Obviously the bracketed body of f must

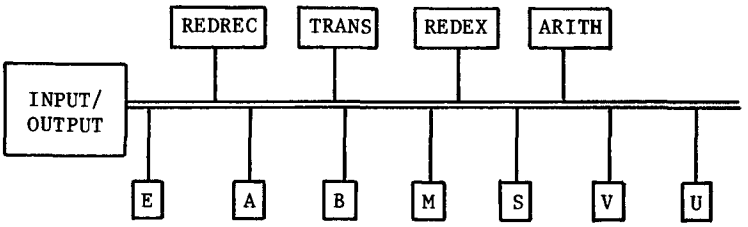

Figure 27. GMD reduction machine.

contain a terminating condition to prevent the recursion's being infinite.

The GMD machine organization is classified as a centralized organization, particularly by the way it represents and executes programs. A block diagram of the machine architecture is shown in Figure 27. It consists of the reduction unit (comprising four subunits named TRANS, REDREC, REDEX, and ARITH), a set of seven 4-kbyte push-down stacks (of which E, A, B, U, V, and M are used to process expressions, and S serves as the system control stack), and a 1-byte-wide bus system for communication between the various units. In the reduction unit the four subunits perform the following tasks. TRANSport performs all traversal algorithms; REDuction-RECognition looks for an instance of a reducible expression during traversal and, upon finding one, halts the TRANS unit and passes control to the REDEX unit. REDuction-EXecution essentially provides a fast control memory containing all the control programs to perform the reductions. In this task it is assisted by the ARITHmetic unit, which performs all the arithmetic and logical operations.

In the traversal of an expression by the machine, three principal stacks are used. These are E, M, and A, referred to as *source*, *intermediate*, and *sink*. The source stack holds the tree expression to be reduced with the root constructor on top of the stack. As the expression is traversed, a succession of pop operations moves the symbols off the source stack onto the sink stack. For reasons of consistency, the expression ending up on the sink stack must appear with the constructors on top of their respective subtrees. To accomplish this, the third intermediate stack is used as temporary storage for constructors that emerge from the source stack ahead of their subexpressions, but must enter the sink stack after them.

The GMD reduction machine has been built and is connected to a microcomputer system supporting a library and programming tools. The whole system has been operational since 1978. An attempt has also been made to implement Backus' FP language [BACK78], but in general this is less successful than the original lambda calculus language for which the machine was designed. The main contribution of the GMD project is to demonstrate that there is sufficient understanding of reduction to implement a workable machine. The project has also shown that string manipulation is a useful technique but may be inefficient when adhered to rigorously.

6.2 Newcastle Reduction Machine

The Newcastle reduction machine project aimed to investigate the use of parallelism in such machines and also explore the feasibility of basing these designs on a few replicated large-scale integrated (LSI) parts. This project resulted in the design and simulation of a parallel string reduction machine, the major feature of the design being the use of state-table-driven processors that allowed the computer to be used as a vehicle for testing different reduction (language) schemes. The presentation given here is based on TREL80a and uses an example reduction language described there.

The program organization uses string manipulation; references may occur in a string, and these are substituted by the corresponding definition at run time. A parallel innermost computation rule is used. An expression in the program representa-

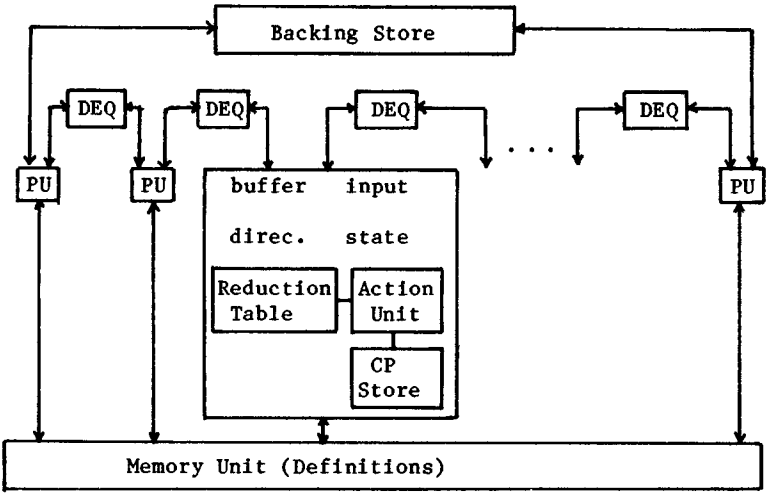

Figure 28. Newcastle reduction machine.

tion is delimited by left bracket "(" and right bracket ")" symbols, and consists of a function followed by a list of arguments "(function arg 1 arg 2 ...)." Here function is a simple operator, but an argument may be a literal value, a reference to a definition, or a bracketed expression to be reduced.

Besides the normal arithmetic, logical, and conditional operators, there are LOAD, STORE, and APPLY operators used to access definitions explicitly. LOAD is used for dereferencing and replaces the reducible expression (LOAD ref) by the definition corresponding to ref. STORE is used (STORE ref def) to create or update stored definitions and can, if not used carefully, violate the referential transparency property of reduction machines. APPLY is used to bind arguments to a parameterized function.

The machine organization, an expression manipulation type, of the Newcastle reduction machine is shown in Figure 28. It consists of three major parts: (1) a common memory unit containing the definitions; (2) a set of identical, asynchronous processing units (PU); and (3) a segmented shift register containing the expression being evaluated. This shift register comprises a number of double-ended queues (DEQ) containing the parts of the expression being traversed, and a backing store to hold the inactive parts of the expression. Each processing unit has direct access to the whole memory unit and two double-ended queues. Figure 28 also shows the architecture of an individual processing unit. It consists of four registers containing information on the subexpression being traversed, the reduction table that contains the user-defined state transition table controlling the evaluation, an action unit performing the actions specified by the reduction table, and the operation store holding user-defined code for the action unit.

The basic aim of each processing unit is to build up a reducible expression "(operator constant...)" in its buffer register and then rewrite it. Each processing unit can read or write to either of its double-ended queues, the current direction being maintained by the direction register. When an item is read and removed from a DEQ, it is transferred into the input register. Associated with each item is a type field (e.g., operator, operand, left bracket, right bracket, empty), which is used in conjunction with the current state, held in the state register, to index into the reduction table. The selected reduction table entry defines an action to be performed, such as move item to buffer register and new values for the state and direction registers. For instance, the registers of a processing unit might contain the following—buffer: "(+ 4 2"; input ")"; direction: "right"; state: "3"—when reading from the right and a right bracket is encountered. For the example

reduction language the action selected in the state transition table would reduce the expression. Had a left bracket been input instead, the selected action would have emptied the contents of the buffer register into the left-hand DEQ, and attempted to find a new innermost reducible expression.

The asynchronous operation of the processing units and their parallel traversal of the expression clearly provide scope for deadlock and starvation. For example, two adjacent units might be attempting to reduce simultaneously the same innermost, reducible expression. To avoid problems such as these, the state transition table obeys certain protocols; in this instance the processing unit on the right reading an empty DEQ would output the contents of its buffer register and reverse direction. To enforce the use of these protocols, a software package called the reduction table generator is used to automatically generate a consistent reduction table for a user's language, input as a Backus–Naur Form (BNF) syntax. This package employs ideas similar to compiler-compilers that are used to generate table-driven LR parsers.

For this proposed reduction machine design, the novel features stated are the use made of state tables to support a class of user-defined reduction schemes and the use made of parser generator concepts for generating these tables. The main disadvantages of the proposal seem to be the normal ones of innermost reduction, such as correctly handling conditionals, and the global memory unit, which is a bottleneck.

6.3 North Carolina Cellular Tree Machine

The cellular computer architecture project [MAGO79a, MAGO79b, MAGO80] at the University of North Carolina, Chapel Hill, is strongly influenced both by VLSI and functional programming. Specifically, the computer has the following four properties: (1) it has a cellular construction, that is, the machine is obtained by interconnecting large numbers of a few kinds of chip in a regular pattern; (2) it executes Backus' FP class of languages [BACK78]; (3) it automatically exploits the parallelism present in FP programs; and (4) its machine language is, in fact, the FP language. Extensive simulation studies of the computer architecture have been carried out and are referenced in Mago's papers. This brief description of the architecture is based on MAGO80.

Since the cellular computer is based on FP, its program organization is string reduction with a parallel innermost computation rule. The program representation in the computer is the symbols of the FP language. In this language, a program is an expression consisting of nested applications and sequences. Each application is composed of an operator and an operand. For example, the expression $\langle 7, (+ : \langle 2,5 \rangle) \rangle$ is a sequence of two elements, the first being the number 7 and the second being an application. In the application the operator is the + and the operand is the sequence of two numbers $\langle 2,5 \rangle$.

An FP machine program is a linear string of symbols that are mapped into a vector of memory cells in the computer one symbol per cell, possibly with empty cells interspersed. This is illustrated by Figure 29. Some of the symbols used to separate expressions in the written form of FP programs are omitted in the machine representation, since their function is served by cell boundaries. In addition, to simplify the operation of the computer, closing application and sequencing brackets are omitted and instead an integer is stored with every remaining FP symbol, indicating the nesting level of that symbol. This is also shown in Figure 29.

The cellular computer's machine organization—an expression manipulation type—is a binary tree structure with two different kinds of cell. Leaf cells (called L cells) serve as memory units, and nonleaf ones (called T cells) provide a dual processing/communication capability. An FP expression is mapped onto this tree structure, each FP symbol being stored in an L cell and a subtree of symbols (i.e., a subexpression) being linked by some dedicated T cells, as shown in Figure 29. A particular set of L and T cells will be dedicated to a subtree for at least the duration of one machine cycle.

Having partitioned the expression to be executed into a collection of cells, itself a cellular computer, the interaction of these cells in the reduction of an innermost ap-

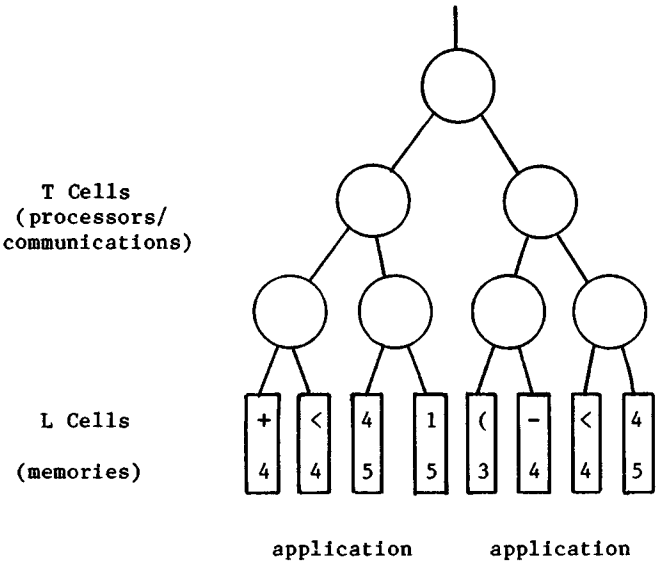

Figure 29. Cellular tree machine.

plication is handled by microprograms. Microprograms normally reside outside the network of cells and are brought in on demand. Once a microprogram is demanded, it is placed in registers in the L cells, each cell receiving a fraction of the microprogram, that part necessary to make its contribution to the total reduction. For example, if one of the L cells wants to broadcast some information to all other L cells involved in reducing a subexpression, it executes a SEND microinstruction [MAGO79a], explicitly identifying the information item to be broadcast. As a result, this information is passed to the root of the subexpression and broadcast to all appropriate L cells.

It often happens that the result expression is too large to be accommodated in the L cells that held the initial expression. In such a case, if the required number of L cells are available elsewhere, then cell contents are repositioned. This storage management is the only kind of resource management needed in the processor because whenever an expression has all the L cells needed, it is guaranteed to have the necessary T cells.

The operation of the cells in the network is coordinated, not by a central clock, but by endowing each cell with a finite-state control, and letting the state changes sweep up and down the tree. This allows global synchronization, even though the individual cells work asynchronously and only communicate with their immediate neighbors.

For a detailed description of the cellular computer's structure and operation the reader should consult Parts 1 and 2 of MAGO79a. Last, a particularly interesting claim made by Magó [MAGO80] is that parallelism in the computer overcomes the overheads associated with copying in a string reduction machine.

6.4 Utah Applicative Multiprocessing System

The Applicative Multiprocessing System (AMPS) is a loosely coupled, tree-structured computer architecture designed to incorporate a large number (say 1000) of processors. The project [KELL78, KELL79] under investigation at the University of Utah aims to increase the programmability of this parallel computer by basing its machine language on a dialect of LISP employing lenient CONS [FRIE76, HEND76]. AMPS uses dynamic strategies for allocating work to processors and also attempts to exploit locality of reference in its programs. This description of AMPS is taken from KELL79.

AMPS is based on a parallel graph reduction program organization, with paral-

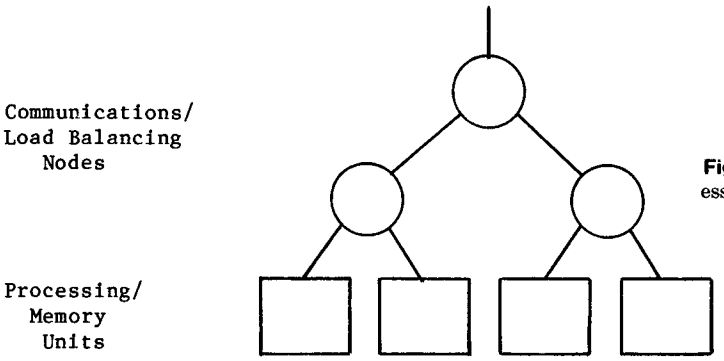

Figure 30. Applicative multiprocessing system

lelism being obtained by demanding both arguments of dyadic operators, such as PLUS, concurrently. The program organization also updates evaluated structures in place but copies subgraphs before applying them. This is necessary because execution overwrites an expression, and unless a copy is taken, a definition would be lost the first time it was used.

Program representation in AMPS is a compiled dialect of LISP called FGL (Flow Graph LISP). A program in FGL consists of a main function graph, together with what are called productions for programmer-defined functions. These productions specify how a node containing a function reference (the antecedent of the production) is to be replaced by a function graph (the consequent of the production). FGL provides a repertoire of basic operators (e.g., the primitive functions of LISP) that may be used in constructing graphs.

Programs are divided into "blocks," a block being either a code block or a data block. The contents of a code block form a linear representation of an FGL graph, which is copied as the source of initial code to be stored in a newly allocated data block. This copying may be viewed as the application of an FGL production, that is, replacing the antecedent node with its consequent graph. Each entry in a data block is either a literal value or an instruction, defining an operator and its arguments. In detail an instruction may contain four types of argument, namely, (1) an operator, (2) references to input operands, (3) so-called *notifiers*, which are references to instructions that have demanded this instruction's value, and (4) a single global reference providing linkage across blocks.

The machine organization of the AMPS computer is based on packet communication, in particular, what may be viewed as a token-matching variety. When an instruction is invoked, demand packets are dispatched for the input operands, and the instruction suspends execution. The instruction is reenabled by the arrival of result packets on which it executes. The physical arrangement of components in AMPS, shown in Figure 30, is a binary tree structure with two types of node. Combined processing/memory units are attached as leaf nodes, while the internal nodes of the tree structure are dual communication and load-balancing units.

The packet-switched communication network in AMPS is designed to take advantage of locality of information flow, to reduce communication costs. Information first travels up the tree toward the root node until it encounters a node that spans the destination leaf, at which point it proceeds down the tree. Thus relatively local communication is separated from more global flows and takes less time. In its load-balancing role, a node periodically obtains load-monitoring signals from its subordinates, which it uses to reallocate work to underutilized nodes, while attempting to maintain physical locality of references.

A processing unit, roughly the size of a conventional microcomputer, is able to execute program tasks sequentially and also to allocate storage in response to the execution of *invoke* instructions. An invoke instruction creates a task, which is then executed in the local processing unit or in another unit, as dictated by system loading. Execution of an invoke causes the allocation of storage for a data block, the copying

of a code block into the storage, and the initialization of various linkage instructions. These provide linkage between the nodes of the graph containing the antecedent of the production and those of the consequent.

Tasks to be executed (i.e., operators with their associated arguments) are placed in pools of work. There are two classes of pools:

(1) demand—containing references to operators for which evaluation is to be attempted;
(2) result—containing references to operators, along with their corresponding values after evaluation.

Each processing unit has its own demand pool, called the invoke list, but it is unclear from KELL79 whether the result pool is also distributed.

At the start of executing a program, a reference to the instruction producing the result is placed on an invoke list and the instruction is then fetched. If the arguments of the instruction are ready, then the instruction is executed; otherwise, a reference to each argument, together with a notifier so it may return the result, is placed on the invoke list. These notifiers support graph reduction by the reversal of pointers, as discussed in Section 1.4. Several notifiers may be contained in an entry in an invoke list, defining all the instructions that have demanded the result. Once evaluated, a result value replaces the instruction that calculates it. Via the result list, any instructions that were specified by notifiers as awaiting this result as an argument are then notified by being placed on an invoked list to be retried.

Current status of the project is that a simulator for the program organization has been written in PASCAL and another one, in SIMULA-67, is being written to evaluate the tree architecture. Apparently [KELL79] there are no immediate plans for construction of a physical realization of the machine.

6.5 S-K Reduction Machine

Turner's S-K reduction machine [TURN79a, TURN79b], unlike the other projects we have examined, is not strictly a proposal for a new computer architecture; instead, it is a novel implementation technique for functional languages. This work has attracted considerable attention and is sufficiently relevant to warrant discussion here. Using a result of Schonfinkel [SCHO24] from combinatory logic, Turner has devised a variable free representation for programs which contain bound variables. He has also designed a graph reduction machine that efficiently executes that representation as machine code. Our discussion of the S-K reduction machine and its use of combinators is taken from TURN79a.

The program organization of the S-K reduction machine is lazy evaluation [HEND76], based on graph manipulation with a leftmost outermost computation rule. However, the central feature of the machine design is its use of combinators, special operators that serve the role of bound variables in a program and hence allow them to be removed from the code. Let us consider the role of bound variables. A bound variable in a programming language and a corresponding reference in the machine code provide access to an object. The logical role of this reference is to associate or bring together some operand and operator at run time, since it is not physically possible to place each operand next to its operator.

Compilation into combinators removes bound variables from the program. Execution of the resulting machine code routes actual values back into the places in the program where bound variables formerly occurred. Compilation and execution are thus symmetric. The following illustrates the combinators and their transformations in the S-K machine:

Combinators	Transformations
S f g x	f x (g x)
K x y	x
C f g x	(f x) g
B f g x	f (g x)
I x	x
COND TRUE x y	x
COND FALSE x y	y

For example, the definition "DEF fac" will be represented as

```
DEF fac = S(C(B COND(EQ 0))1)
          (S TIMES(B fac(C MINUS 1))).
```

The compiler transforms each incoming expression into a variable free machine code. Code is stored as a binary tree whose internal nodes represent function applications and whose leaves will be constants such as 1, PLUS, or S. The trees are built using references, and these references may be manipulated at run time without the contents of the corresponding subtree being known. Recursive definitions are handled using an additional Y combinator. Execution of Y produces a cyclic reference at run time.

The run-time system consists of a reduction machine (currently implemented in software), which progressively transforms the combinator code as discussed above. To schedule the sequence of leftmost reductions, a *left ancestor stack*, which initially contains only (a pointer to) the expression to be evaluated, is used. This is illustrated by Figure 31. As long as the expression at the front of the stack is an application, the machine continues to take its left subtree (the function of the function–argument pair), pushing it onto the stack. Eventually an atom is at the front of the stack. If it is a combinator, then the appropriate transformation rule is applied, using the pointers on the stack to gain access to the arguments where necessary. Figure 31 shows the state of the stack before and after applyng the C transformation. All structures manipulated by the run-time system are built out of two-field cells, and a LISP-style storage allocation scheme is used with mark bits and a garbage collector.

Turner has compared his S–K reduction machine with the more conventional SECD machine of Landin [LAND64] used for implementing functional languages and has noted the following [TURN79a]. First, the object code of the S–K machine seems to be consistently twice as compact as the SECD code. Second, the execution speed of the S–K machine is slightly slower than a nonlazy SECD machine, but much superior when a lazy (evaluation) SECD machine is used. Further details of these comparisons are given in TURN79a.

6.6 Cambridge SKIM Machine

The SKIM reduction machine [CLAR80] at Cambridge University is, to our knowledge,

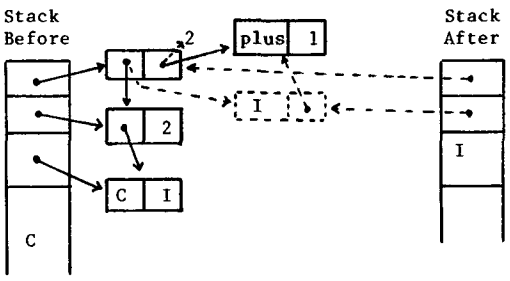

Figure 31. The S–K reduction machine's stack behavior

the first hardware graph reduction machine to be built. A conventional microprocessor is microcoded to emulate combinators as used above in the S–K reduction machine. The technique of using combinators to support applicative programming was first developed by Turner in his software reduction machine, which is described above. The SKIM machine is fully operational, and some interesting performance measurements have been obtained. This present account of the machine is based in information from Clark et al. [CLAR80].

SKIM employs lazy evaluation. Programs are evaluated outermost first and, wherever possible, common subexpressions are shared. The instruction set is similar to that of the S–K reduction machine, containing combinators (S, K, I, . . .), list operators (HD, TL, . . .), and standard operators (+, −, . . .). Programs in SKIM are represented by a graph built of two element cells. In SKIM, these are implemented by dividing the memory into two banks, HEAD and TAIL, and using a microcoded garbage collector to handle memory management. SKIM has no stacks; instead, programs are traversed by pointer reversal.

SKIM is driven by a combinator reducer that scans down the leftmost branch of the program tree to find an operator (combinator) at the leaf. When a pointer has been used to go down one level in the tree, it is reversed to indicate the return route back up the tree. Eventually a sequence of pointers from root to leaf is transformed into a sequence of pointers from leaf to root (see Figure 32). The leaf operator is now executed, using the back pointers to access its arguments in a way analogous to accessing the top few elements of a stack.

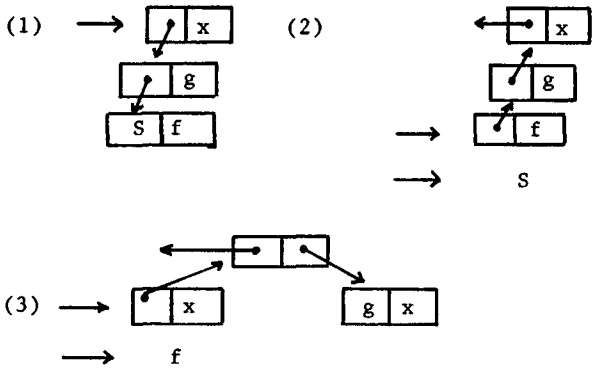

Figure 32. Pointer-reversing traversal and execution

Some operators (mainly the arithmetic and Boolean ones) are strict. That is, their arguments must be reduced to values before being used. For instance, if we wish to add together two arithmetic expressions E1 + E2, both E1 and E2 must be reduced to their values n1, n2 before the operator + can be executed. This case arises as a consequence of the outermost computation rule of lazy evaluation, which means an operator is always reached before its arguments. SKIM handles this recursive evaluation of arguments by simulating a stack by a linked list in the main HEAD-TAIL memory. This mechanism, coupled with the pointer-reversing traversal, means that no special fixed storage area is set aside for evaluation stacks.

The SKIM machine organization (see Figure 33) consists of 16 internal registers and 32K words of 16-bit memory. Only three microinstruction types are provided: memory read, memory write, and ALU operations. The microinstruction cycle time is given as 600 nanoseconds. As mentioned above, memory is divided into two banks, HEAD and TAIL. These are accessed by 15-bit addresses, one bit being used to select the appropriate bank.

The SKIM experiment has demonstrated that combinators form a simple elegant machine code to support functional programming. The main difference between SKIM and a conventional microcomputer is that it is a reduction machine. Execution progresses by rewriting the program. The performance measures obtained indicate that SKIM compares favorably with conventional architectures. For example, in comparison with BASIC on a microprocessor, SKIM was about twice as fast as interpreted BASIC and a little slower than compiled BASIC. In comparison with LISP running on a large IBM/370 mainframe, SKIM was found to be about half as fast as interpreted LISP and eight times slower than compiled LISP. The performance figures seem to justify their claim that minicomputer performance was obtained at microcomputer cost simply by using an instruction set suited to the application.

6.7 Other Projects

Research into reduction machines, although not as firmly established as dataflow computers, is starting to expand rapidly. Besides the projects described above, there are a number of others worthy of description, including those of Darlington [DARL81] at Imperial College, London, and Sleep [SLEE80, SLEE81] at the University of East Anglia, who are both investigating interesting packet communication machine organizations that support parallel graph reduction.

7. FUTURE DIRECTIONS

The research described above into data-driven and demand-driven computer architecture is motivated by the growing belief [JIPD81c] that the next, the fifth, generation of computers will not be based on the traditional von Neumann organization. The question we have been addressing is: Which architectural principles and features from

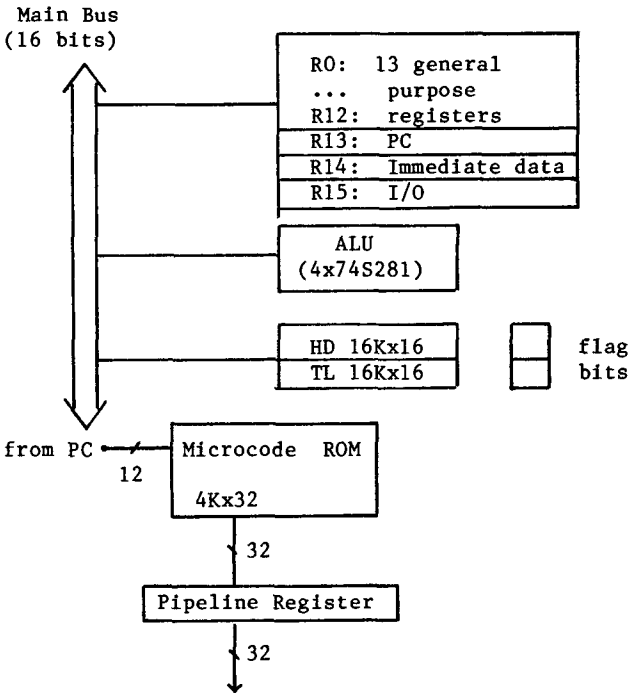

Figure 33. SKIM machine block diagram.

the various research projects will contribute to this future general-purpose computer?

One means of evaluating the potential of the various control-flow, data-flow, and reduction approaches is to compare them to the motivations for data-driven and demand-driven computing discussed in the introduction. These are

(1) utilization of concurrency;
(2) exploitation of VLSI;
(3) new forms of programming.

For the computation organization it is clear that the sequential control-driven model, which has long been predominant, has not encouraged the use of highly concurrent programs. (However, parallel control-driven computation organizations are possible.) It is also clear that the new forms of programming, such as functional languages, are naturally matched with data-driven and demand-driven computation organizations, and that these models allow utilization of concurrency. The differences between data-driven and demand-driven computation organizations are still being explored.

For the program organization it is significant that control flow, data flow, and reduction regard the by-value and by-reference data mechanisms and the sequential, parallel, and recursive control mechanisms as sets of alternatives. This results in each program organization having specific advantages and disadvantages for program representation and execution. For example, in comparing by-value and by-reference data mechanisms, the former is more effective when manipulating integers and the latter is more effective when manipulating arrays. Each program organization is suited to a particular form of programming language. Thus each program organization is, although "universal" in the sense of a Turing machine, somewhat restricted in the classes of computation it can efficiently support. We may speculate that it should be possible and indeed desirable for general-purpose computing to design computer architectures whose program organization is a synthesis of both sets of data and control mechanisms [TREL81b].

For the machine organization it is clear that centralized, packet communication, and expression manipulation gravitate toward, respectively, control flow, data flow, and reduction. However, we have shown that other pairings of the machine organizations and the program organizations are viable. When evaluating the three machine organizations against the motivations for data-driven and demand-driven computers listed above, the utilization of concurrency would seem to preclude centralized organizations in favor of the other two organizations. In addition, VLSI requires an organization in which a replication of identical computing elements can be plugged together to form a larger parallel computer. But it is also necessary for a computing element to have a centralized organization so that it can function independently. Thus the three machine organizations, instead of being competitive, seem in fact to be complementary organizations. Each organization is based on a sequential building block: a computing element containing a processor, communications, and memory. The centralized organization defines how a single computing element must be able to function as a self-contained computer. The packet communication organization shows how concurrency within a computing element may be increased by replicating resources. Last, the expression manipulation organization specifies how a group of computing elements may be interconnected, at a system level, to satisfy the VLSI attributes of replication.

In conclusion, having examined the computation organizations, program organizations, and machine organizations for control flow, data flow, and reduction, and also the approaches taken by the individual research groups, it is regrettably impossible at this time to identify the future "von Neumann." We were, however, able to analyze the advantages and disadvantages of the various approaches. Using such knowledge it is even possible to "engineer" new program organizations and machine organizations [WILN80, TREL81a].

ACKNOWLEDGMENTS

In acknowledging all the people who have contributed to the writing of this paper it is difficult not to list a significant proportion of the computing science community. First, let us thank those people investigating data-driven and demand-driven computing, whose work was discussed above, for taking time to read this document and comment on our description of their work Second, we express our gratitude to Klaus Berkling, Jean-Pierre Banatra, Al Davis, Ronan Sleep, and Graham Wood for their detailed comments on an early version of this paper. Third, we would like to thank our colleagues at the University of Newcastle upon Tyne, in particular, current and past members of the Computer Architecture Group. Fourth, we thank the referees for their helpful comments. Finally, we wish to thank the Distributed Computing Systems Panel (its current and past members) of the United Kingdom Science and Engineering Research Council, which not only funds our research but has also been largely responsible for establishing and encouraging data-driven and demand-driven computing research in the United Kingdom.

REFERENCES

ACKE79a ACKERMAN, W. B. "Data flow languages," in *Proc. 1979 Nat. Computer Conf.* (New York, N.Y., June 4-7), vol. 48, AFIPS Press, Arlington, Va, 1979, pp. 1087-1095.

ACKE79b ACKERMAN, W B., AND DENNIS, J. B. "VAL—A value oriented algorithmic language, preliminary reference manual," Tech. Rep. TR-218, Lab. for Computer Science, Massachusetts Institute of Technology, June 1979.

AMAM80 AMAMIYA, M., HASEGAWA, R., AND MIKAMI, H. "List processing and data flow machine," Lecture Note Series, No. 436, Research Institute for Mathematical Sciences, Kyoto Univ., Sept 1980.

ARVI75 ARVIND AND GOSTELOW, K. P. "A new interpreter for dataflow and its implications for computer architecture," Tech. Rep. 72, Dep Information and Computer Science, Univ. of California, Irvine, Oct. 1975.

ARVI77a ARVIND AND GOSTELOW, K. P. "A computer capable of exchanging processors for time," in *Proc IFIP Congress* (1977), 849-854.

ARVI77b ARVIND AND GOSTELOW, K P. "Some relationships between asynchronous interpreters of a dataflow language," in *Proc. IFIP Working Conf. Formal Description of Programming Languages*

(Aug. 1977), E. J. Neuhold, Ed., Elsevier North-Holland, New York, 1977.

ARVI78 ARVIND, GOSTELOW, K. P., AND PLOUFFE, W. "An asynchronous programming language and computing machine," Tech. Rep. 114a, Dep. Information and Computer Science, Univ. of California, Irvine, Dec. 1978.

ARVI80a ARVIND, KATHAIL, V., AND PINGALI, K. "A processing element for a large multiprocessor dataflow machine," in *Proc. Int. Conf Circuits and Computers* (New York, Oct. 1980), IEEE, New York, 1980.

ARVI80b ARVIND AND THOMAS, R E. "I-structures: An efficient data type for functional languages," Rep. LCS/TM-178, Lab. for Computer Science, Massachusetts Institute of Technology, June 1980.

ASHC77 ASHCROFT, E. A, AND WADGE, W. W "LUCID, a nonprocedural language with iteration," *Commun. ACM* **20,** 7 (July 1977), 519-526

BACK72 BACKUS, J. "Reduction languages and variable free programming," Rep. RJ 1010, IBM Thomas J Watson Research Center, Yorktown Heights, N Y., Apr. 1972.

BACK73 BACKUS, J. "Programming languages and closed applicative languages," in *Proc. ACM Symp Principles of Programming Languages*, ACM, New York, 1973, pp 71-86.

BACK78 BACKUS, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM* **21,** 8 (Aug. 1978), 613-641.

BERK71 BERKLING, K. J. "A computing machine based on tree structures," *IEEE Trans. Comput.* **C-20,** 4 (Jan 1971), 404-418.

BERK75 BERKLING, K. "Reduction languages for reduction machines," in *Proc. 2nd Int. Symp Computer Architecture* (Houston, Tex., Jan. 1975), IEEE, New York, 1975, pp. 133-140.

CHAM71 CHAMBERLIN, D. D. "The single assignment approach to parallel processing," in *Proc Nat. Computer Conf.* (Las Vegas, Nev., Nov. 16-18), vol. 39, AFIPS Press, Arlington, Va., 1971, pp. 263-269.

CLAR80 CLARKE, T. J. W., GLADSTONE, P. J S, MACLEAN, C. D., AND NORMAN, A. C. "SKIM—The S, K, I reduction machine," in *Proc. LISP-80 Conf.* (Stanford, Calif., Aug. 1980), pp. 128-135

COMT76 COMTE, D., DURRIEU, A., GELLY, O., PLAS, A., AND SYRE, J C. "TEAU 9/7: SYSTEME LAU—Summary in English," CERT Tech. Rep. #1/3059, Centre d'Études et de Recherches de Toulouse, Oct. 1976.

COMT79b COMTE, D., AND HIFDI, N. "LAU Multiprocessor. Microfunctional description and technological choices," in *Proc. 1st European Conf. Parallel and Distributed Processing* (Toulouse, France, Feb. 1979), pp. 8-15.

CORN79 CORNISH, M. "The TI data flow architectures: The power of concurrency for avionics," in *Proc. 3rd Conf. Digital Avionics Systems* (Fort Worth, Tex., Nov. 1979), IEEE, New York, 1979, pp. 19-25

DARL81 DARLINGTON, J., AND REEVE, M. "ALICE· A multiprocessor reduction machine for the parallel evaluation of applicative languages," in *Proc Int. Symp Functional Programming Languages and Computer Architecture* (Goteborg, Sweden, June 1981), pp. 32-62.

DAVI78 DAVIS, A. L. "The architecture and system method of DDM1: A recursively structured data driven machine," in *Proc. 5th Annu. Symp. Computer Architecture* (Palo Alto, Calif., Apr. 3-5), ACM, New York, 1978, pp. 210-215.

DAVI79a DAVIS, A. L. "DDN's—A low level program schema for fully distributed systems," in *Proc. 1st European Conf. Parallel and Distributed Processing* (Toulouse, France, Feb. 1979), pp. 1-7.

DAVI79b DAVIS, A. L. "A data flow evaluation system based on the concept of recursive locality," in *Proc. 1979 Nat. Computer Conf.* (New York, N.Y., June 4-7), vol. 48, AFIPS Press, Arlington, Va., 1979, pp. 1079-1086

DENN71 DENNIS, J. B. "On the design and specification of a common base language," in *Proc. Symp Computers and Automata*, Polytechnic Institute of Brooklyn, Brooklyn, N.Y , 1971.

DENN72 DENNIS, J. B., FOSSEEN, J. B., AND LINDERMAN, J. P. "Data flow schemas," in *Int. Symp. on Theoretical Programming*, A. Ershov and V. A. Nepomniascuy, Eds., *Lecture notes in computer science,* vol. 5, 1972, Springer-Verlag, New York, pp. 187-216.

DENN74a DENNIS, J. B. "First version of a data flow procedure language," in *Programming Symp.: Proc. Colloque sur la Programmation* (Paris, France, Apr. 1974), B Robinet, Ed., *Lecture notes in computer science*, vol. 19, Springer-Verlag, New York, 1974, pp. 362-376.

DENN74b DENNIS, J. B., AND MISUNAS, D. P. "A computer architecture for highly parallel signal processing," in *Proc. 1974 Nat. Computer Conf.*, AFIPS Press, Arlington, Va , 1974, pp. 402-409.

DENN75b DENNIS, J. B., AND MISUNAS, D. P "A preliminary architecture for a basic data flow processor," in *Proc. 2nd Int Symp.*

Computer Architecture (Houston, Tex, Jan. 20-22), IEEE, New York, 1975, pp 126-132.

DENN79a DENNIS, J. B., LEUNG, C. K. C., AND MISUNAS, D P. "A highly parallel processor using a data flow machine language," Tech. Rep. CSG Memo 134-1, Lab. for Computer Science, Massachusetts Institute of Technology, June 1979

DENN79b DENNIS, J B. "The varieties of data flow computers," in *Proc 1st Int. Conf Distributed Computing Systems* (Toulouse, France, Oct. 1979), pp. 430-439.

EGAN79 EGAN, G. K. "FLO: A decentralised data-flow system," Dep. Computer Science, Univ. of Manchester, England, Oct. 1979

ERIK80 ERIKIO, L., HEIMONEN, J HIETALA, P., AND KURKI-SUONIO, R. "PLEXUS II—A data flow system," Tech. Rep. A43, Dep. Mathematical Sciences, Univ. of Tampere, Finland, Apr. 1980.

FARR79 FARRELL, E. P., GHANI, N., AND TRELEAVEN, P. C "A concurrent computer architecture and a ring based implementation," in *Proc. 6th Int. Symp. Computer Architecture* (April 23-25), IEEE, New York, 1979, pp. 1-11.

FRIE76 FRIEDMAN, D. P., AND WISE, D. S. "CONS should not evaluate its arguments." in *Automata, languages and programming*, S. Michaelson and R. Milner, Eds., Edinburgh Univ. Press, Edinburgh, U. K., 1976, pp. 257-284.

GELL76 GELLY, O, et al. "LAU software system. A high level data driven language for parallel programming," in *Proc. 1976 Int. Conf. Parallel Processing* (Aug 1976), p. 255

GOST79a GOSTELOW, K P., AND THOMAS, R. E. "A view of dataflow," in *Proc. Nat Computer Conf* (New York, N.Y., June 4-7), vol 48, AFIPS Press, Arlington, Va., 1979, pp. 629-636.

GOST79b GOSTELOW, K. P., AND THOMAS, R E "Performance of a dataflow computer," Tech. Rep. 127a, Dep Information and Computer Science, Univ. of California, Irvine, Oct. 1979

GOTO77 GOTO, E., AND IDA, T. "Parallel hashing algorithms" *Inf Process Lett.* **6**, 1 (Feb. 1977), pp 8-13.

HEND76 HENDERSON, P., AND MORRIS, J M. "A lazy evaluator," in *Proc 3rd Symp Principles of Programming Languages* (Atlanta, Ga., Jan. 19-21), ACM, New York, 1976, pp. 95-103.

HOMM79 HOMMES, F., AND SCHLUTTER, H "Reduction machine system User's guide," Tech. Rep. ISF—Rep. 79, Gesellschaft fur Mathematik und Datenverarbeitung MBH Bonn, Dec. 1979.

HOPK79 HOPKINS, R. P., RAUTENBACH, P. W., AND TRELEAVEN, P C "A computer supporting data flow, control flow and updateable memory," Tech. Rep. 156, Computing Lab., Univ. Newcastle upon Tyne, Sept. 1979.

JIPD81a JIPDC. "Preliminary report on study and research on fifth-generation computers 1970-1980," Japan Information Processing Development Center, Tokyo, Japan, 1981.

JIPD81b JIPDC. "Research reports in Japan," Japan Information Processing Development Center, Tokyo, Japan, Fall 1981

JIPD81c JIPDC. in *Proc. Int. Conf. Fifth Generation Computer Systems*, Japan Information Processing Development Center, 1981

JOHN79 JOHNSON, D., et al "Automatic partitioning of programs in multiprocessor systems," in *Proc. IEEE COMPCON 80* (Feb. 1980), IEEE, New York, pp 175-178

KELL78 KELLER, R. M., PATIL, S, AND LINDSTROM, G. "An architecture for a loosely coupled parallel processor," Tech. Rep. UUCS-78-105, Dep. Computer Science, Univ. of Utah, Oct. 1978.

KELL79 KELLER, R. M, et al. "A loosely coupled applicative multiprocessing system," in *Proc. Nat Computer Conf.*, AFIPS Press, Arlington, Va., 1978, pp. 861-870.

KLUG79 KLUGE, W. E. "The architecture of a reduction language machine hardware model," Tech. Rep. ISF—Rep 79.03, Gesellschaft fur Mathematik und Datenverarbeitung MBH Bonn, Aug. 1979.

KLUG80 KLUGE, W. E., AND SCHLUTTER, H "An architecture for the direct execution of reduction languages," in *Proc. Int Workshop High-Level Language Computer Architecture* (Fort Lauderdale, Fla., May 1980), Univ of Maryland and Office of Naval Research, pp. 174-180.

LAND64 LANDIN, P. J "The mechanical evaluation of expressions," *Comput J.* **6** (Jan. 1964), 308-320.

LECO79 LECOUFFE, M. P. "MAUD. A dynamic single-assignment system," *IEE Comput Digital Tech.* **2**, 2 (Apr 1979), 75-79.

MAGO79a MAGÓ, G. A. "A network of microprocessors to execute reduction languages," *Int. J Comput. Inform. Sci.* **8**, 5 (1979), 349-385, **8**, 6 (1979), 435-471.

MAGO80 MAGÓ, G. A "A cellular computer architecture for functional programming," in *Proc. IEEE COMPCON 80* (Feb. 1980), IEEE, New York, pp 179-187.

MCCA62 MCCARTHY, J., et al. *LISP 1 5 programmers manual*, M.I.T. Press, Cambridge, Mass., 1962

MEAD80 MEAD, C A, AND CONWAY, L. A *Introduction to VLSI systems*, Addison-Wesley, Reading, Mass., 1980.

MIRA77 MIRANKER, G. S "Implementation of procedures on a class of data flow processors," in *Proc. 1977 Int. Conf. Parallel Processing* (Aug. 1977), J. L. Baer, Ed., IEEE, New York, pp 77-86

ORGA79 ORGANICK, E. I. "New directions in computer system architecture," *Euromicro J* **5,** 4 (July 1979), 190-202.

PLAS76 PLAS, A., et al "LAU system architecture· A parallel data driven processor based on single assignment," in *Proc. 1976 Int. Conf. Parallel Processing* (Aug. 1976), pp. 293-302.

RODR69 RODRIGUEZ, J. E. "A graph model for parallel computation," Tech. Rep. ESL-R-398, MAC-TR-64, Lab for Computer Science, Massachusetts Institute of Technology, Sept 1969.

RUMB77 RUMBAUGH, J. E "A data flow multiprocessor," *IEEE Trans. Comput.* **C-26,** 2 (Feb 1977), 138-146.

SCHO24 SCHONFINKEL, M. "Uber die Bausteine der Mathematischen Logik," *Math. Ann* **92,** 305 (1924).

SEIT79 SEITZ, C. (Ed) *Proc. Conf Very Large Scale Integration* (Pasadena, Calif., Jan. 1979).

SHRO77 SHROEDER, M. A., AND MEYER, R. A "A distributed computer system using a data flow approach," *Proc. 1977 Int. Conf Parallel Processing* (Aug. 1977), p. 93.

SLEE80 SLEEP, M R "Applicative languages, dataflow and pure combinatory code," *Proc IEEE COMPCON 80* (Feb. 1980), IEEE, New York, pp. 112-115.

SLEE81 SLEEP, M. R, AND BURTON, F. W. "Towards a zero assignment parallel processor," in *Proc. 2nd Int Conf. Distributed Computing* (Apr. 1981)

SYRE76 SYRE, J. C, et al. "Parallelism, control and synchronization expression in a single assignment language" (abstract), in *Proc. 4th Annu. ACM Computer Science Conf.* (Feb 1976), ACM, New York.

SYRE77 SYRE, J. C., COMTE, D., AND HIFDI, N. "Pipelining, parallelism and asynchronism in the LAU system," in *Proc 1977 Int Conf Parallel Processing* (Aug. 1977), pp. 87-92.

TESL68 TESLER, L. G., AND ENEA, H. J. "A language design for concurrent processes," in *Proc. Nat. Computer Conf* (Atlantic City, N.J., April 30-May 2), vol 32, AFIPS Press, Arlington, Va, 1968, 403-408.

TREL78 TRELEAVEN, P. C "Principle components of a data flow computer," *Proc 1978 Euromicro Symp.* (Munich, W. Germany, Oct 1978), pp. 366-374.

TREL79 TRELEAVEN, P. C. "Exploiting program concurrency in computing systems," *Computer* **12,** 1 (Jan. 1979), 42-49

TREL80a TRELEAVEN, P C, AND MOLE, G. F. "A multi-processor reduction machine for user-defined reduction languages," in *Proc. 7th Int. Symp. Computer Architecture* (May 6-8), IEEE, New York, 1980, pp. 121-130.

TREL80b TRELEAVEN, P. C (Ed) "VLSI: Machine architecture and very high level languages," Tech. Rep. 156, Computing Lab, Univ. of Newcastle upon Tyne, Dec. 1980 (summary in *SIGARCH Comput. Archit News 8,* 7, 1980).

TREL81a TRELEAVEN, P. C., AND HOPKINS, R. P. "A recursive (VLSI) computer architecture," Tech. Rep. 161, Computing Lab., Univ. of Newcastle upon Tyne, Mar. 1981.

TREL81b TRELEAVEN, P. C., AND HOPKINS, R. P. "Decentralised computation," in *Proc. 8th Int. Symp. Computer Architecture* (Minneapolis, Minn., May 12-14), ACM, New York, 1981, pp. 279-290.

TREL82 TRELEAVEN, P. C., HOPKINS, R. P., AND RAUTENBACH, P W. "Combining data flow and control flow computing," *Comput. J.* **25,** 1 (Feb 1982).

TURN79a TURNER, D. A. "A new implementation technique for applicative languages," *Soft. Pract. Exper.* **9** (Sept, 1979), 31-49.

TURN79b TURNER, D. A. "Another algorithm for bracket abstraction," *J. Symbol Logic* **44,** 2 (June 1979), 267-270.

VEEN80 VEEN, A. H. "Reconciling data flow machines and conventional languages," Tech. Rep 1W 146/80, Mathematical Center, Amsterdam, Sept 1980.

WATS79 WATSON, I., AND GURD, J "A prototype data flow computer with token labeling," in *Proc Nat Computer Conf.* (New York, N.Y., June 4-7), vol. 48, AFIPS Press, Arlington, Va., 1979, pp. 623-628.

WENG75 WENG, K. S. "Stream-oriented computation in recursive data flow schemas," Tech. Rep TM-68, Lab. for Computer Science, Massachusetts Institute of Technology, Oct 1975

WILN80 WILNER, W "Recursive machines," Intern. Rep., Xerox PARC, Palo Alto, Calif., 1980.

BIBLIOGRAPHY

ADAM68 ADAMS, D. A. "A computation model with data flow sequencing," Tech. Rep. CS 117, Computer Science Dep., Stanford Univ, Stanford, Calif., December 1968

ARVI77c ARVIND, GOSTELOW, K. P., AND PLOUFFE, W. "Indeterminacy, monitors and dataflow," in *Proc. 6th ACM Symp Operating Systems Principles* (Nov. 1977), ACM, New York, pp. 159-169.

BAHR72 BAHRS, A. "Operational patterns: An extensible model of an extensible language," in *Lecture notes in computer science*, vol. 5, Springer-Verlag, New York, 1972, pp. 217–246.

BANA79 BANATRE, J. P., ROUTEAU, J. P., AND TRILLING, L. "An event-driven compiling technique," *Commun. ACM* **22**, 1 (Jan. 1979), 34–42.

BOLE80 BOLEY, H. "A preliminary survey of artificial intelligence machines," Rundbrief der Fachgruppe Künstliche Intelligenz in der Gesellschaft fur Informatik, Universität Hamburg, 1980.

BURG75 BURGE, W. H. *Recursive programming techniques*, Addison-Wesley, Reading, Mass., 1975.

CHUR41 CHURCH, A. *"The calculi of lambda-conversion,"* Princeton Univ. Press, Princeton, N.J , 1941.

DARL82 DARLINGTON, J., HENDERSON, P., AND TURNER, A., EDS. *Functional programming and its applications*, Cambridge Univ. Press, in preparation.

DAVI80 DAVIS, A. L , AND DRONGOWSKI, P. J. "Dataflow computers: A tutorial and survey," Tech. Rep. UUCS-80-109, Dep. Computer Science, Univ. of Utah, July 1980.

DAVI81 DAVIS, A. L., AND LOWER, S. A. "A sample management application program in a graphical data-driven programming language," in *Proc. IEEE COMPCON 81* (Feb. 1981), IEEE, New York, pp. 162–165.

DENN75a DENNIS, J. B. "Packet communication architecture," in *Proc. 1975 Computer Conf. Parallel Processing*, 1975, pp. 224–229.

DENN77 DENNIS, J B., AND WENG, K.-S. "Application of data flow computation to the weather problem." in *Proc. Symp. High Speed Computer and Algorithm Organisation*, 1977, pp 143–157.

DENN80 DENNIS, J. B. "Data-flow supercomputers," *Computer* **13**, 11 (Nov. 1980), 48–56.

DOMA81 DOMAN, A. "PARADOCS: A highly parallel dataflow computer and its dataflow language," *Euromicro J.* **7** (1981), 20–31

FRIE77 FRIEDMAN, D. P., AND WISE, D. S. "Aspects of applicative programming for file systems," *ACM SIGPLAN Not.* **12**, 3 (Mar. 1977), 41–55.

FRIE78 FRIEDMAN, D. P., AND WISE, D. S. "Aspects of applicative programming for parallel processing," *IEEE Trans. Comput.* **C-27**, 4 (Apr. 1978), 289–296.

GAJS81 GAJSKI, D. D., et al. "Dependence driven computation," in *Proc. IEEE COMPCON 81* (Feb. 1981), IEEE, New York, pp. 156–161.

GURD78 GURD, J., AND WATSON, I. "A multi-layered data flow architecture," in *Proc. 1977 Int. Conf Parallel Processing* (Aug. 1977), p. 94.

HEWI77 HEWITT, C. E., AND BAKER, H. "Actors and continuous functionals," in *Proc. IFIP Working Conf Formal Description of Programming Concepts* (St. Andrews, N. B., Canada, Aug 1977), E J. Neuhold, Ed., Elsevier North-Holland, New York, 1977, pp. 16.1–16.21.

KARP66 KARP, R. M., AND MILLER, R. E. "Properties of a model for parallel computations: Determinacy, termination and queuing," *SIAM J. Appl. Math.* **11**, 6 (Nov. 1966), 1390–1411.

KARP69 KARP, R. M., AND MILLER, R. E. "Parallel program schemata," *J. Comput. Syst. Sci.* **3**, 4 (May 1969), 147–195.

KOSI73a KOSINSKI, P. R. "A data flow programming language," Tech. Rep. RC 4264, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., Mar. 1973.

KOSI73b KOSINSKI, P. R. "A data flow language for operating system programming," *ACM SIGPLAN Not* **8**, 9 (Sept. 1973), 89–94.

KOTO80 KOTOV, V. E. "On basic parallel language," in *Proc. IFIP 80 Congr.* (Tokyo, Japan and Melbourne, Australia), Elsevier North-Holland, New York, 1980.

KOWA79 KOWALSKI, R. "Algorithms = logic + control," *Commun. ACM* **22**, 7 (July 1979), 424–436

MAGO79b MAGÓ, G A. "A cellular, language directed computer architecture," in *Proc. Conf. Very Large Scale Integration* (Pasadena, Calif., Jan. 1979), pp. 447–452.

MAGO81 MAGÓ, G. A., STANAT, D. E., AND KOSTER, A. "Program execution on a cellular computer Some matrix algorithms," Tech. Rep., Dep. Computer Science, Univ. of North Carolina, Chapel Hill, May 1981.

MANN74 MANNA, Z. *Mathematical theory of computation*, McGraw-Hill, New York, 1974.

MEYE76 MEYER, S. C. "An analytic approach to performance analysis for a class of data flow processors," in *Proc 1976 Int. Conf. Parallel Processing* (Aug. 1976), pp. 106–115.

MILL72 MILLER, R. E., AND COCKE, J. "Configurable computers· A new class of general purpose machines," in *Lecture notes in computer science*, vol. 5, Springer-Verlag, New York, 1972, pp. 285–298.

MISU75a MISUNAS, D. P. "Deadlock avoidance in a data-flow architecture," in *Proc. Symp. Automatic Computation and Control* (Milwaukee, Wis., Apr. 1975).

MISU75b MISUNAS, D. P "Structure processing in a data flow computer," in *Proc. 1975 Int. Conf Parallel Processing* (Aug. 1975), pp. 230–234.

MISU76 MISUNAS, D. P. "Error detection and recovery in a data-flow computer," in *Proc. 1976 Int. Conf. Parallel Processing* (Aug. 1976), pp. 117–122.

SHAR80 SHARP, J. A. "Some thoughts on data flow architectures." *SIGARCH Comput Archit News* (ACM) **8**, 4 (June 1980), 11–21

SHRI78 SHRIVER, B. D., AND LANDRY, S. P. "An overview of dataflow related research," Tech. Rep. Dep. Computer Science, Univ of Southwestern Louisiana, 1978

WENG79 WENG, K. S. "An abstract implementation for a generalized data flow language," Tech. Rep. TR-228, Lab. for Computer Science, Massachusetts Institute of Technology, May 1979

Received June 1981; final revision accepted November 1981