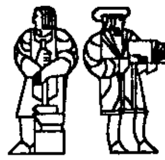
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Programming Generality, Parallelism and Computer Architecture

Computation Structures Group Memo 32

Jack B. Dennis

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MAC-M-409

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

COMPUTATION STRCUTURES GROUP MEMO NO. 32

PROGRAMMING GENERALITY, PARALLELISM AND COMPUTER ARCHITECTURE

Jack B. Dennis

PROGRAMMING GENERALITY, PARALLELISM AND COMPUTER ARCHITECTURE *

JACK B. DENNIS

Abstract

Two current trends in computing fortell a major transformation in the architecture of computer systems. One is the increasing importance of concurrency or parallelism, both to improve hardware utilization through multiprogramming, and to heighten the useability of a computer through "multi-access" operation. The other is "programming generality" - the independence of an algorithm description of the environment in which it is used. Programming generality is the ability to move a program between computer installations; the ability to maintain a program within changing hardware; the ability to use a program in the construction of another - without altering the program description in any way.

To exhibit programming generality a computer system must permit a program module to 1) create information structures of arbitrary extent; 2) call on procedures with unknown requirements for storage and information structures; and 3) transmit information structures of arbitrary complexity to a called procedure. These requirements imply that the hardware or operating system must make storage allocation decisions and, hence, that location-independent addressing must be used. A small unit of storage

---

*An early version of this paper was submitted to IFIP Congress '68 and a condensed version will appear in the Proceedings of the Congress.

allocation is necessary to prevent waste of valuable storage capacity and unnecessary information transfer. Exploitation of detailed parallelism is then essential to fully utilize the processing hardware.

A naming scheme is a set of rules for relating instances of identifiers in algorithm descriptions to information units represented in a computer system memory. To obtain programming generality, a naming scheme must provide for 1) referencing components of information structures, 2) sharing information structures among computations, 3) extending information structures indefinitely, and 4) transmitting arbitrary structures as procedure parameters.

A graph model of programs is given that satisfies the naming requirements of programming generality and exhibits the detailed parallelism of a computation. The nodes of a program graph represent operators; the arcs indicate the flow of data from one operator to another. Some node types permit reference to information structures that are subtrees of an environment tree. The apply node permits application of subordinate program graph procedures.

A computer organization is proposed in which a machine language procedure is simply a transliteration of a program graph. A multiple-level memory system is suggested which employs associative, location-independent, addressing. The processing hardware consists of many autonomous cells that employ service-on-demand control.

Concurrency and Parallelism

The trend toward increasingly greater parallelism in computer systems is a consequence of several forces now widely recognized: One is the problem of keeping the various parts of a computer system productively busy a high fraction of the time. In a general purpose system, ensuring high equipment utilization requires the allocation of the hardware to a collection of computation tasks so that resources not required by one computation are available to the others. In this way, a large population of tasks running on a modular hardware system with suitable multiplexing features can achieve a statistically high hardware utilization. This principle established the design philosophy of early multiprogrammed computers, notably the Burroughs D825 and the Honeywell 800 systems.

Parallelism is also important because it permits sharing of information, e.g. algorithmic procedures, among many parts of a task or among different computations, thus reducing the memory hardware for a specified throughput.

The rapid introduction and exploitation of "interactive computer systems" has emphasized the significance of these two points.

From the hardware technology viewpoint, many authors have argued the necessity of greater exploitation of parallelism and principles of modular system organization to reap the rewards promised by the large-scale integrated circuit technology.

Parallelism within a single computation is a common subject of study. It is known [19] that it is possible to describe computations so as to run in parallel on asynchronously timed equipment without loss of repeatability.

Programming Generality

Because the cost of program development has become the dominant expense in the use of computers, more and more emphasis is being placed on developing and maintaining programs with less human effort. This trend will undoubtedly continue as hardware innovations further reduce the cost of computing, and applications demanding sophisticated software proliferate.

The "software problem" shows up in three circumstances:

1) When one attempts to run a program at a different installation from the one at which it was developed, whether or not the hardware is of the same model.
2) When a hardware system is replaced with new equipment at a given installation.
3) When one tries to use one program as a component in building more sophisticated programs.

In the first instance, the issue is the convenient exbhange of programs among users of different installations. In the second, it is the maintainance of programs within a changing environment. In the third, it is the use of checked out procedures as building blocks for new programs as a potentially powerful means of reducing the human effort required to develop large programs. Indeed, as application programs progress in sophistication it will make less and less sense for the programming effort to start from scratch. This latter aspect of programming generality contains elements of the first two, and has far-reaching implications for programming languages and machine organization.

Composition of Programs

Suppose the author of a procedure P wishes to make use of a procedure Q. We assume the author of Q had no knowledge of the manner in which procedure Q would be used in the construction of P. Further, we ask that it be unnecessary for the author of P to know the internal workings of program Q in order to use it. He must, of course, be familiar with the interface requirements of Q--i.e. the types and valid domains of its parameters, and he must have sufficient understanding of the method used in Q to apply it intelligently -- but he should not be required to understand its implementation.

We insist that these assumptions hold regardless of the size and complexity of procedure Q, and regardless of the manner in which it, in turn, is built from other independently written procedures. In general, we must provide for the following characteristics of these procedures.

1) They create information structure of arbitrary size not known prior to their execution.
2) They call on further procedures unknown to their own caller. These procedures will in turn generate demands for storage and for other procedures.
3) They require transmission of elaborate information structures as arguments.

In the next sections we study how these characteristics affect the organization of computer memory systems and the naming of information structures.

Memory organization

Present and future computer systems are forced by economics to use memory hierarchies composed of levels representing various compromises between access time and capacity. Large application programs will have to operate (as at present) with their information structures distributed among the levels of the memory hierarchy. The following argument is to show that programming generality requires:

> The computer system (hardware or operating software) rather than the designer or user of a procedure must decide where information items should reside within the memory hierarchy

Suppose procedure Q is used in the construction of procedure P. During their execution, P and Q both may generate and process information structures of arbitrary complexity. The extent of the information generated by either routine is unknown to the designer of the other. The author of P cannot decide where the internally generated information structures of Q should reside, because he is not aware of their actual extent, or the times during which they would be called into activity. Similarly, the author of Q is unable to make storage allocation decisions for information structures created by P. Neither procedure may make its own storage assignments independently of the other since these decisions would eventually conflict. Similar reasoning precludes a compiler from making storage allocation decisions. Hence the division of information structures among the levels of physical storage is a responsibility of the computer system.

A second conclusion is an immediate consequence of the first:

> The referencing of information structures by a procedure must be by means of a location-independent addressing mechanism.

If the system is to have sufficient freedom in making its storage allocation decisions, information references within a procedure must be specified in a way not dependent on where the information is stored.

These considerations also lead to the following conclusion:

> Information can be moved upward in the memory hierarchy only on demand, that is, upon being referenced by an active computation.

In the example of procedure Q used within procedure P, it is not possible for procedure P to anticipate Q's use of information structures without knowledge of the construction of Q. Neither procedure can make a valid decision about its own information without knowledge of the other.

Once we have agreed that the system must control storage allocation, and should move information on demand, the question of what size unit of information should be chosen for storage allocation. We claim that:

> The unit of storage allocation should be the information unit upon which the primitive operations of the computer system are carried out, i.e., the <u>word</u>.

The choice of a larger unit of allocation will always cause some information to be brought into a higher memory level and not subsequently referenced by an active computation. The cost of a larger information unit is wasted cells at high memory levels and unnecessary information transfer. The choice of such a unit as a word can be made practical by radical redesign of auxiliary storage systems and the way they interact with the processing hardware. These storage systems must be capable of a high data transmission rate, and must deal efficiently with hundreds or even thousands of simultaneously pending retrieval requests.

Assuming information units are retrieved on demand, and a small unit of information is used, monosequence execution of a moderate size procedure would be intolerably slow. Suppose a procedure is represented by N information units (<u>items</u>) initially residing in a level of memory having mean retrieval time T. Then execution of the procedure (assuming all N items require reference) would take at least $N \times T$ time units. Thus we conlude:

> The exploitation of parallelism at the detailed level is essential to the efficient achievement of programming generality.

Embrionic forms of these ideas have already appeared in computer system designs. Location-independent addressing and the notion of paging on demand were introduced by the designers of Atlas [10], and

have since been adopted by an impressive number of computer architects. In one instance, the Multics system [3], the notion of a file as a distinct class of information structure has been abandoned in favor of a uniform addressing scheme in which all information structures are treated alike [7, 4].

Associative addressing techniques are becoming more common in computer system design. Parallel search memories are a part of most address transformation schemes now in use. Software implemented associative addressing has long been used in operating systems for locating files or items within files.

The realization of parallel execution of computation at the detailed level requires a radical departure from conventional organization of the information processing hardware. Machines such as the CDC 6600 and the IBM 360/91 are interesting in that they exploit any potential parallelism that is discovered by looking ahead a short distance in the instruction sequence. Yet these are essentially monosequence processors. Several rather unusual computer designs have been proposed - but none have paid significant heed to the demands of programming generality. The SOLOMON (and later, the Illiac IV) machine [18] is regarded by its designers as a special purpose machine. The organization of IMP [1] permits the sharing of processing hardware (e.g. arithmetic operations) among several monosequence programs. Nevertheless, to each program the machine appears conventional.

A less radical concept is building a computer system as a symmetrical arrangement of memory modules and conventional processing units, a system design introduced by the Burroughs B5000 and D825 systems. This structure can only process in parallel at a gross level because of the cost of switching the activity of a processor. Also, the interconnection schemes do not readily generalize to large numbers of modules.

Naming Schemes

A naming scheme is a set of rules for relating instances of identifiers in procedures to items represented in a computer system memory. In order to meet the requirements of programming generality, a naming scheme must provide:

1) A means for referencing components of structures, which may themselves be structures.

2) A means of sharing information structures among computations or parts of a computation.

3) A means of extending information structures indefinitely by appending new parts.

4) A mechanism whereby ability to reference an arbitrary structure may be transmitted between procedures by a parameter.

A representation of procedures and data structures which meets these naming requirements is discussed in the following paragraphs. It is closely related to the environment concepts of Landin [11] and Burge [2], and the "definition sets" of Illife [9].

Information structures

The environment contains as a part each information structure existing in a computer system. The environment tree is a directed, acyclic graph in which there exists at least one directed path to each node from a particular node called the root of the environment. We use the word "tree" with reluctance, as there may be distinct paths to nodes from the root node. Since the nodes of the environment tree can be partially ordered, we indicate branch direction by the relative position of the connected nodes, as for example in Fig. 1. A subtree of the environment is any node of the environment (the root node of the subtree) together with all nodes to which it is connected by a directed path. An information structure is a subtree of the environment.

Each branch of the environment is labelled with a branch name which must distinguish it from all other branches originating from the same node. A branch originating at the root node of one information structure terminates on the root node of a second information structure which is a component of the firs

The leaves of the environment tree are individual items of information - data values or instructions. Subtrees are files, records, arrays or procedures. An item (or structure) may be identified by the string of branch names that selects a path from the root of the environment tree to the node of interest. The item may also be identified in the context of some subtree containing it (or, as we shall say, relative to the root node of the subtree) by the string of branch names that selects a path to the

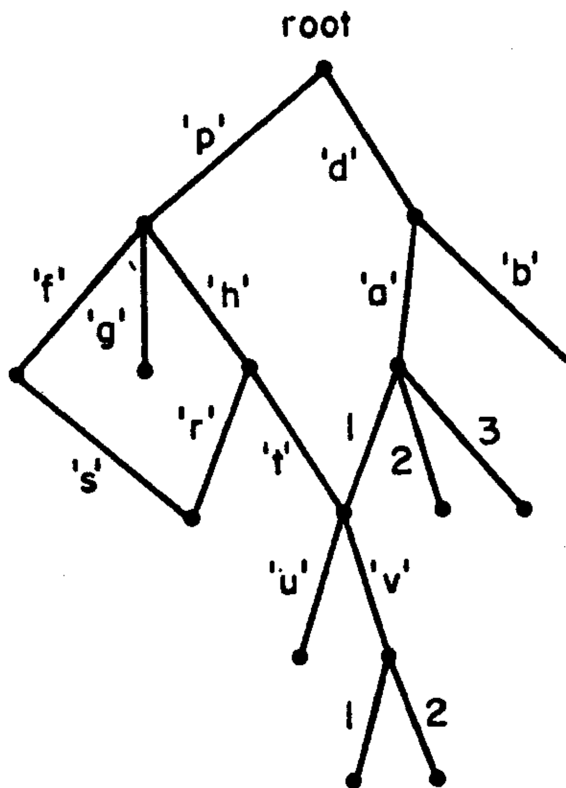


Fig. 1. An environment.

item from the root node of the subtree. Sharing of an information structure is accomplished by permitting a subtree to be a component of many nodes of the environment. The tree form of the environment makes it possible to append an arbitrary structure as a component of a higher level structure without introducing conflicts with the component's internal identifiers.

The transfer of ability to reference an information structure is realized through a data type we shall call pointer. A pointer value uniquely specifies an information structure of the environment and is to be associated with the root node of the structure. A pointer has the property that it specifies the same information structure regardless of the situation in which it is used. Thus a pointer may be passed as a parameter through any number of procedure interfaces without loss of its meaning. This is in contrast to branch names whose meanings are very much dependent on the context of their use.

Another statement of these naming considerations has been given in [8] where the term "capability" is used in place of "pointer".

Program graphs

A program graph is a way of representing expressions. For example, the program graph in Fig. 2 represents the computation

$$(x-y)^2 \div (x+y)^2 \rightarrow z$$

Some nodes of a program graph represent operators that transform a set of input values (usually two) into a result. The arcs of the graph are

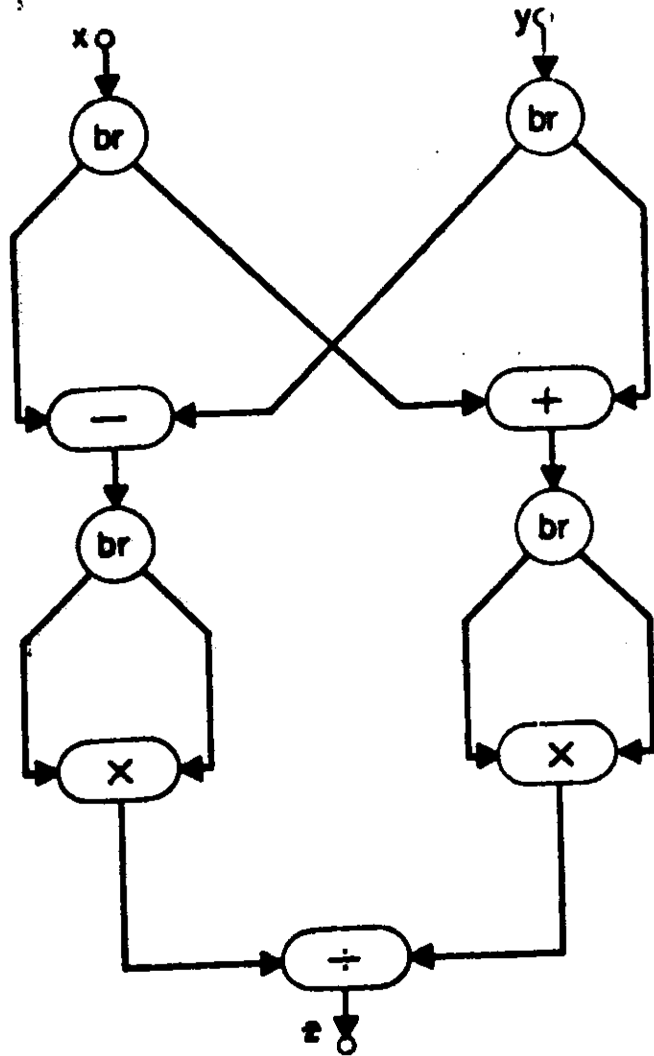


Fig. 2. An elementary program graph.

called links. Each link connects an output of a node to an input of one other node; the first node is a predecessor of the second; the second is a successor of the first. The branch node has two output links and transmits a data value to two successor nodes when a result must be passed to several operators. A node carries out its function just when all of the required input values have arrived. It performs its computation asynchronously with respect to other nodes and, when finished, transmits the output values to its successors.

Rodriguez [16] has studied a more general form of program graph model that encompasses decision operators and iteration. His work does not cover structured information; since this aspect is of particular interest to us we prefer to start from the more elementary model to keep the discussion tractable. Rodriguez has shown that his model, interpreted according to rules that permit asynchronous concurrency, provides a deterministic description of computations. That is, the computed results are independent of the relative timing with which the allowed parallelism occurs. Similar program models have been studied by Martin and Estrin [14], and by Sutherland [19], but with different objectives.

Extended Program Graphs

To handle information structures, we postulate additional building blocks for program graphs. Pointers and branch names become new data types that can appear as input or output values of a program graph node. Links that convey pointer values will be shown as heavy lines. Each pointer link connects a superior node to an inferior node. The new node types are shown in Fig. 3.

The fetch node takes a pointer p and a branch name n as inputs and produces a data value v. The result v is the value of the component

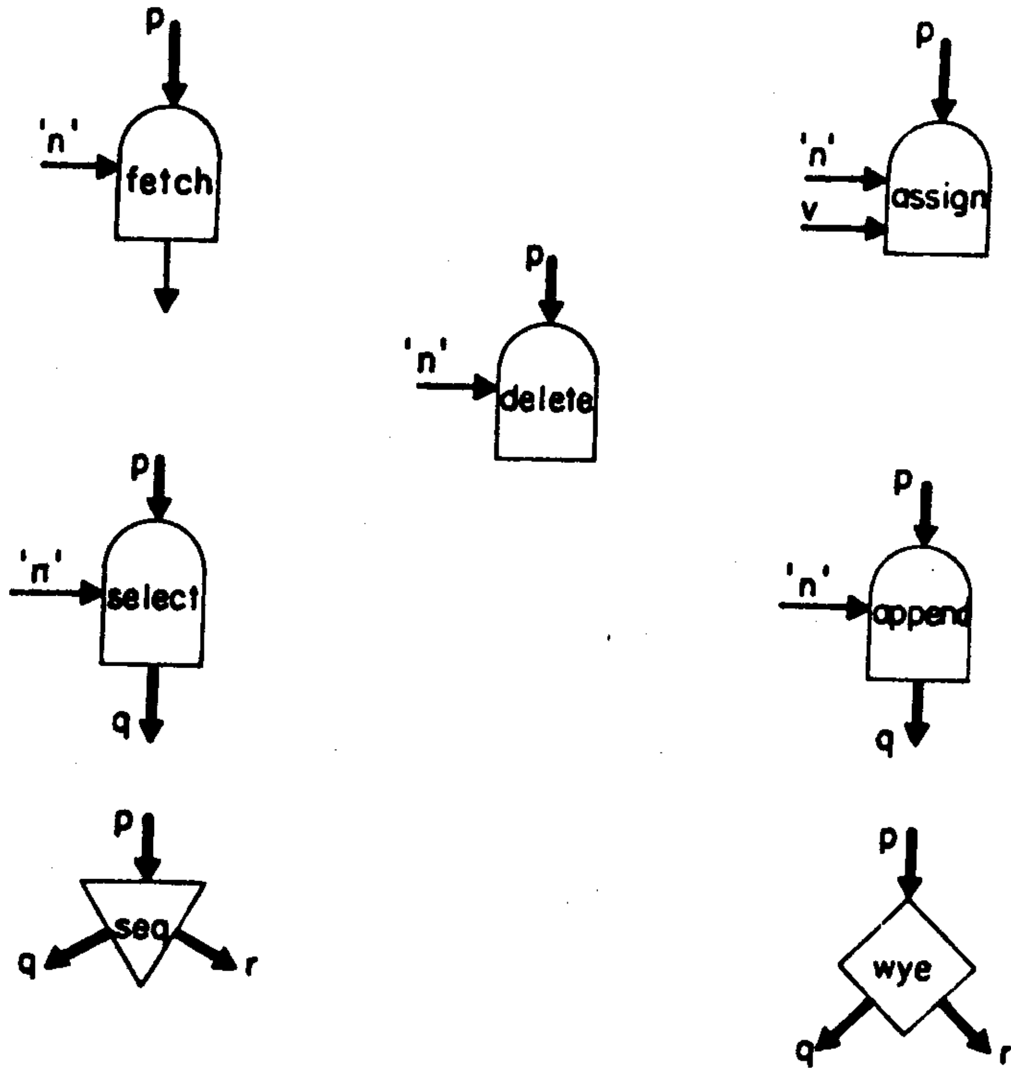


Fig. 3. Node types for extended program graphs.

(which must be a leaf) identified by branch name n within the unique information structure associated with pointer p. If the branch name is a computed integer, fetch models the subscripting operation for accessing elements of an array.

The assign program graph node takes as inputs a pointer p, a branch name n, and a data value v, and associates the value with the environment leaf selected by point p and branch name n. Creation of a leaf is implied if it did not already exist. However, assign is undefined if the selected component exists but is not a leaf.

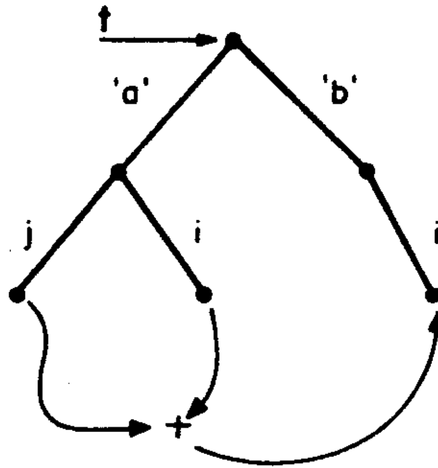
The select and append nodes have pointer and branch name inputs p and n, and produce a pointer q associated with the information structure indentified by n as a component of p. The creation of a unique pointer q is implied if it did not already exist. If p and n specify a leaf, select and append are identified. The select node is used where there is no alteration of the component information structure through use of pointer q. The append node is used where alteration takes place.

The delete node has pointer and branch name inputs, and erases the sp specified branch. All subtrees thereby separated from the environment tree cease to exist. Subtrees that were also components of other structures remain in the environment.

The wye node provides the function of a branch node for pointer values. The seq node is used to control the sequencing of operations on information structures. The meaning of wye and seq nodes is explained further in the following paragraph. Fig. 4 illustrates a simple computation described by an extended program graph in Fig. 5. The notation

t • b • i

means the environment part found by qualifying the structure specified by pointer t successively by branch names b and the value of i.



$$t \cdot 'a' \cdot j + t \cdot 'a' \cdot i \rightarrow t \cdot 'b' \cdot i$$

Fig. 4. A simple computation on a structure.

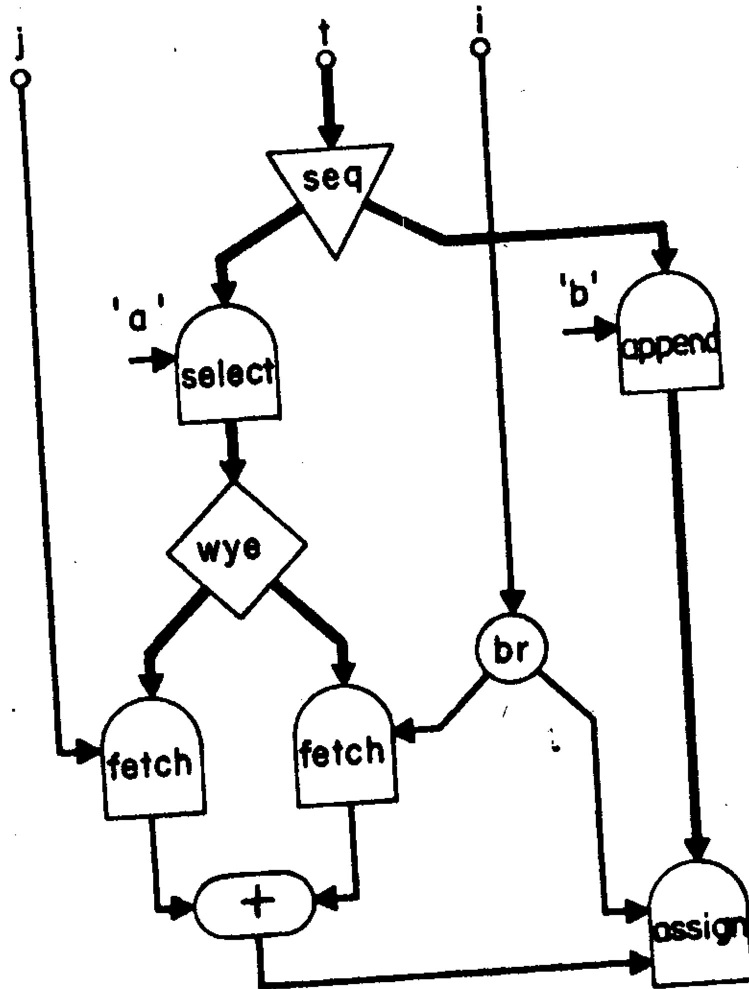


Fig. 5. An extended program graph.

Determinism

When program graphs involve operations on information structures, constraints must be imposed to ensure that the computation defined is deterministic. Specifically, we must make sure that one part of a computation is not modifying an information structure while a second part, enabled to proceed concurrently with the first, is accessing the same information structure. Yet in the interest of exploiting parallelism, we wish to permit simultaneous accesses to information structures whenever there is no danger of ambiguity.

Authority to make reference to an information structure is conveyed by the associated pointer value. Therefore we associate two degrees of access priviledge with pointer values -- read capability and write capability. A pointer value carrying read capability conveys just the privilege of access to the associated information structure through select and fetch nodes. A pointer value carrying write capability conveys also the privilege to modify the information structure via append, assign, and delete nodes.

If a pointer is required for two parts of a computation, the two parts may be run concurrently just if each part requires only read access to the corresponding information structure. If a pointer is required by one part of a computation that needs write capability for an information structure then any other part of the computation requiring use of the same pointer must either preceed or follow the one. In the example (Fig. 5) the seq and wye nodes transmit, via a pointer value, access capability for the information structure to three parts of the computation. The wye enables

simultaneous operation; the seq forces the parts to operate in sequence.

To establish precise meanings for the extended program graph nodes, links that convey pointer values must transmit signals both to and from the inferior node. The forward or start signal carries the pointer value with either read or write capability. This signal indicates that the pointer may be employed to access the associated information structure. The reverse or done signal means that the computation enabled by the inferior node has completed all access to the information structure. Now we may complete the description of the extended program graph nodes by specifying when transmission of start and done signals occurs (Refer to Fig. 3).

fetch — The done signal is returned on link p as soon as the start signal is received and value w has been read from the information structure.

assign — The done signal is returned on link p as soon as the start signal is received and value w has been used to update the information structure. Write capability is required.

select / append — Pointer value q is transmitted when a pointer value arrives through link p. A done signal is returned via link p when a done signal is received from link q. An append node requires and transmits write capability; a select node transmits only read capability.

delete — The done signal is returned as soon as the deleted component has been disconnected from the structure. Write capability is required.

In addition each of the above node types must delay its operation until a value is supplied to the branch name input if the branch name is a computed value.

seq — A pointer received through link p is transmitted through link q. When a done signal is received from link q, the pointer is transmitted through link r. When a done signal is received from r, a done is returned through link p. Write capability is transmitted via links q and r just if write capability is supplied through link p.

wye — A pointer received through link p is transmitted through both links q and r. Just when done signals have been received from links q and r, a done signal is returned through link p. Write capability is transmitted just if it is supplied.

Proper use of the extended program graph nodes will yield unambiguous descriptions of computations. However, if the interconnection of nodes satisfies a few simple rules, determinism may be guaranteed.

Each select node of program graph distributes a pointer value with read capability to a collection of fetch and select nodes through a tree

of wye nodes. This permits concurrent read access to the components of an information structure. No additional constraints are necessary.

Each append node distributes a pointer value with write capability through a tree of seq and wye nodes to a collection C of nodes. The collection contains two classes of nodes -- those requiring only read capability and those requiring write capability:

read class: fetch, select

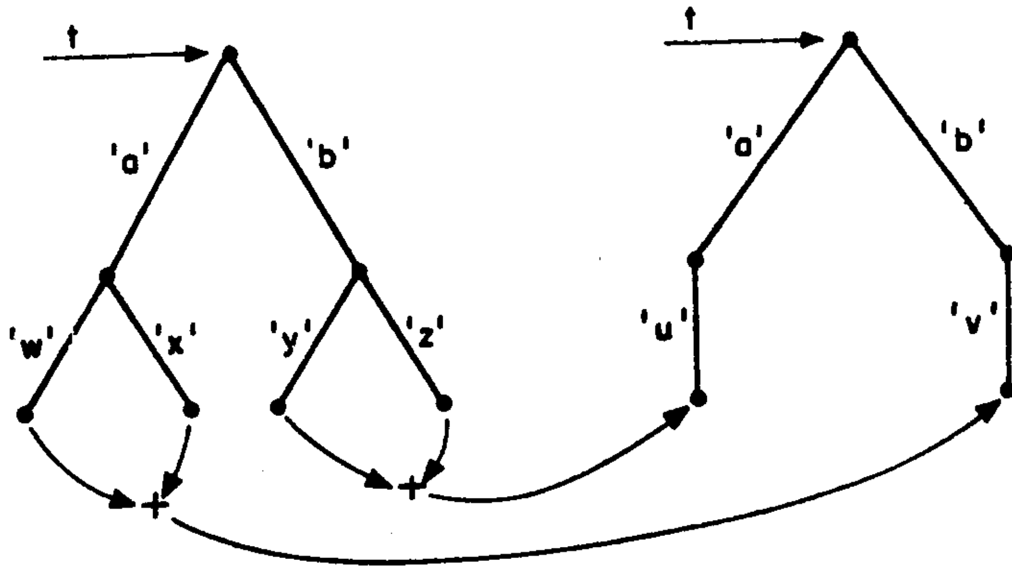
write class: assign, append, delete

To ensure determinacy we constrain the structure of the tree of seq and wye nodes so that simultaneous operation of a write class node and another node in the collection C on the same component of the information structure is not possible. The constraint is:

For a pair of nodes (x,y) in C as specified below, the paths to x and y in the tree of seq and wye nodes must diverge at a seq node.

1) A write class node with computed branch name and any other node in C.
2) A write class node with fixed branch name and any node of C with a computed branch name.
3) A write class node with fixed branch name and any node of C with the same fixed branch name.

A more elaborate computation on an information structure (Fig. 6) is described by a program graph in Fig. 7. Note that the above rules are satisfied.



$$t \cdot \text{'b'} \cdot \text{'y'} \quad + \quad t \cdot \text{'b'} \cdot \text{'z'} \rightarrow t \cdot \text{'a'} \cdot \text{'u'}$$

$$t \cdot \text{'a'} \cdot \text{'w'} \quad + \quad t \cdot \text{'a'} \cdot \text{'x'} \rightarrow t \cdot \text{'b'} \cdot \text{'v'}$$

Fig. 6. Second example of a computation on a structure.

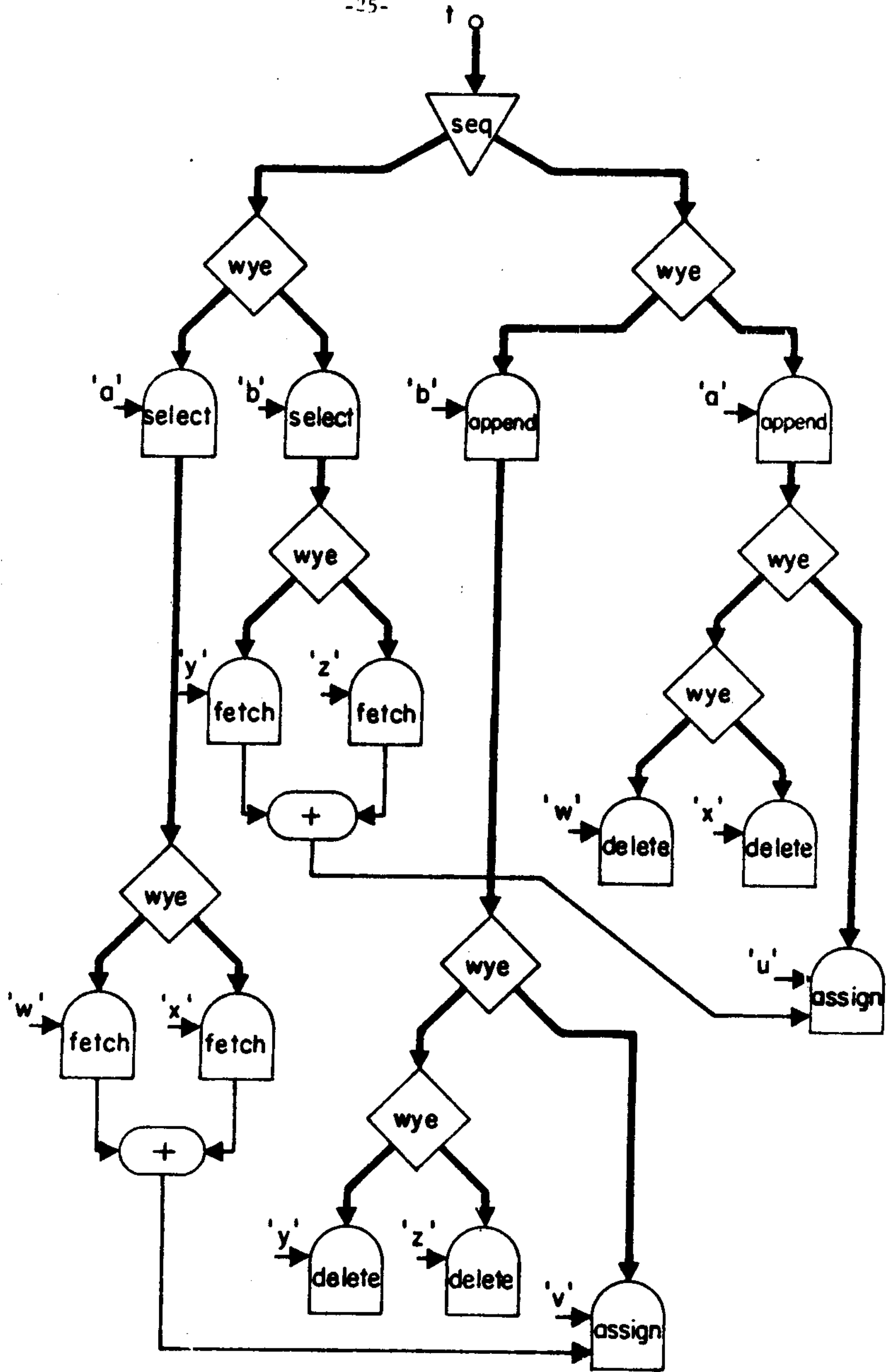


Fig. 7. Program graph fpr the computat;on of Fig. 6.

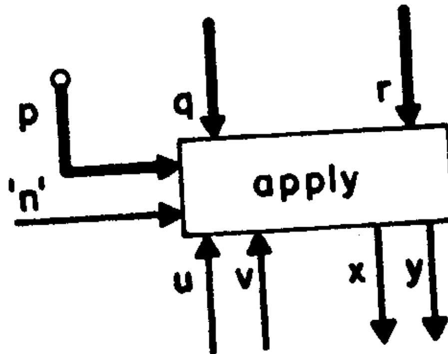
Nesting of Program Graphs

The construction of complex procedures from simpler component procedures is made possible in program graphs by the apply node illustrated in Fig. 8. We suppose that each procedure is described by a program graph and is represented as an information structure in the environment. The pointer and branch name inputs on the left side of the apply node identify the procedure as component n of information structure p. The data interface with the procedure consists of four kinds of items:

1) data input values for use in the sub-computation performed by the applied procedure.
2) data output values produced by the subcomputation.
3) pointer values conveying read capability for information structures.
4) pointer values conveying write capability for information structures.

The data inputs and outputs of an apply node are connected just as in the case of an elementary program graph operator. The pointer links to an apply node are treated just as for select and append nodes with computed branch names. The interface links to an apply operator are distinguished by argument names that correspond to the external links of the applied procedure.

Fig. 9 shows another computation involving information structures. Structure c with component procedures f and g, and data structure d are supplied to procedure t as arguments. Procedures f and g are applied to



argnment names

u,v data input values

x,y data output values

q,r pointers

Fig. 8. The *apply* program graph node.

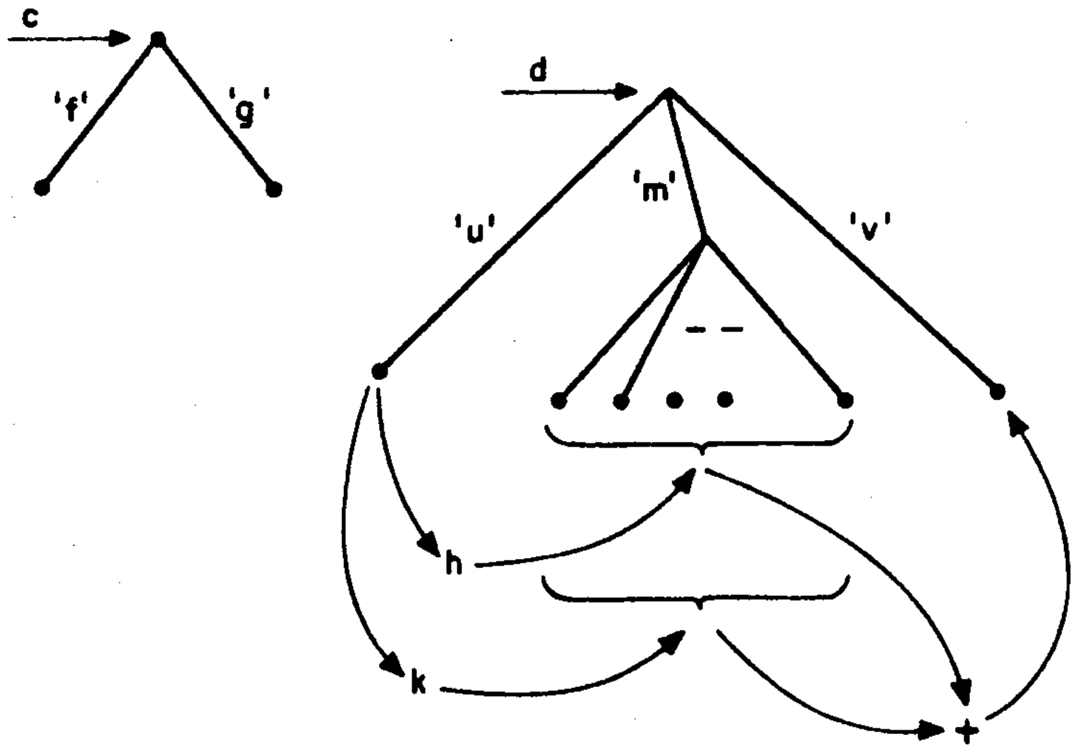


Fig. 9. Schematic of computation by procedure t(c, d).

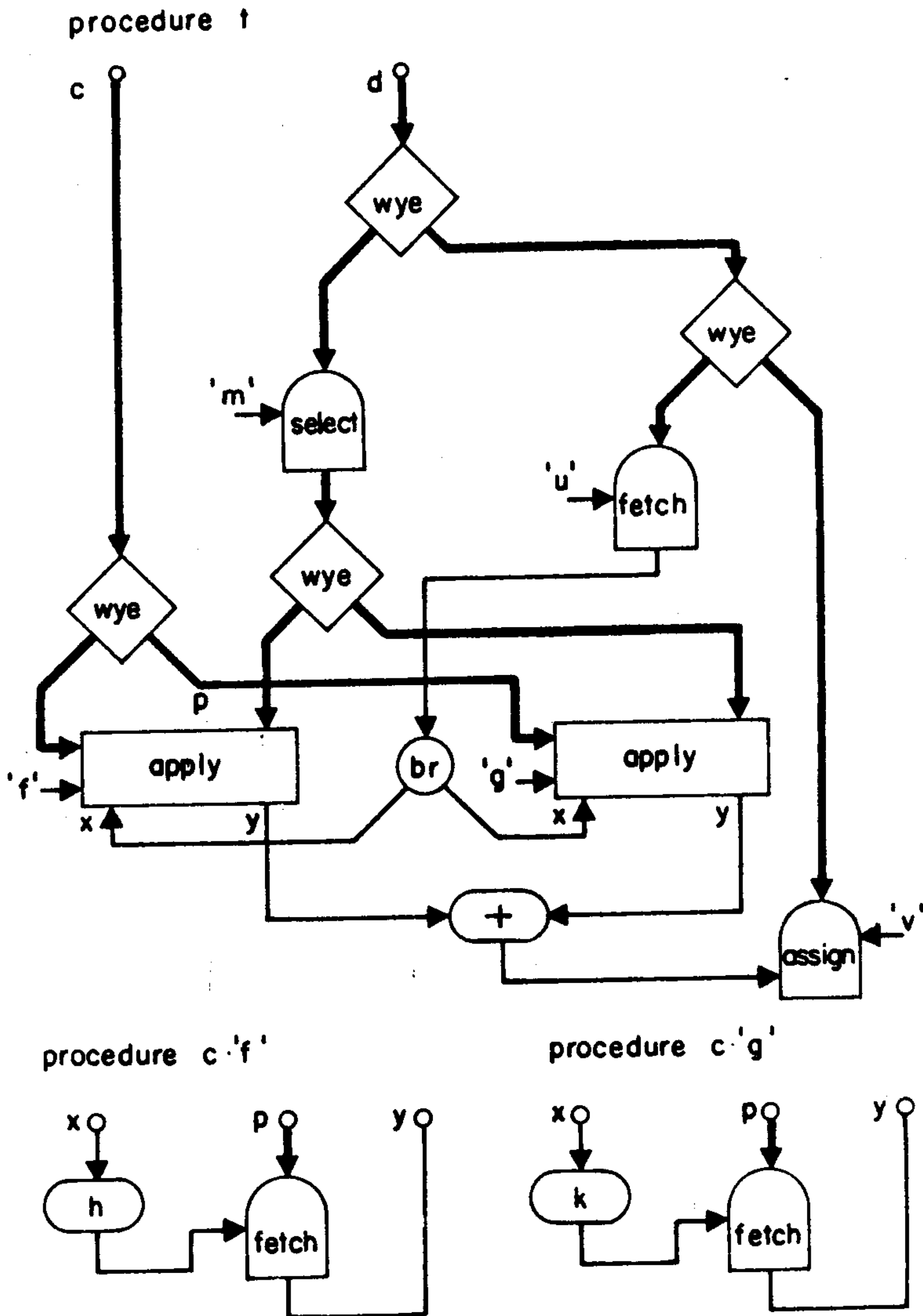


Fig. 10. Nest of program graphs for the computation of Fig. 9.

select two components of structure m under d which are added and assigned as component v of structure d. A set of program graphs for procedures t, f, and g is given in Fig. 10. Note that the representation of procedure t is entirely independent of the point of attachment to the environment, either of itself or structures c and d.

For the subsequent discussion of a machine organization, we need a representation of the apply operator in terms of simpler parts. This is shown in Fig. 11. We use the argument names as branch names of an argument information structure. The execution of an apply operator has four phases:

1) Formation of the argument information structure from pointer and input data values.
2) Transmission of the argument information structure to the applied procedure.
3) Return of the argument structure upon completion of the sub-computation.
4) Forwarding of output values to successor nodes of the apply, and release of pointers by done signals.

Four new node types have been introduced (Fig. 12).

bind/attach - When pointer p and q have been received, information structure p is made component n of structure q and a done signal is returned through link q. Then when pointer value r arrives, component n of structure r is deleted and done signals are returned through links p and r. A bind node requires write capability for pointer p. Both node types require write capability for pointers q and r.

execute - Pointer p and branch name n identify a procedure which is applied to argument structure q. Read capability is required for p and write capability is required for q. Done signals are returned through links p and q when execution of the sub-computation is complete.

create - The output of a create node is a unique pointer that is transmitted to its inferior node. When a done signal is returned the corresponding information structure is deleted.

To correspond with this explicit form of the apply operator, a procedure declaration must include the nodes necessary to access and append to the argument information structure as in Fig. 13. The place of the argument information structure in the environment will be brought out later.

A Machine Organization

A computer organization that realizes the principles expressed in the first sections of this paper is in an early stage of evolution at M.I.T. Our work is related to the interesting proposal of Seeber and Lindquist [17], and is in part based on the research of Patil [15], Van Horn [20], Rodriguez [16], and Luconi [13]. The machine has several unusual characteristics.

1) The machine representation of an algorithm is very nearly a transliteration of its program graph as developed above. Hence parallelism is implemented at the finest level of detail.

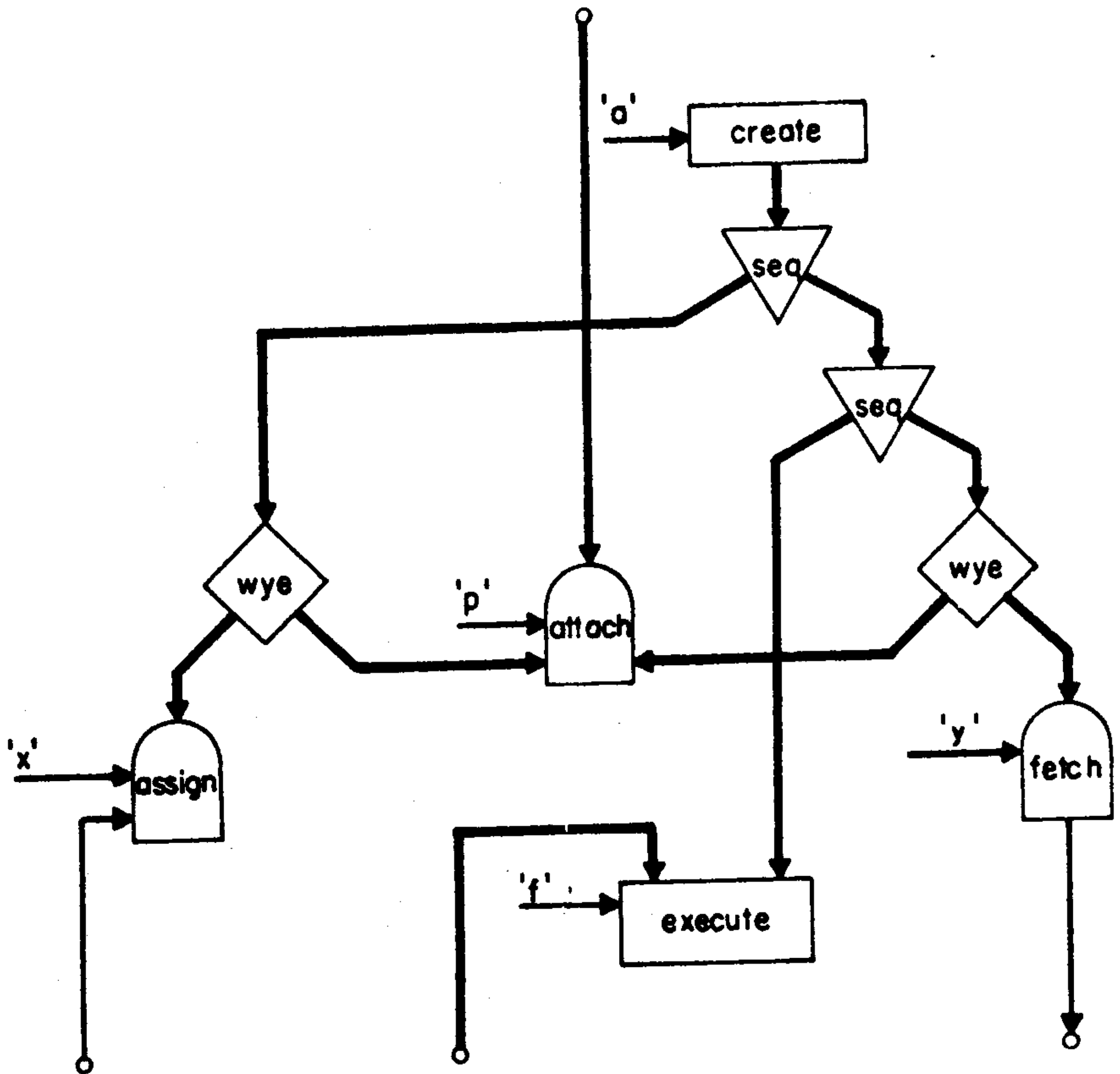


Fig. 11. Elaboration of the apply node.

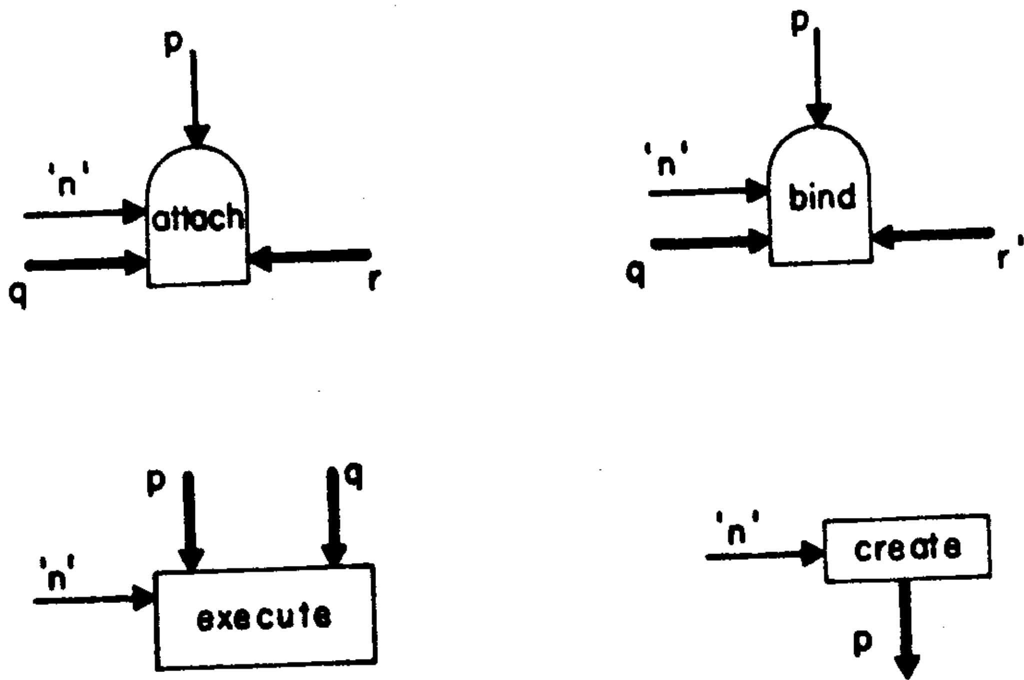


Fig. 12. Program graph nodes for elaboration of the apply node.

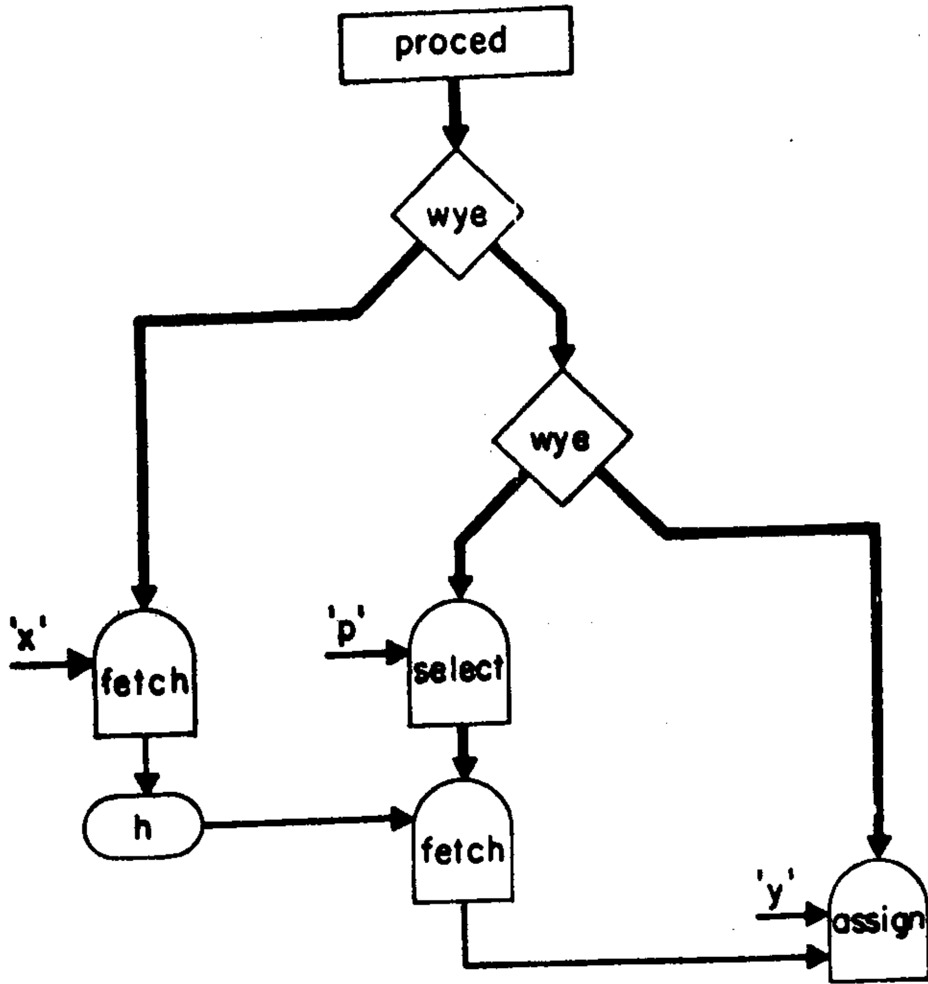


Fig. 13. Elaboration of procedure f.

2) The memory organization is associative at all levels of physical media, using the same addresses throughout, thus realizing a completely location-independent form of reference.

3) The sharing of hardware units among concurrent computations is possible at a fine level, permitting greater hardware utilization.

4) Use of tree structures as the basic form of information structure permits a degree of information sharing not currently practical.

The machine has two parts that can be best described separately, although they are not necessarily separated physically. The parts are the <u>memory system</u> and the <u>processing hardware</u>.

<u>Memory System</u>

The memory system is a collection of memory cells, each cell holding one item. Each item corresponds to a leaf or to an information structure of the environment, and is an association of a <u>value</u> and a <u>type</u> code with a unique <u>key</u> consisting of a <u>pointer</u> code and a <u>name byte</u>:

<pointer> : <name byte> | <value> : <type>

Pointers might be represented by 32-bit strings, name bytes by eight bits, and values by 40-bit strings. So that heavily splintered nodes, or many-character branch names may be used, branch names having long binary representations are transformed in machine code to tree structures having several levels instead of one (See Fig. 14). The basic value types are:

<u>dta</u> data values
<u>ptr</u> pointer values with read capability
<u>ptw</u> pointer values with write capability
<u>ins</u> instructions

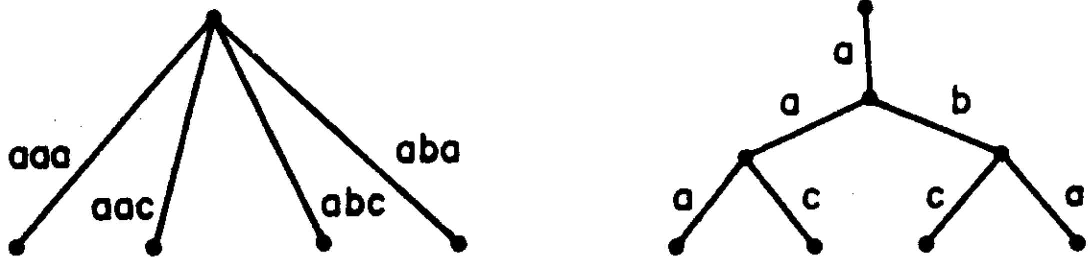
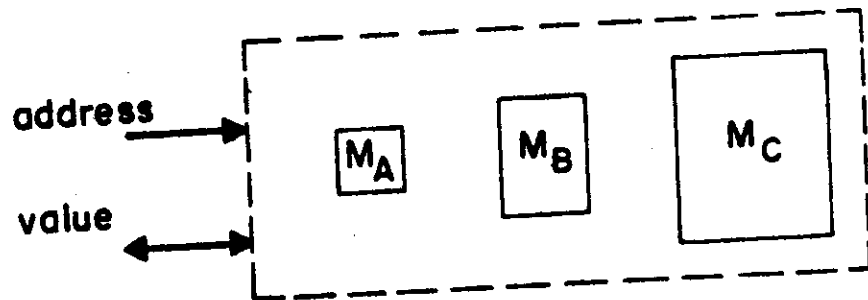


Fig. 14. Treatment of long branch names.

The memory system has, say, three levels, M_A, M_B, M_C arranged as in Fig. 15 where representative access times and capacities are suggested. In each level the addressing is associtive using a key consisting of a pointer and a name byte. The memory read operation presents a given key to levels M_A, M_B and M_C in succession until the corresponding item is found, and is a direct implementation of the fetch program graph node. The memory write operation uses a given key, value and type to update or create an item in memory level M_A, and implements the assign node.

The allocation of words among levels M_A, M_B and M_C is accomplished by building a timer into each memory cell using appropriate time scales for each level. Each time an item is referenced by any ongoing computation, the timer is reset. If the timer runs out, the item is automatically written into the next lower memory level, if necessary, and is deleted from the higher level. Items are brought into higher levels of memory on demand, the item being also entered in intermediate levels.

Each level of the memory hierarchy must be designed for an enormous degree of concurrency in comparison with current practice. It is expected that balance in system usage can be obtained through constraints imposed on the initiation of computations by users, and adjustments of item lifetimes in the different levels of the memory hierarchy (See Denning [5], [6]).



level	access time	capacity
A	10^{-7} sec.	10^5 bits
B	10^{-4} sec	10^8 bits
C	10^{-1} sec	10^{11} bits

Fig. 15. Arrangement of memory system.

The Processing Hardware

The organization of the processing hardware is intended to permit extensive sharing of multiple specialized cells by many computations to ensure statistically high utilization. It is envisioned that there be tens to hundreds of units of each cell class, operating independently and asynchronously using a "service on demand" principle of control. This implies the use of "arbiters" to sequence simultaneous requests for service from any unit.

The associative memory cells constitute the high level memory M_A. The read cells handle requests for items that must be retrieved from lower memory levels. The write cells handle transmission of new items and update requests to storage levels M_B and M_C.

The operator cells perform the basic primitive operations for the data types of the system. Examples are floating point arithmetic for real number representations, integer arithmetic, character string and bit string manipulation. We assume that each operator cell takes two operands (items) as inputs and produces one result item as output. Each class of operator cell is included in quantity suitable for the anticipated (or experienced) demand for its services.

The process cells are analogous to processing units in contemporary computers. Each instruction performed is decoded and interpreted by some process cell through the services of the operator and memory cells. The operation of a process cell will be described shortly.

The Instruction Code

A program for the machine is a collection of procedures. Each procedure in a machine language is a representation of a program graph by a collection of instructions in one-to-one correspondence with nodes of the program graph. In terms of the environment, a procedure is an information structure in which each instruction is a leaf component of the root. Since instructions are in correspondence with program graph nodes, we will apply the relations successor/predecessor and superior/inferior to instructions just as we have with nodes. A few instructions are defined in Fig. 16 in terms of program graph nodes. Each instruction has an operation code field that identifies that kind of node represented, and three name type fields. The latter fields identify the successor, predecessor, superior and inferior instructions within the procedure, or contain fixed branch names. The attach and bind nodes require two instructions as shown.

An encoding of procedure t(c, d) in Fig. 10 is given in Fig. 17. The create instructions are made inferior to wye instructions that cause their execution upon activation of the procedure by arrival of an argument pointer.

Observe that the machine code form of a procedure is pure, that is, no alteration of a procedure information structure will occur in the course of its application. Thus arbitrarily many instances of application and procedure may proceed concurrently.

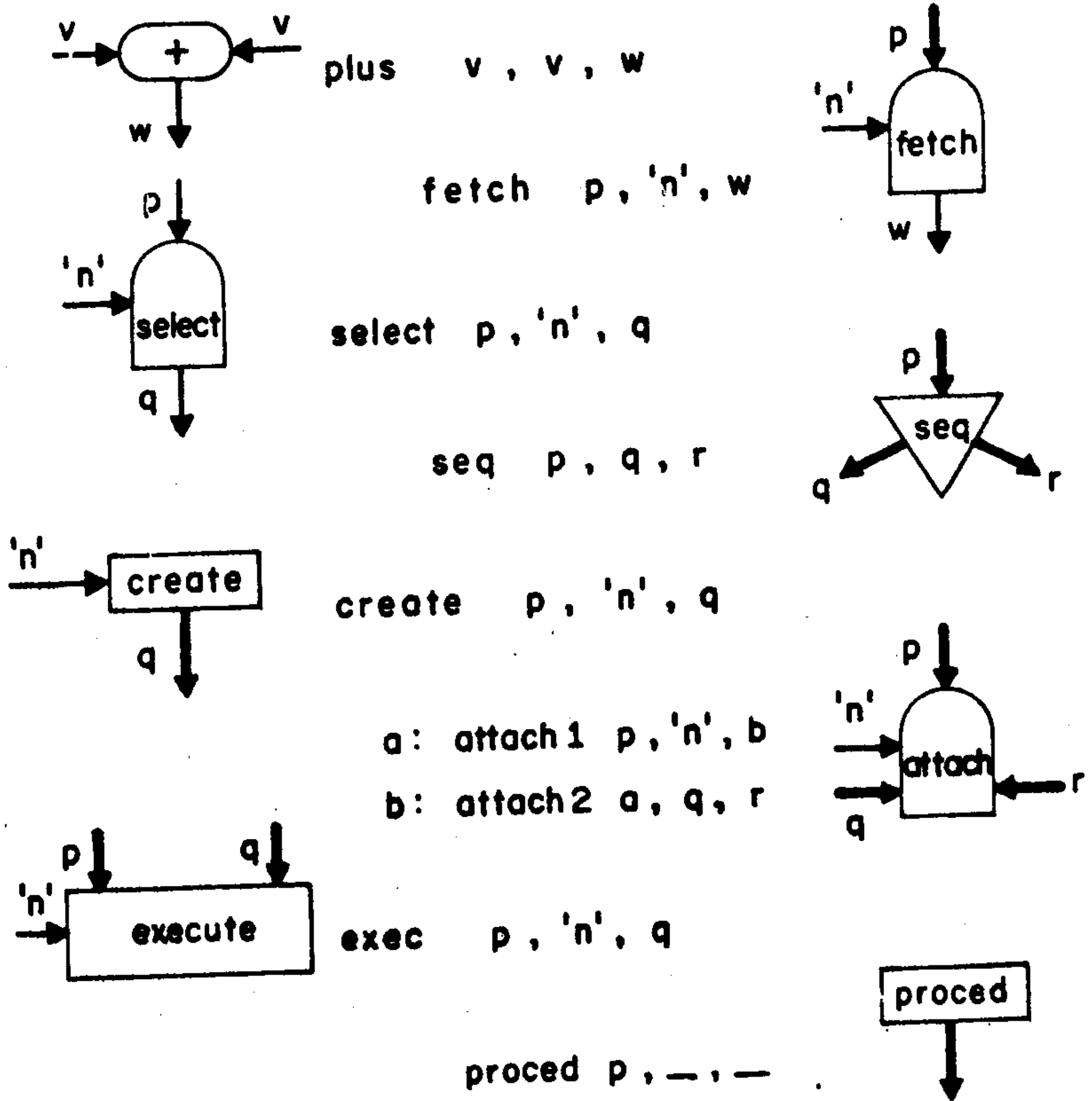


Fig. 16. Representative machine language instructions.

```
0: proced 1
1: wye 0, 2, 3
2: wye 1, 13, 23
3: wye 1, 4, 6
4: select 3, 'c', 5
5: wye 4, 20, 30
6: append 3, 'd', 7
7: wye 6, 8, 9
8: wye 7, 11, 34
9: fetch 7, 'u', 10
10: branch 9, 16, 26
11: select 8, 'm' 12
12: wye 11, 17, 27
13: create 2, 'a', 14
14: seq 13, 15, 19
15: wye 14, 16, 18
16: assign 15, 'x', 10
17: attch1 12, 'p', 18
18: attch2 17, 15, 21
19: seq 14, 20, 21
20: exec 5, 'f', 19
21: wye 19, 18, 22
22: fetch 21, 'y', 33
23: create 2, 'b', 24
24: seq 23, 25, 29
25: wye 24, 26, 28
26: assign 25, 'x', 10
27: attch1 12, 'p', 28
28: attch2 27, 25, 31
29: seq 24, 30, 31
30: exec 5, 'g', 29
31: wye 29, 28, 32
32: fetch 31, 'y', 33
33: plus 22, 32, 34
34: assign 8, 'v', 33
```

Fig. 17. Machine language encoding of procedure t(c, d).

Process Cell Operation

During execution of a procedure, the value produced by an instruction named n is made the leaf named n of a local data structure associated with the activation of the procedure. The local data structure is referenced by the local pointer for the activation.

Cells in memory level M_A may contain items known as process states that are associated with instruction instances enabled by completion of their predecessor and superior instructions. A process state item has the format

<local pointer> : <name byte> | <procedure pointer> : <type>

where the procedure pointer designates the procedure information structure for the activation, and <type> is one of

- ena enabled status - available for execution.
- en2 waiting status 1 - pending completion of one additional predecessor or superior/inferior instruction.
- enl waiting status 2 - pending completion of two additional predecessor or superior/inferior instructions.

All process state items of type ena compete for service from process cells.

We explain first the action of a process cell for the instruction

p : c | plus, a, b, c, : ins

executed in an activation of procedure p having local pointer q. Initially, the memory system will contain the following relevant items aside from the instruction: the enabled process state

q : c | p : ena

and the data values computed by predecessor instructions

q : a | u : dta
q : b | v : dta

step 1) The process cell obtains the procedure pointer p from item q • c and with name byte c retrieves item p • c from the memory system.

step 2) From instruction p • c the process cell determines that items q • a and q • b are required and initiates their retrieval from the memory system.

step 3) Once values u and v are in hand, they and the operation code plus define the transaction requested from an operator cell.

step 4) When the operator cell returns a result w = u+v the process cell writes it into item q • c and changes its type to dta.

q : c | w : dta

step 5) The process cell requests item q • d from memory level M_A; if it exists step 7 is next, otherwise step 6 follows.

step 6) The process cell retrieves instruction item p • d and determines from its operation code whether one, two or three instruction completions are required to enable it. Item q • d is created as

q : d | p : <type>

with <type> set to ena, en1, or en2 accordingly. Interpretation of instruction p • c is then complete.

step 7) Item q • d will be

q : d | p : <type>

where <type> is en1 or en2. It is changed to ena or en1 respectively to complete interpretation of instruction p • c.

Other instruction types are treated similarly by process cells. Nodes having pointer links to inferior nodes require several actions by process cells that may be arbitrarily separated in time. A <u>seq</u> instruction requires three seprate actions: 1) to enable the first inferior when a <u>start</u> signal is received from the superior instruction; 2) to enable the second inferior after a <u>done</u> signal is received from the first, and 3) to enable the superior upon receipt of a <u>done</u> from the second inferior.

The <u>exec</u> and <u>proced</u> instructions deserve specific explanation. Let the initial situation be thus: Item p • c is the <u>exec</u> instruction

p : c | <u>exec</u>, a, f, b : <u>ins</u>

It is supplied a pointer t

q : a | t : <u>ptr</u>

for gaining access to procedure f via procedure pointer r

t : f | r : <u>ptr</u>

and an argument pointer u.

q : b | u : <u>ptr</u>

The initial instruction of procedure r is

r : C | <u>proced</u>, d, -, -: <u>ins</u>

The process cell interpreting the process state

q : c | p : <u>ena</u>

retrieves the <u>exec</u> instruction p • c. Then it creates a local pointer s for the new activation, and saves the procedure pointer p, the local pointer q and the branch name c of the <u>exec</u> instruction by forming the items

```
s : - |        p          : pro
s : - |        q          : loc
s : - |  -,  -, -,  c     · ret
```

The new local data structure is appended as the c component of local structure q

```
q : c |        s          : ptr
```

and an enabled process state

```
s : d |        r          : ena
```

is formed to initiate the new activation. When a done signal returns to the proced instruction, the saved items are retrieved, and done signals are sent to the superiors of the exec instruction.

Concluding remarks

It is clear that a structure such as the one outlined would be difficult to construct as a centrally clocked system without unnecessary loss of speed. Fortunately the theory of asynchronous structures has seen considerable progress. Recent work of Luconi [12, 13] provides a theoretical basis by which one can envision asynchronous structures synthesized entirely with time-independent logic. His theory shows how one can implement shared hardware structures by a straightforward design procedure.

The embrionic status of this research prohibits discussion of additional issues that arise in computer system design - for example, the implementation of program graphs containing decisions and iteration, the representation in program graphs of application of a procedure in parallel to the components of an information structure, and questions of information integrity and security. These topics are under investigation.

References

1. Aschenbrenner, R. A., M. J. Flynn, and G. A. Robinson. Intrinsic Multiprocessing. AFIPS Conference Proceedings Vol. 30 (Thompson Books, Washington, D. C., 1967) pp 81-86.

2. Burge, W. H. A Reprogramming Machine. Comm. of the ACM, Vol. 9, No. 2 (February, 1966) pp 60-66.

3. Corbato, F. J. and V. A. Vyssotsky. Introduction and Overview of the Multics System. AFIPS Conference Proceedings, Vol. 27, Part I (Spartan Books, Washington, D. C. 1965) pp 185-196.

4. Daley, R. C. and J. B. Dennis. Virtual Memory, Processes, and Sharing in Multics. Comm. of the ACM, Vol. 11, No. 5 (May, 1966) pp 306-312.

5. Denning, P. J. The Working Set Model for Program Behavior. Comm. of the ACM, Vol. 11, No. 5 (May, 1966) pp 323-333.

6. Denning, P. J. Resource Allocation in Multiprocess Computer Systems Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, (June 1968).

7. Dennis, J. B. Segmentation and the Design of Multiprogrammed Computer Systems. Journal of the ACM, Vol. 12, No. 4 (October, 1965) pp 589-602.

8. Dennis, J. B. and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. Comm. of the ACM, Vol. 9, No. 3 (March, 1966) pp 143-155.

9. Iliffe, J. K. The use of the Genie System in Numerical Calculation. Annual Review in Automatic Programming, Vol. 2, R. Goodman, Ed. (Pergamon Press, London, 1961) pp 1-28.

10. Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-Level Storage System. IEEE Trans. Vol. EC-11, No. 2 (April 1962) pp 223-235.

11. Landin, P. J. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I, Comm. of the ACM, Vol. 8, No. 2 (February 1965) pp 80-100; Part II, Comm. of the ACM, Vol. 8, No. 3 (March 1965) pp 158-165.

12. Luconi, F. L. Completely Functional Asynchronous Computational Structures. IEEE Conference Record of the Eighth Annual Symposium on Switching and Automata Theory. (October 1967) pp 62-70.

13. Luconi, F. L. Asynchronous Computational Structures. Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering. (February 1968).

14. Martin, D. E. and G. Estrin. Models of Computational Systems--Cyclic to Acyclic Graph Transformations. IEEE Trans. Vol. EC-16, No. 1 (February 1967) pp 70-79.

15. Patil, S. An Abstract Parallel Processing System. S. M. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, (June 1967).

16. Rodriguez-Besos, J. E. A Graph Model for Parallel Computations. Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, (September 1967).

17. Seeber, R. R. and A. B. Lindquist. Associative Logic for Highly Parallel Systems. AFIPS Conference Proceedings, Vol. 24, (Spartan Books, Baltimore, 1963) pp 489-493.

18. Slotnick, D. L., W. C. Borck, R. C. McReynolds. The Solomon Computer. AFIPS Conference Proceedings, Vol. 22, (Spartan Books, Baltimore, 1962) pp 97-107.

19. Sutherland, W. R. On-line Graphical Specification of Computer Procedures, Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering (December 1965).

20. Van Horn, E. C. Computer Design for Asynchronously Reproducible Multiprocessing. Project MAC, Massachusetts Institute of Technology, (November 1966).