UNIVERSITA' DEGLI STUDI DI PISA

ISTITUTO DI SCIENZE DELL INFORMAZIONE

A multiprocessor interpreter for a high-level machine language

L. Cardelli - D. Citterico - M. Greco - G. Prini - M. Vanneschi

NOTE SCIENTIFICHE

S - 77 - 10

MARZO 1977

A MULTIPROCESSOR INTERPRETER FOR A HIGH-LEVEL MACHINE LANGUAGE

Luca Cardelli , student

Dario Citterico , student

Maurizio Greco , student

Gianfranco Prini

Marco Vanneschi

Istituto di Scienze dell'Informazione
Università di Pisa

1. INTRODUCTION

Several recent programming languages, such as ALGOLW, SIMULA67, ALGOL68, PASCAL, EL1 and CLU, have been designed to allow data <u>extensibility</u>, that is the ability of <u>declaring</u> (vs. explicitly <u>programming</u>) data types. Since most computers are not very well suited to support extensible languages [9], designers are almost inevitably induced to introduce several restrictions to the treatment of data types, most of them being suggested by the need of minimizing run-time type-checking. It is a common opinion that the problem of designing a truly general and efficient extensible language would be greatly simplified provided that the existence of a fast and sophisticated memory manager is assumed.

In [2] a machine, called <u>SMOM</u>, is proposed which allows data types to be specified already at the machine language level, in a way similar to [7,10]. SMOM may be considered as a quite powerful memory manager which operates under the control of a zero-address, stack-oriented machine language, into which definitional interpreters (in the sense of [11]) for high level programming languages may be easily hand- or cross-compiled.

Here we do not wish to debate the advantages of the machine language of SMOM for the implementation of interpreters and compilers (for a somewhat elaborate discussion see [2]). Rather we wish to point out how such a machine language may be supported by a multiprocessor architecture which significantly speeds up the execution of SMOM programs by concurrently evaluating arguments to functions and by performing most type checking in parallel with other significant computations.

Section 2 contains a short but self contained description of SMOM. In Section 3 the characteristics of parallel processing in SMOM code, including a unique interprocess communication discipline, are presented, leading to the general ideas for a SMOM architecture. In Section 4 a multiprocessor organization is suggested which implements the ideas of Section 3.

2. SMOM: A SHORT DESCRIPTION

2.1 Data types

All data are manipulated through data descriptors (dd's). A dd is a typed pointer, i.e. a pair <T,A>, where T is the type of the described datum, and A is its memory address. Both types and addresses are explicitly manipulated by the hardware only: the user is not enabled to access them. This implies that memory management is completely transparent to the user. Data may be either basic (i.e. with no inner structure) or structured (i.e. arrays and records of dd's). Both basic and structured data types may be either built-in or user-defined.

Data type definitions are supported by the notion of class. A class is a structured data type that records all information relative to a specified data type. It contains a template which defines the inner structure of all data of that type (the template being empty for basic data types), and an unspecified number of locations, called registers, whose main purpose is to record (dd's for) the definitions of the functions associated with that class (see Section 2.3 and 2.4). Thus a class may be seen as a data type having a behaviour, which is specified by its associated functions.

As an example, Fig. 1 shows the declaration of the class of COMPLEX numbers, which are records of two fields, called REAL and IMAG, containing two REAL numbers.

```
(CLASS COMPLEX
       (TEMPLATE (REAL REAL)
                 (IMAG REAL))
       (REGISTER PLUS
                 <code for adding
                  two COMPLEX numbers>)
              :
       (REGISTER FOO
                 <code for FOOing
                  two (three?) COMPLEX
                  numbers>))
```

Fig. 1 - The definition of the class of COMPLEX numbers.

Evaluation of class definitions is explained in Section 2.3.

2.2 Stacks

In order to process a datum, a dd for it must be present in the argument stack (A-stack), whose building blocks are the A-cells. Though the A-cells contain dd's, they are also said to contain data. The A-cell at the top of the A-stack is named A-top. The datum contained into the A-top is the top-datum: its type is the top-type.

Continuation points of programs (see below) are held in the control stack (C-stack), which is made of C-cells. The C-cell at the top of the C-stack is named C-top.

2.3 Microprograms

Two special built-in data types are present in SMOM, called microprograms and programs. Microprograms implement the basic computational functions of SMOM (i.e. they are supposed to be executable by the hardware). Microprograms may be grouped into the following categories.

i) A-managers and C-managers are used for the data control (i.e. for the management of the A-stack) and for the sequence control (i.e. for the management of the C-stack) respectively. The main A-managers and C-managers are GET, PUT, PUSH, POP, SQUEEZE and JUMP, JUMPT, JUMPF, TYPEJUMPT, TYPEJUMPF, CALL, RETURN, respectively: they are described in the Appendix. Their descriptors (i.e. entry points) are stored in the homonymous registers of a special class, called ANY (see below).

ii) Operations are used for standard computations. Each of them has an associated arity (i.e. number of arguments). Execution of an n-ary operation (i.e. an operation whose arity is n) causes n arguments to be popped off the A-stack and a value to be pushed onto the A-stack itself.

SMOM contains several built-in microprograms (A-and C-managers and most microprograms which specify the behaviour of built-in data types fall into this category). Many others (mostly operations) are automatically synthesized when a class definition is evaluated, and dd's for them are stored in the appropriate registers. For instance, referring to the example of Fig. 1, the following actions are generated:

1. A <u>constructor</u> operation for building new COMPLEX numbers out of two REAL numbers is stored in the register COMPLEX of the class ANY.
2. Two <u>selector</u> operations for treating the components (i.e. the real and imaginary parts) of a COMPLEX number are stored in the REAL and IMAG registers of the class COMPLEX. They may operate either in <u>load mode</u> (in which case they are 1-ary and are used to retrieve the components of COMPLEX numbers) or in <u>store mode</u> (in which case they are 2-ary and are used to update the components of COMPLEX numbers: the first argument replaces the appropriate component of the second argument). The default mode is the load mode. The store mode is used when a selector is invoked via the special operation UPDATE (see section 2.4).
3. Several utility operations for reading, writing, editing, etc. COMPLEX numbers are stored in the appropriate registers of the class COMPLEX.

The mechanism which allows microprograms to be retrieved and executed is explained in Section 2.4.

<u>2.4 Programs</u>

Programs are linear sequences of <u>functions</u>, each one being made of an <u>operation code</u> and some (possibly zero) <u>operands</u>. Both operation codes and operands are represented as <u>bytes</u>: thus a program is represented as a linear sequence of bytes. Each byte is interpreted either as an operation code or as an operand according to its position, as it usually happens in a standard byte-oriented computer.

Programs are interpreted by a fetch-decode-execute loop which performs the following operations.

A byte is fetched, according to the address contained in the C-top (fetching always increments the contents of the C-top), and it is interpreted as an operation code. This means that a register having the same internal name as the fetched byte is searched <u>in the special class ANY</u>. If such a register is found, the datum described by it is called as a function according to the following rules.

1. If the datum is a <u>microprogram</u>, it is directly executed by the hardware. It may fetch some bytes of the calling program and use them as operands: this is the case of all the functions described in the Appendix (other than RETURN) and the function UPDATE (which has the name of a selector as its only operand).
2. If the datum is a <u>program</u>, a dd for it (i.e. a pointer to its first instruction) is pushed onto the C-stack and execution goes on with the new program counter. Notice that recursion is naturally exploited, as it is a built-in control structure.
3. Otherwise the datum is pushed onto the A-stack and the standard function APPLY is called (after having pushed the continuation point of the calling program onto the C-stack). By the type driven call machinism (see below), the function APPLY is searched in the class to which the datum itself belongs. Thus each datum may be used as a function, provided that an APPLY function (i.e. an interpreter) is defined for it.

If the class ANY does not contain a register with the same internal name as the fetched byte, such a register is searched <u>in the class of the top-datum</u>. If such a register is found, processing goes on as in the previous case, otherwise an error is generated. The call mechanism just described is named <u>type-driven</u>, since the top-type may "drive" the access to the code of the called function.

A sample program which performs the addition of two COMPLEX numbers in shown in Fig. 2. The reverse polish notation is used.

```
(PROG (GET,2) (REAL)
      (GET,2) (REAL) (PLUS)
      (GET,3) (IMAG)
      (GET,3) (IMAG) (PLUS) (COMPLEX)
      (SQUEEZE,2)
      (RETURN))
```

Fig. 2 - A program for adding two COMPLEX numbers.

Note that, though the program is written in a "pretty format", its memory representation is simply a sequence of bytes, the first byte being the internal name of "GET", the second one being the binary code of "2", etc.

3. SMOM ARCHITECTURE AND PARALLEL PROCESSING

3.1 General discussion

In Section 2, SMOM functions have been supposed to be executed in a strictly sequential order, the order being specified by contiguity and jump operations (program calls and returns are to be considered as jump operations).

However, the features of the SMOM language allow (we could say force) to exploit the inherent parallelism of the algorithms in a natural way: notably, the equivalence of form between functions and data and the encouragement to use recursion as a control structure. As it happens with the high-level languages with such features (e.g. LISP), a SMOM computation consists essentially in a number of function calls, where a function takes the form of an expression. Thus, the tree (i.e. particular partial ordering) instances are heavily present in the computation graph, contrary to what happens in sequence-of-statements based languages (in particular, in conventional assembler languages) where such instances are confined to the arithmetical-logical expressions only. Partial ordering, which allows the most natural and controllable form of parallelism, is further exploited, in the form of trees, through the use of the reverse polish notation. Moreover, notice that the SMOM functions (see for instance A- and C-managers) are more powerful than the conventional assembler instructions; thus, exploiting the parallelism among them leads to much faster and efficient execution and, as we shall see, to an easy modular separation of firmware controlled system tasks.

We intend now to show how the various parts into which the program may be divided can be detected, ordered and executed in parallel. Of course, determinacy and efficiency impose suitable synchronization requirements: these will be defined in a unique way.

In what follows the reader is recommended not to take care of the type-driven call mechanism of SMOM. Also the existence of functions (such as PUT and UPDATE) that perform side-effects is to be ignored by now. These features will be treated later.

Referring to Fig. 2, it is evident that the computation graph of that program is the tree which is shown in Fig. 3.

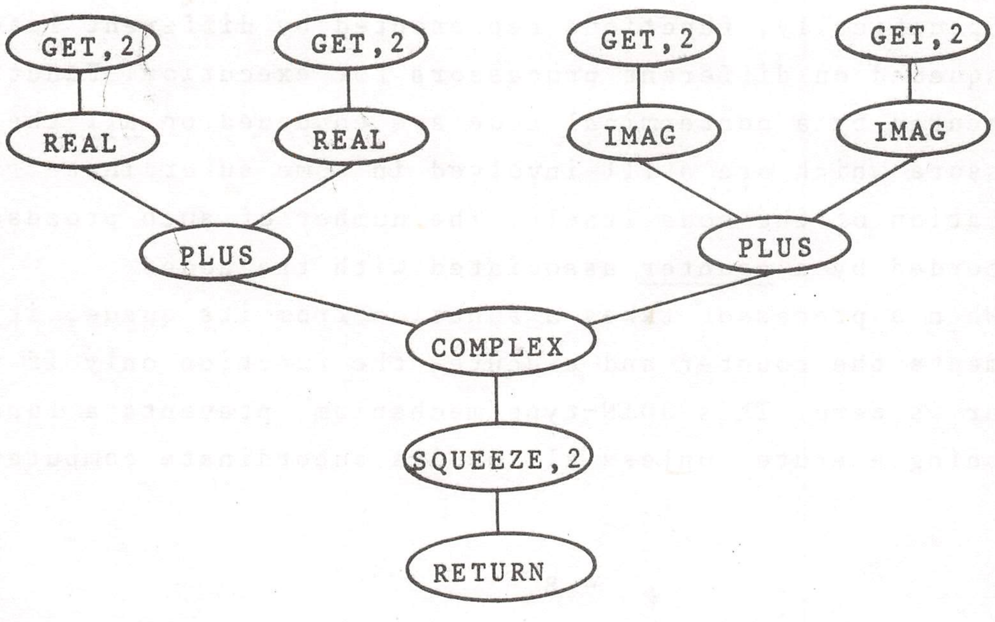


Fig. 3 - The computation graph of the program of Fig. 2.

Each leaf represents a function which starts an independent computation. Each nonterminal node with K sons represents the execution of a function with K subordinate computations.

Computation trees may be reproduced in a multiprocessor environment by means of several synchronization techniques which all are implementations on the Dijkstra's P and V primitives [8] or, equivalently, on the Conway's FORK and JOIN instructions [4]. However, such techniques are efficient chiefly in the classical multitasking environment where tasks are relatively complex in size so that high processor-switching overhead is tolerable [13]. Improvements can be devised when the tasks may be of any size (see, for instance, [6,13,5]), but all these techniques have the same characteristics: every task is dispatched to one processor only as soon as it is ready for execution. This fact prevents from employing other parallelism forms, such as <u>prefetch</u> or <u>look-ahead</u>: we wish to employ such forms indeed, as we are concerned with efficient execution of (high-level) machine code. Moreover, we need a synchronization strategy that agrees with the data control (i.e. A-stack management) strategy. Our solution is characterized by the fact that the fetched functions are dispatched to <u>all</u> the processors which are potential candidates to their executions: function execution will actually controlled by a JOIN-type mechanism <u>and</u> by a "data-driven" control mechanism.

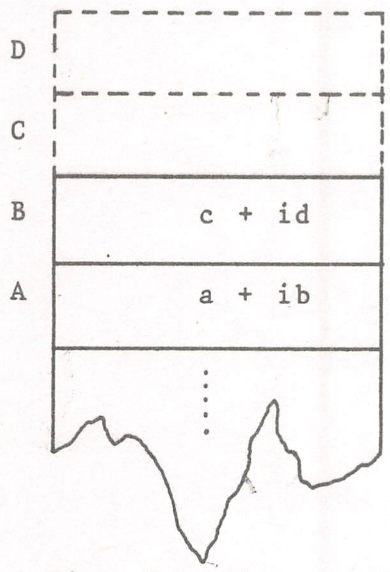
Specifically, functions represented by different leaves are enqueued on different processors for execution. Functions represented by a nonterminal node are enqueued on <u>all</u> the processors which are still involved in some subordinate computation of the node itself. The number of such processors is recorded by a <u>counter</u> associated with the node.

When a processor takes a function from its queue, it decrements the counter and executes the function only if the counter is zero. This JOIN-type mechanism prevents a function from being executed unless all of its subordinate computations

In the following subsections several implementation problems arising with such an architecture are examined in some detail.

3.2 The A-stack

Assume that the CP enqueues the first seven functions of the program of Fig. 2 on the three processors P_1, P_2 and P_3, according to the strategy explained in Section 3.1, as it is shown in Fig. 5.



P_1	P_2	P_3
GET(A→C)	GET(B→D)	GET(A→D)
REAL(C→C)	REAL(D→D)	.
PLUS(C,D→C)	PLUS(C,D→C)	.
:	:	:

Fig. 5 - After prefetching seven functions of the program of Fig. 2. More precisely, the A-cells A and B should contain descriptors for the COMPLEX numbers a+ib and c+id.

Here some A-cells have been labeled in order to single out the A-cells (also called source A-cells) which contain the arguments and the A-cell (also called destination A-cell) which is to contain the result of each enqueued function.

Assume also that the processors execute the prefetched functions according to the timing of Fig. 6.

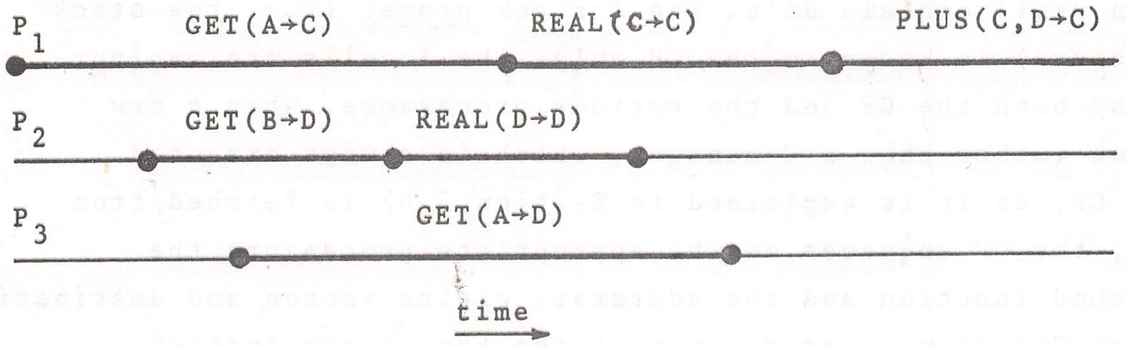


Fig. 6 - A possible timing for the execution of the prefetched functions of Fig. 5.

It is evident that the value deposited by P_2 into the A-cell is overwritten by P_3 before P_1 is enabled to use it for the addition (remember that P_2 cannot perform the addition, since P_1 is still computing the first summand when P_2 completes the computation of the second one).

This example shows that the A-cells cannot be simply overwritten by the processors without possibly generating disastrous side-effects on the A-stack. As mentioned before, semaphores, critical sections and other well-known devices for interprocess communication are not needed to share the A-stack correctly among the various processors.

A much simpler solution is sketched in Fig. 7. The A-stack is made into a stack of pointers to A-cells, which

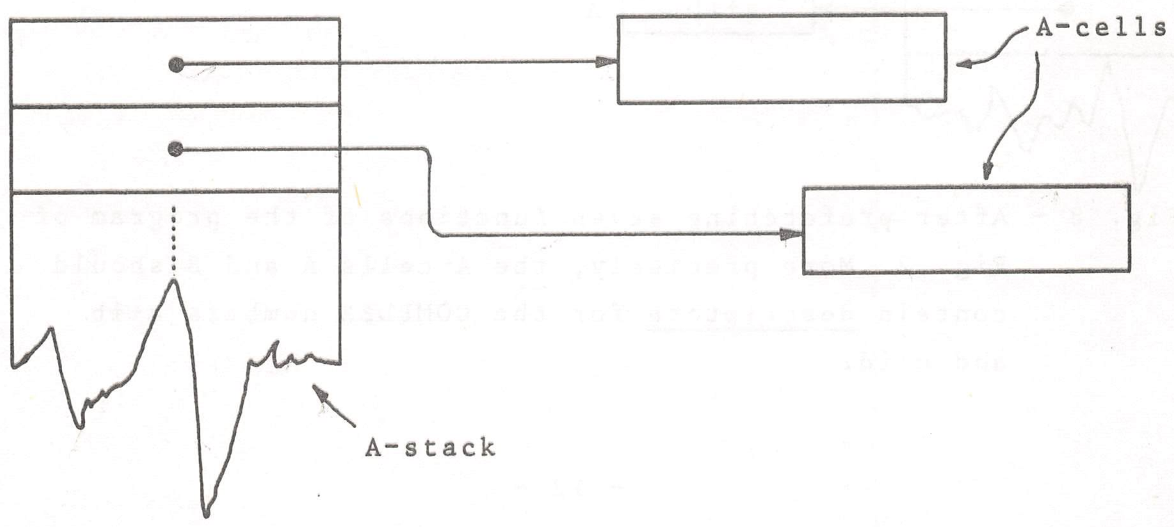


Fig. 7 - The structure of the A-stack.

in turn still contain dd's. The A-stack proper (i.e. the stack of pointers) is known to the CP only. The A-cells are manipulated by both the CP and the various processors. When a new function (other than a C-manager, which is always executed by the CP, as it is explained in Section 3.6) is fetched from memory, the CP enqueues on the appropriate processors the prefetched function and the addresses of its source and destination A-cells. The former are taken near the top of the A-stack proper, the latter is the address of a newly allocated A-cell, which becomes the new A-stop (the A-stack proper is appropriately overwritten). The behaviour of the CP with the program of Fig. 2 is sketched in Fig. 8.

```
     P1                  P2                P3

GET(A→C)            GET(B→E)          GET(A→H)
REAL(C→D)           REAL(E→F)
PLUS(D,F→G)         PLUS(D,F→G)
    .                   .                 .
    .                   .                 .
    .                   .                 .
```

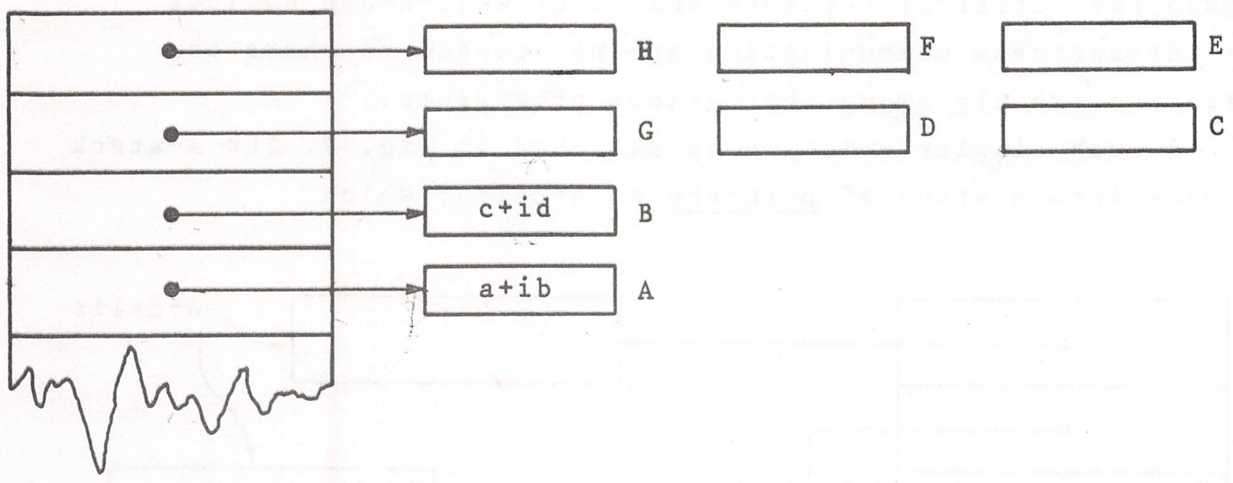


Fig. 8 - After prefetching seven functions of the program of Fig. 2. More precisely, the A-cells A and B should contain descriptors for the COMPLEX numbers a+ib and c+id.

Note that no A-cell is ever overwritten: only pointers contained in the A-stack proper are. However, no information is lost now, since the A-cells containing useful values are still referenced from the processors' queues. When an A-cell becomes unreferenced, it may be reclaimed to free-storage, as it is explained in Section 3.3.

3.3 - A-managers and the A-cell space

In Section 3.2 we have introduced the notions of source and destination A-cells for a function. These concepts are clear enough as far as operations (see Section 2.3) are concerned. But what about the A-managers? Few problems arise with PUSH: it may be considered as a nullary operation (i.e. an operation with zero arguments) which is completely executed by the CP. Also POP and SQUEEZE create few problems: they may be considered as (n+1)-ary operations (i.e. operations with n+1 arguments) which are enqueued and executed according to the strategy explained in Section 3.2. For the three A-managers considered by now and for all the operations the following policy may be devised for the management of the A-cell space. Whenever a function is prefetched, a destination A-cell is allocated, as it has been already explained in Section 3.2. Whenever a function is <u>executed</u>, its source A-cells are deallocated, since they cannot be referenced any more.

As far as GET is concerned, almost the same policy may be adopted: a destination A-cell is allocated by the CP, as it is shown in Fig. 8, but the source A-cell is never deallocated when GET is executed.

Unfortunately, this simple policy does not seem to be applicable to the functions PUT and UPDDATE, unless either the management of the A-cell space is intolerably complicated or the CP is forced to wait until all currently enqueued functions have been completed. The reader is invited to convince himself that any simple solution to this problem, which does not involve a garbage collector for the A-cell space, is actually wrong (counterexamples are easily found in which GET and PUT from the one side, or selectors used in load and store mode from the other side, do not properly interact, if an appropriate timing is assumed for their execution).

A completely different view may also be adopted: each A-cell is provided with a reference count which records the number of existing pointers to that A-cell. This has a number of pleasant consequences. First of all, the A-managers may now be executed directly by the CP, without allocating extra destination A-cells (see Fig. 9 for an example).

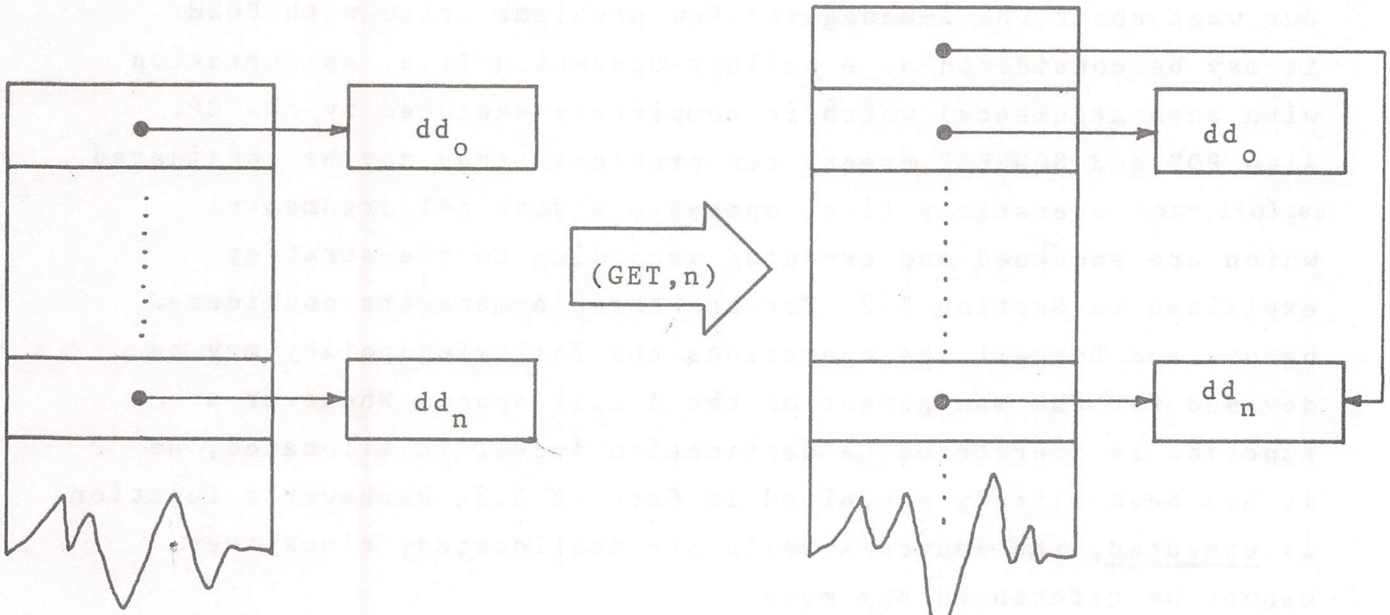


Fig. 9 - An implementation of GET.

Second, PUT may be completely prefetched without unnecessary waitings. Third, no A-cell is to be allocated for unary functions. The main disadvantage of this solution is that the maintenance of reference counts introduces a lot of write operations on A-cells: this may probably cause the performance of the system to be dramatically compromised unless a special purpose processor, the ASP, is introduced for managing the A-cell space.

3.4 - The enqueuing strategy

In Section 3.1 a strategy has been outlined for reproducing computation graphs and dispatching functions to processors. Here we make that strategy effective.

In Fig. 3 each node of the computation graph has been associated with a function. In Section 3.2 each function has been associated with a unique and newly allocated A-cell (namely, the destination A-cell). This one-to-one correspondence between nodes and destination A-cells allows us to associate the counters mentioned in Section 3.1 with the A-cells. Such counters have been supposed to be positive integers in Section 3.1. Obviously an integer counter only conveys information about how many (while the CP must know which) processors are co-operating in the computation associated with a node. An array of boolean indicators is more suitable in this case. Thus an A-cell may be patterned as shown in Fig. 10.

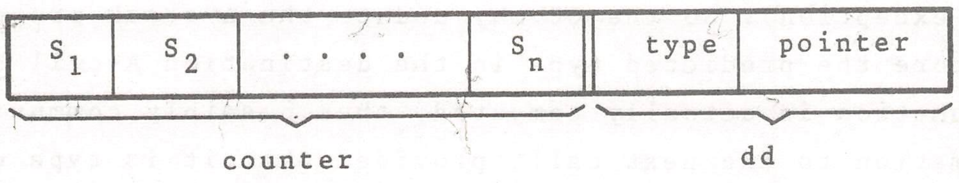


Fig. 10 - The pattern of an A-cell with boolean indicators

At any time, the indicator S_i ($1 \leq i \leq n$) of an A-cell is TRUE if and only if the processor P_i is still involved in some subordinate computation of the node represented by that A-cell. Each indicator of a newly allocated destination A-cell is initially set to the inclusive OR of the corresponding indicators of the source A-cells of the fetched function: in fact these A-cells, which are to contain the arguments of the function, also contain information about the processors which are still computing them. The function is then enqueued (with the addresses of the source and destination A-cells) on all the processors whose corresponding indicator in the destination A-cell is TRUE. If no such indicator is TRUE, the function may be enqueued on an arbitrary processor, and the corresponding indicator is set to TRUE.

When a processor takes a function from its queue, it sets to FALSE the corresponding indicator in the destination A-cell, and executes the function if and only if all the indicators of this A-cell are FALSE.

Otherwise the function is ignored, and the next one is taken from the queue.

3.5 -Type driven calls

Here we explain how the CP may perform the run-time type checking which is necessary for the implementation of the type driven call mechanism.

Suppose first that the prefetched function is not a C-manager and its code is found in the special class ANY. In this case, the arity of the function is known, and the type of its result may be often predicted (input functions are a major exception). So the CP may update the A-stack properly, and store the predicted type in the destination A-cell before the function is actually computed, thus possibly communicating information to the next call, provided that it is type driven.

If the code of the prefetched function cannot be found in the special class ANY, the type driven call mechanism is to be invoked.

If the top-type is manifest (i.e. it has been computed by either the CP or some processor) then the CP operates as in the previous case, using the appropriate class instead of the special class ANY. Otherwise the CP cannot do better than waiting until some processor has computed the top-type.

3.6 - C-managers

They can be directly executed by the CP. The execution of TYPEJUMPT and TYPEJUMPF requires that the top-type is manifest, and the CP operates exactly as in the case of type-driven calls (see Section 3.5). The execution of JUMPT and JUMPF requires that the top-datum is manifest, and the CP must wait until it is.

4. MULTIPROCESSOR ORGANIZATION

4.1 Overall system organization

As shown in Section 3, the overal organization proposed for a SMOM interpreter is a hierarchical multiprocessor, consisting of (see Fig. 11): i) a Control Processor (CP) at the higher level; ii) a set of identical (general) processors ($P_1,\dots,P_n$) at the lowest level; iii) an A-stack Processor (ASP) which manages the processors and CP requests for accessing the A-stack and controls their interactions; iv) a common Memory, with m modules ($M_1,\dots,M_m$).

As it often happens in a hierarchical, parallel organization, the existence of a supervisor control level makes both the design process and the achievment of high performances more easy, possibly at the expense of a less reliability. In our case, performance is further improved by the use of simple and fast synchronization primitives, by an intensive form of prefetch and by the independent management of the A-stack. A modular organization of the memory, possibly with a suitable separation of information among the modules, is a standard technique for balancing the execution and the memory bandwidths.

Every system unit is microprogrammed and, whenever possible and/or convenient, <u>dynamically microprogrammable</u> in order to improve system flexibility and extendibility. Moreover, the overall organization is modular in principle and thus well suited to better exploit the microprogramming characteristics. This further improves the system efficiency; as every unit has its own microprogramming language and

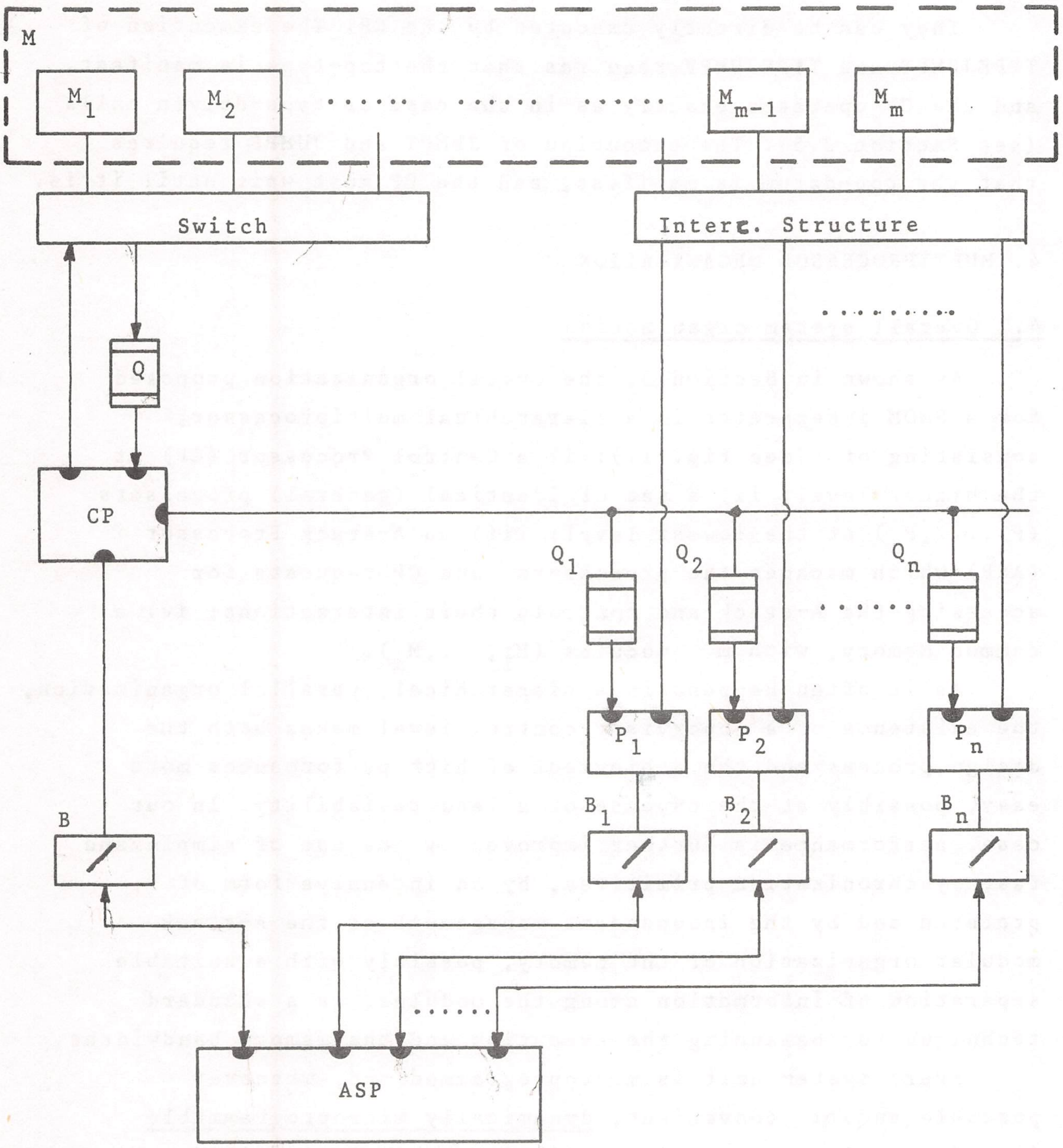


Fig. 11 - Overall system organization

organization, both chosen in order to optimize the performance/ cost ratio "on a local basis" [14].

In the following subsections we shall describe the internal organization of the main system parts, according to the general behaviour discussed in Section 3.

4.2 - Control Processor

As a primary task, CP must perform the sequence control of the computation. Therefore, the C-stack must be a private resource of CP. Moreover, it is convenient that CP contains a private copy of the A-stack, called the virtual A-stack (VA-stack), in order to reduce the number of conflicts with the processor for accessing the A-stack and to maintain a picture of the contents of the A-cells allocated to prefetched, but not yet executed, functions. It is sufficient that the VA-cells are composed of only two fields, the V-type and the V-link. The first is assigned the type of the fetched datum according to well-defined predictions as discussed in Section 3. A special mark in the V-type field signals the wait situation.

The C-stack and the VA-stack are implemented as fast memories of relatively small sizes (tipically, 128 - 256 cells). Two other similar memories are used as branch tables for registers of class ANY and for registers of the other classes. Actually, the second one acts as a cache of the complete table stored in main memory.

The CP organization is completed by a set of standard computational resources, (hardware) registers both for general and for special purposes, and buffering and interaction mechanisms interfaced with the processors, ASP and Memory.

The scheme of the microprogram defining the behaviour of CP is shown in Fig. 12. Remember that the microprograms that affect the control flow have directly executed by CP. The function APPLY, if present, is an interpreter defined for using a datum as a function: when it is called by the type-driven mechanism, it is searched in the corresponding class of the datum.

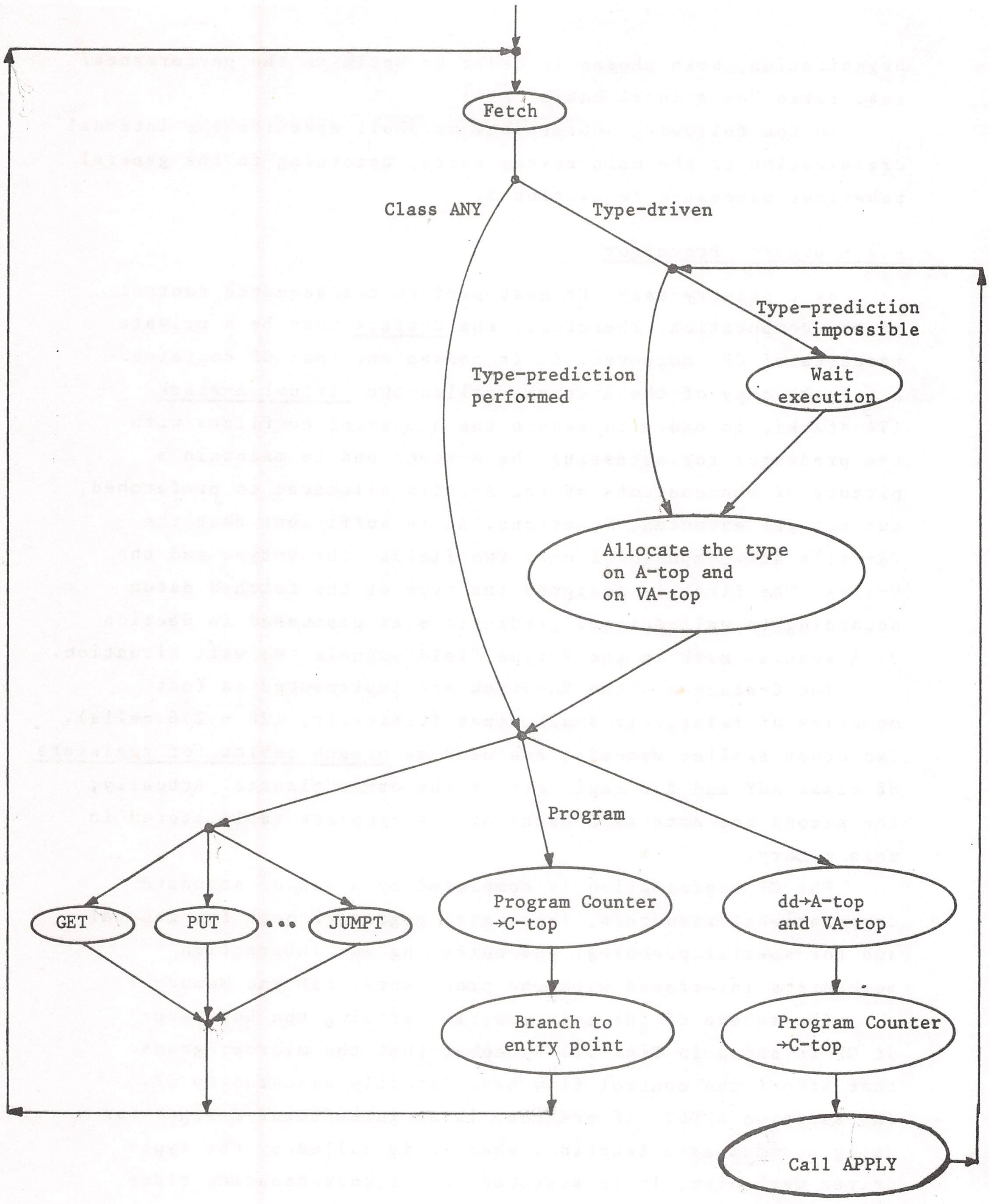


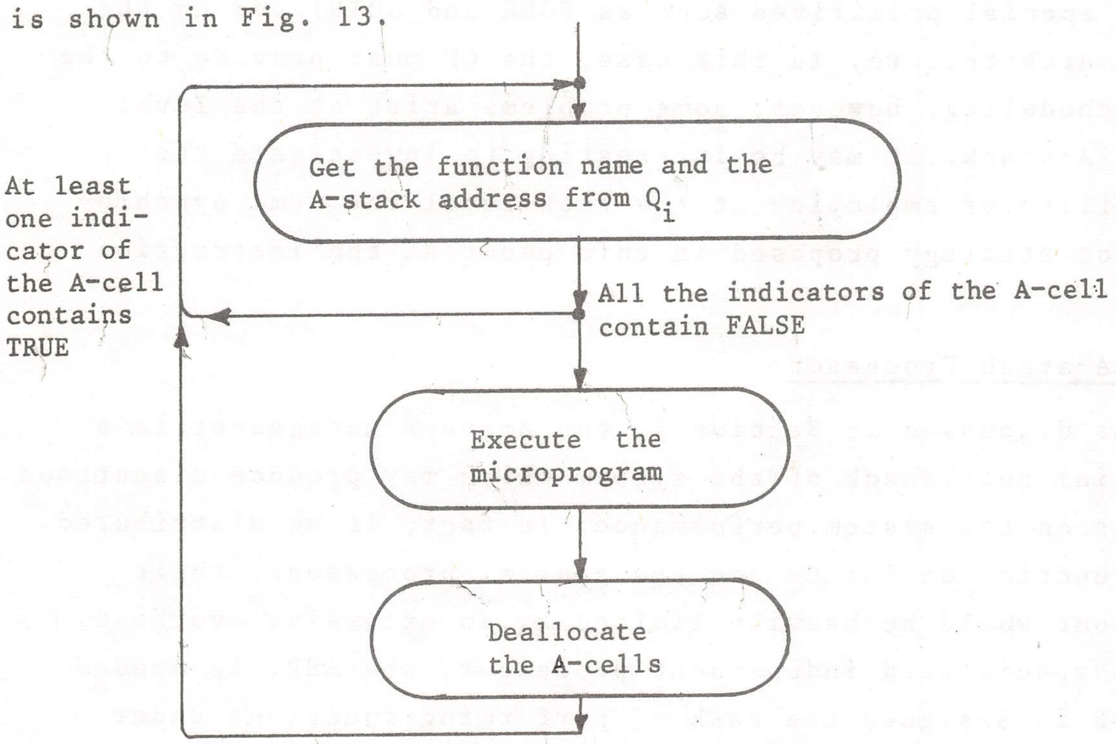
Fig. 12 - Scheme of Control Processor microcode

It can be seen that the characteristics of the algorithm being implemented are such that an horizontal microprogramming appears suitable, as CP must be as fast as possible for a continuous supply of functions to the processors. In practice, a "diagonal" microprogramming [1], is more effective, with several horizontal microinstruction formats encoding subsets of microcommands for interdependent sections of CP hardware: these correspond approximately to the main CP resources and interconnections mentioned before.

Alternatively, the CP might be realized as a network of microprocessors, each one corresponding to a "physical" section and all controlled by a supervisor microprocessor, according to the Kehl's [13] or the T.C. Chen's [3] proposals.

4.3 - General Processors

The task of a generic processor P_i is to execute the functions which do not modify the control flow. Its behaviour has been sketched in Section 3 and the scheme of its microcode is shown in Fig. 13.



Get the function name and the A-stack address from Q_i
At least one indicator of the A-cell contains TRUE
All the indicators of the A-cell contain FALSE
Execute the microprogram
Deallocate the A-cells

Fig. 13 - Scheme of generic processor microcode

The internal organization and, correspondingly, the micro-programming type depends not only on the characteristics of the implemented algorithms but heavily on overall system evaluations too. More precisely, the conflicts for accessing the A-stack and the Memory, together with the parallelism degree of the computation, impose a saturation to the system performance vs. both the number of processors and their average speed. Unless we are willing to make the Memory and the ASP organizations very sophisticated, at a non-proportional cost, the threshold speed of the processors seems of the same order of magnitude of typical mini- or microprocessors, as it happens in general for a multiprocessor architecture. This consideration leads to adopt a vertical organization and microprogramming, or at most a diagonal one with limited parallelism in the microistruction.

By taking into account the tree form of the computations, a number of 4-8 processors seems the most suitable, unless a multitasking feature is introduced in the SMOM language (by adding special primitives such as FORK and JOIN) and in the system architecture. In this case, the CP must provide to the task scheduling; however, some problems arise at the level of the A-stack. It may be interesting to investigate the possibility of employing at the task level the same synchro-nization strategy proposed in this paper at the instruction level.

4.4 - A-stack Processor

As discussed in Section 3, the A-stack management is a potential bottleneck of the system which may produce disastrous effects on the system performance. In fact, if we distributed this function on the CP and the general processors, their behaviour would be heavily limited by an excessive overhead. Thus a specialized independent processor, the ASP, is needed. The ASP is assigned the task of performing functions under the command of the CP and the processors, such as: ALLOCATE and DEALLOCATE A-cells; READ and WRITE A-cells, either with

LOCK or UNLOCK feature respectively; conflicts resolution and priority updating, queues management, and so on.

Commands to ASP are sent by the CP and the processors, and responsens are returned from ASP, through a set of input-output buffer (B,B1,...,Bn in Fig. 11). The commands specify, together with their code, the relative A-cell address and its configuration if it has to be modified.

The indirect interaction mechanism can be realized, in a simple but effective way, by associating a lock-bit to every A-cell and by implementing an arbitration algorithm of the general or of the circular type.

A fast memory of hardware registers is employed to implement the A-stack. A dynamic microprogramming, though desirable, is not essential in this case, for the functions being implemented are easily foreseen during the design phase.

Moreover, flexibility can be sacrificed in behalf of speed, owing to the critical task of this unit. The possibility of realizing ASP as a network of asynchronous, cooperating submits (at the extreme, one for every A-cell) is to be taken into account. However, economy considerations, together with a much less then proportional gain in speed, recommend to follow the proposed solution.

4.4 Memory

In a standard multiprocessor the interleaved organization of the memory and the use of a crossbar switch (as intercommunication structure between memory modules and processors) are adopted. In our case, two factors play an important role in derising a more cost effective organization of the memory. First, the particular function of the CP, roughly behaving as the Instruction Preparation Unit in a look-ahead computer [12]. Second, the possibility of individuating two memory areas, one essentially accessed by CP and the other accessed by the processors. Therefore, we can assign a subset of modules to CP and share the remaining modules among the processors. CP is connected to its modules by a simple switch and a queue

(Q in Fig. 11), while the processors are connected through a distributed crossbar or, perhaps more effectively, through a time-shared bus. This subdivision can be easily realized in a quite flexible way, provided that the switch and the interconnection structure are implemented in a modular way.

5. CONCLUSIVE REMARKS

The guidelines among which the design of the SMOM machine language evolved were essentially two, namely that the language had to be particularly well suited to implement fast interpreters for extensible high level programming languages, and that it had to support the good programming style of defining and using as many data types as many logically different categories of objects are dealt with in one's programs. Having as many data types as logically required usually means that the type of the components of a data type are almost always known from the definition of the data type itself. This amounts to saying that the prefetching algorithm seldom falls asleep while waiting for the top-type to become manifest. In other words, a programming style that allows an almost complete compile-time type checking also allows an equally complete run-time type checking to be performed by the prefetching algorithm. So, when the prefetching algorithm is waiting, it is almost always waiting for the completion of some test having a really unpredictable outcome, e.g. a test that precedes a JUMPT or a JUMPF.

A hand-made analysis of several SMOM programs (most of them being obtained by compiling various LISP functions to SMOM) has shown that about one half of the instructions contained in a SMOM program are A- and C-managers, and that the number of functions which are executed in parallel by the processors (other than the control processor) is typically varying from four to eight. These numbers perfectly match the number of processors that is suspected to be optimally supported by the proposed architecture, i.e. the maximum number of processors which does not generate too many conflicts on the A-stack (see also Section 4). Even though more accurate estimates are foreseen, we believe that the interpretation of a SMOM-like language is hardly supported by a really better organization (i.e. an organization in which many more processors are employed which are almost always busy) than the one presented in the paper. This apparently amounts to saying that the maximul speed-up allowed by parallel interpreters for SMOM-like languages is about one order of magnitude. We believe that something better can be achieved, provided that the conflicts for accessing the various storage modules and the A-stack are held to a minimum by a careful design of the algorithms for allocating new data items or new A-cells. Results obtained in implementing extensible high level languages in paged environments are surprising in this respect. Further improvements in the performance of such systems can be probably achieved only if machine languages based on completely different (possibly new) concepts are designed, even though we obviously do not know what these concepts should be.

As a final remark we point out that an interesting field for further research would be to investigate to what extent the (somewhat new) synchronization techniques presented in the paper may be profitably used to implement the well estabilished parallel control constructs which are present in some programming languages, such as the multitasking feature of PL/1.

REFERENCES

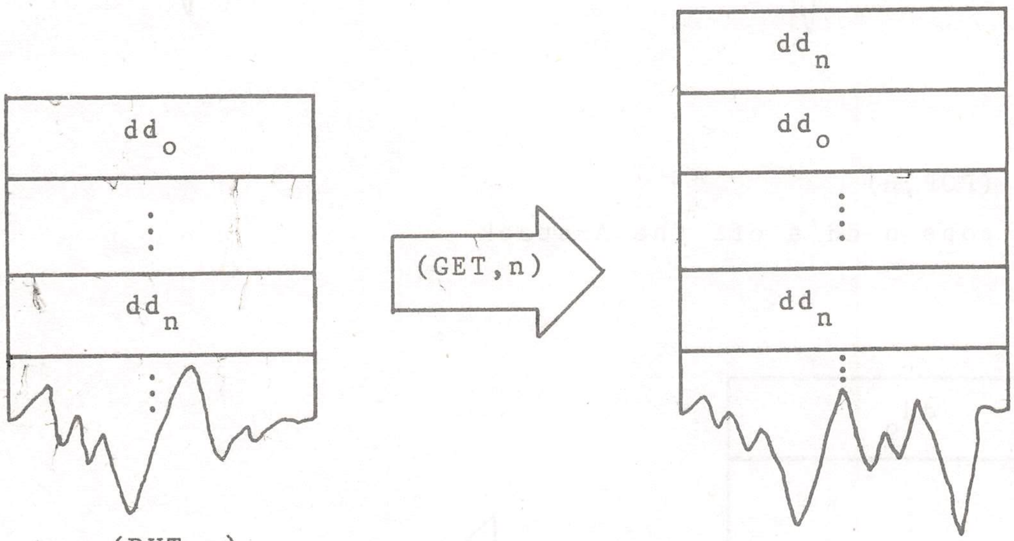
1. A.K. Agrawala, T.G. Rauscher, Foundations of microprogramming: architecture, software and applications, Academic Press, 1976.
2. G. Attardi, C. Montangero, G. Prini, A high-level machine for artificial intelligence, Proc. of the AISB Summer Conference, Edinburgh, 1976, 26-37.
3. T.C. Chen, Distributed intelligence for user-oriented computing, AFIPS Conf. Proc., 1972 FJCC, 1041, 1056.
4. M. Conway, A multiprocessor system design, AFIPS Conf. Proc., 1963 FJCC, 136-146.
5. N. De Francesco, G. Vaglini, M. Vanneschi, Implementation of parallel computation schemata, Proc. of 2nd Euromicro Symposium, Venice, Oct. 1976, 157-163.
6. J.B. Dennis, D.P. Misunas, A preliminary architecture for a basic data-flow processor, Proc. of 2nd Annual Symposium on Computer Architecture, New York, Jan. 1975, 126-132.
7. P. Deutsch, A LISP machine with very compact programs, Proc. of 3rd Int. Joint Conf. on Artificial Intelligence, Stanford, 1973, 697-703.
8. E.W. Dijkstra, Cooperating sequential processes, in Programming Languages, F. Genuys (ed.), Academic Press, 1968, 43-112.
9. E.A. Feustel, On the advantages of tagged architecture, IEEE Trans. on Comp., C-22,7, July 1973, 644-656.
10. A. Kay, Personal Computing, invited paper at the Meeting on 20 Years of Computer Science, Pisa, 1975.
11. J. Reynolds, Definitional interpreters for higher-order programming languages, Proc. of Annual ACM Conference, Atlanta, 1972.
12. H. Stone (ed.), Introduction to Computer Architecture, Science Research Associates, 1975.
13. J.A. Torode, T.H. Kehl, The Logic Machine: a modular computer design system, IEEE Trans. on Comp., C-23, 11, 1974, 1164-1169.
14. M. Vanneschi, On the microprogrammed implementation of some computer architectures, Euromicro Newsletter, Vol. 2, N. 2, 1976, 14-20.

APPENDIX

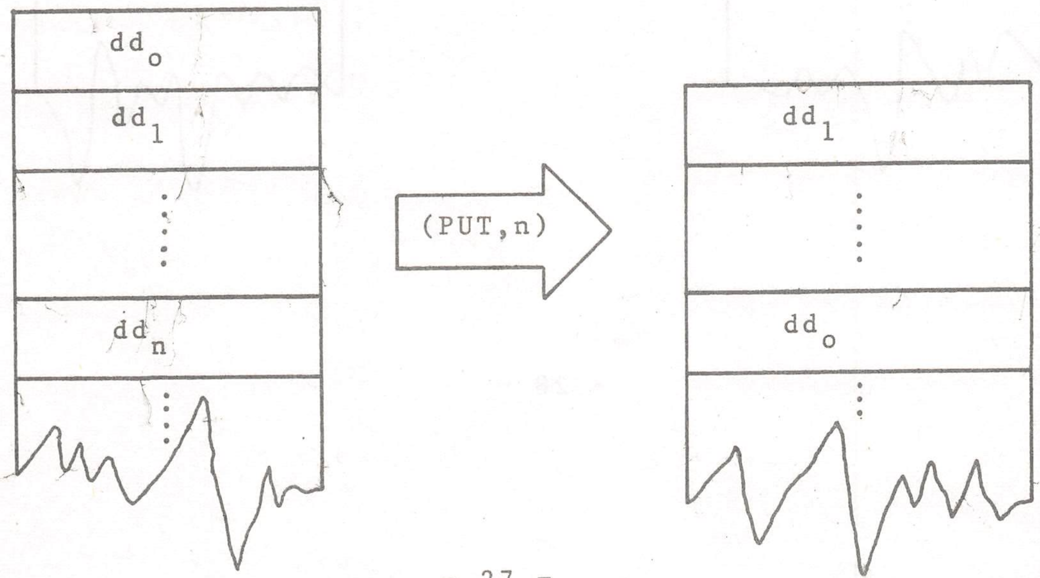
A.1 - A-managers

Here n is a positive integer which denotes the address of an A-cell relative to the A-top (i.e. 0 is the address of the A-top, 1 is the address of the A-cell immediately below the A-cell, etc.). An A-cell having address n is also called the n-th A-cell.

Format : (GET,n)
Action : copies the n-th A-cell onto the A-stack.
Picture :

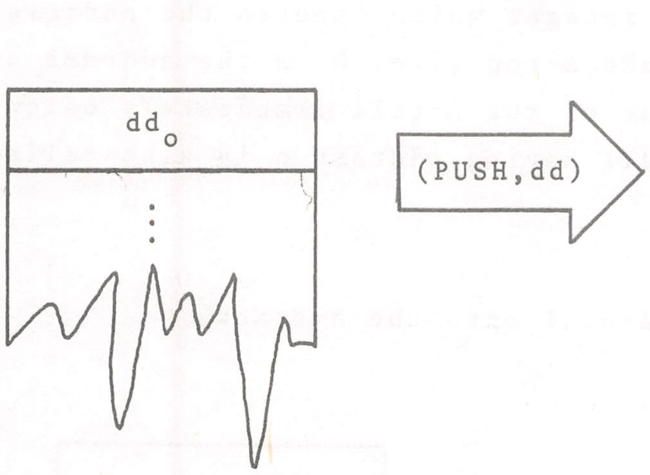


Format : (PUT,n)
Action : copies the A-top into the n-th A-cell.
Picture:



Format : (PUSH, dd)

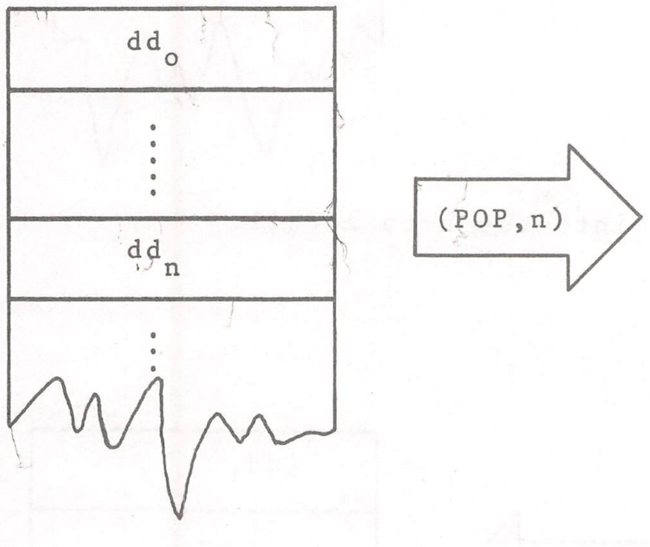
Action : copies dd onto the A-stack



Format : (POP,n)

Action : pops n dd's off the A-stack

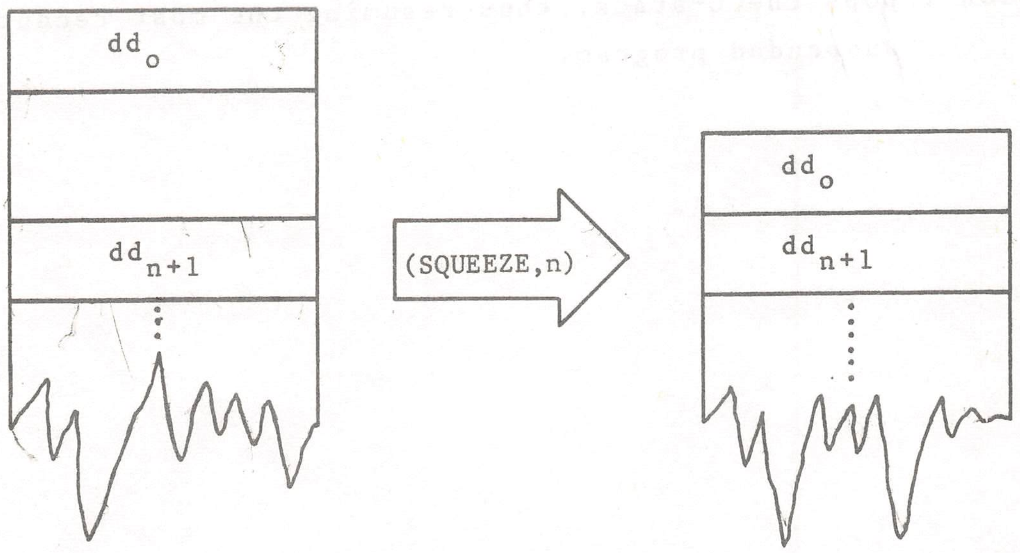
Picture:



Format : (SQUEEZE,n)

Action : pops the n dd's between the A-top and the (n+1)-th
A- cell off the A-stack

Picture:



A.2 C-manager

Here n is a (possibly negative) integers which is to be added to the C-top (such an operation is called jump); t is a type.

Format : (JUMP,n)

Action : the jump is always performed.

Format : (JUMPT,n)   (JUMPF,n)

Action : the jump is performed if the truth value (basic data type) TRUE (resp. FALSE) is found in the A-stop.

Format : (TYPEJUMPT,t,n)   (TYPEJUMPF,t,n)

Action : the jump is performed if the top-type is (resp. is not) t.

Format : (CALL,dd)

Action : pushes dd (which must be a program descriptor) onto the C-stack, thus suspending the program containing the call and activating the program described by dd.

Format : (RETURN)

Action : pops the C-stack, thus resuming the most recently suspended program.