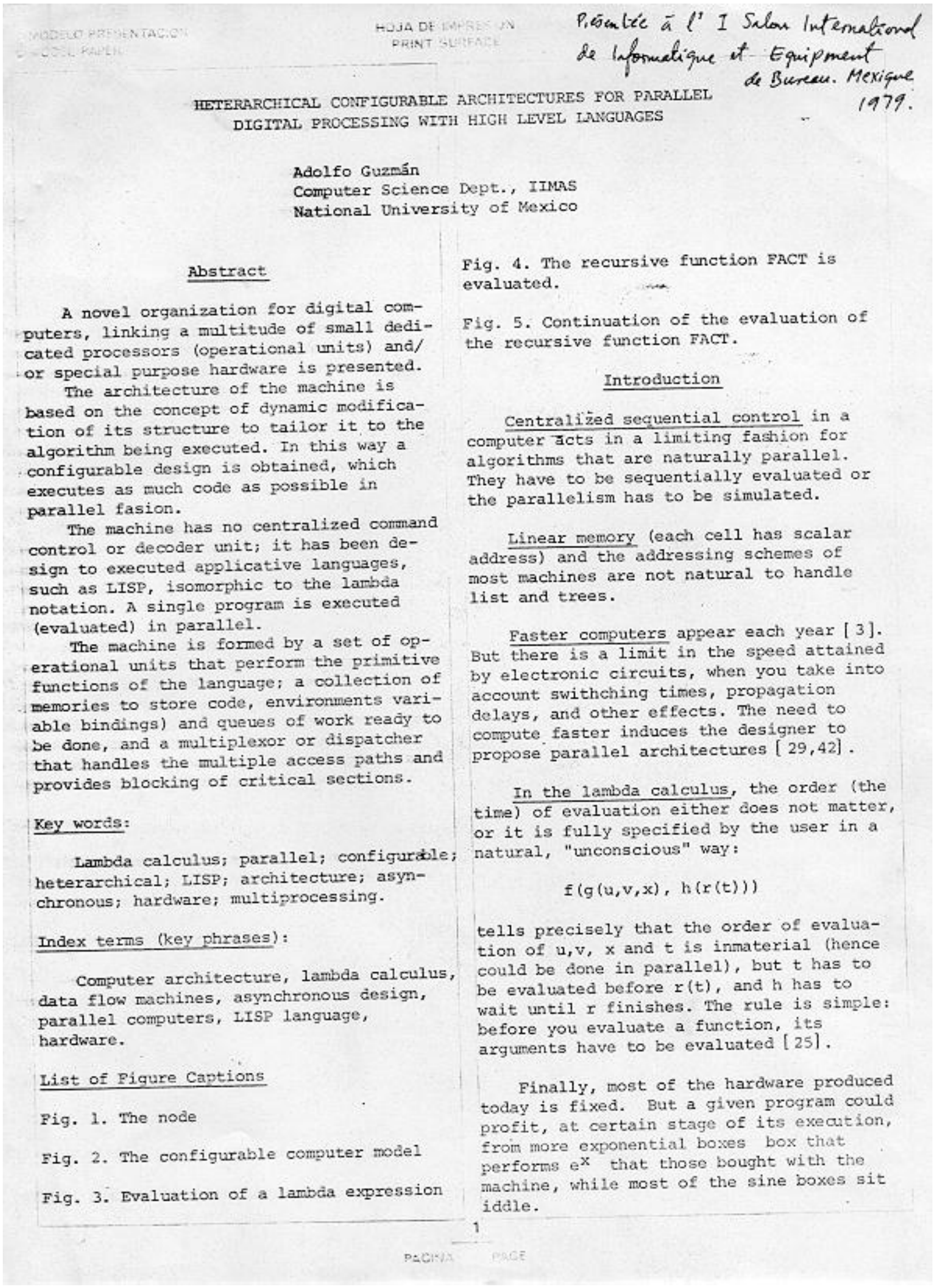


MODELO PRESENTACION
MODEL PAPER

HOJA DE IMPRESION
PRINT SURFACE

Présentée à l'I Salon International de Informatique et Equipment de Bureau. Mexique 1979.

HETERARCHICAL CONFIGURABLE ARCHITECTURES FOR PARALLEL DIGITAL PROCESSING WITH HIGH LEVEL LANGUAGES

Adolfo Guzmán
Computer Science Dept., IIMAS
National University of Mexico

Abstract

A novel organization for digital computers, linking a multitude of small dedicated processors (operational units) and/or special purpose hardware is presented.

The architecture of the machine is based on the concept of dynamic modification of its structure to tailor it to the algorithm being executed. In this way a configurable design is obtained, which executes as much code as possible in parallel fasion.

The machine has no centralized command control or decoder unit; it has been design to executed applicative languages, such as LISP, isomorphic to the lambda notation. A single program is executed (evaluated) in parallel.

The machine is formed by a set of operational units that perform the primitive functions of the language; a collection of memories to store code, environments variable bindings) and queues of work ready to be done, and a multiplexor or dispatcher that handles the multiple access paths and provides blocking of critical sections.

Key words:

Lambda calculus; parallel; configurable; heterarchical; LISP; architecture; asynchronous; hardware; multiprocessing.

Index terms (key phrases):

Computer architecture, lambda calculus, data flow machines, asynchronous design, parallel computers, LISP language, hardware.

List of Figure Captions

Introduction

Centralized sequential control in a computer acts in a limiting fashion for algorithms that are naturally parallel. They have to be sequentially evaluated or the parallelism has to be simulated.

Linear memory (each cell has scalar address) and the addressing schemes of most machines are not natural to handle list and trees.

Faster computers appear each year [3]. But there is a limit in the speed attained by electronic circuits, when you take into account swithching times, propagation delays, and other effects. The need to compute faster induces the designer to propose parallel architectures [29,42].

In the lambda calculus, the order (the time) of evaluation either does not matter, or it is fully specified by the user in a natural, "unconscious" way:

$$f(g(u,v,x), h(r(t)))$$

tells precisely that the order of evaluation of u,v, x and t is inmaterial (hence could be done in parallel), but t has to be evaluated before r(t), and h has to wait until r finishes. The rule is simple: before you evaluate a function, its arguments have to be evaluated [25].

Finally, most of the hardware produced today is fixed. But a given program could profit, at certain stage of its execution, from more exponential boxes box that performs e^x that those bought with the machine, while most of the sine boxes sit iddle.

PAGINA PAGE

The Problem to Solve

We would like to address the design of a machine that:

(1) Executes in parallel several portions of the same program; (2) Frees the programmer from the need to specify what things are computed before, simultaneously or after other things; (3) Dynamically reconfigures itself to the algorithm being executed. The machine should sense the need of different execution boxes at different times, and should convert the non-used boxes to more useful ones; (4) Exhibits tolerance to failures; (5) Has self-checking capabilities.

The Solution

The model proposed consists of:

1. The grill (work memory). It holds the program(s) being executed.

2. The (passive) memory. It contains programs, data and results that are not in the grill.

3. The boxes (operational units). Each box performs a primitive operation of the language, such as MULTIPLY, SINE, LIST. A MULTIPLY box searches the grill, through the blackboard, for a multiplication immediately. It leaves the result in the same grill cell which previously contained the multiplication ready to be done. Then it goes to look for another multiplication.

4. The blackboard (dispatcher). It points to places in the grill containing functions ready to be evaluated. The boxes look in the blackboard for work to do in the grill, instead of searching directly the grill itself.

An expression to be evaluated is placed in the memory by some I/O mechanism. At the end of the I/O mark on the blackboard is placed. Such mark is a pointer to the head of the expression in memory. A mark on the blackboard is a demand for attention of the boxes. In the particular case of the initiation of an execution (an evaluation) a START box is called in order to build the head node in the grill, from the code (the expression) placed in memory, and from then on a tree is built (a tree of stacks) on the grill [7].

Each call for a function (each node of the tree on the grill) has a counter of unevaluated arguments that is decremented in one for each evaluated argument. (Notice that the arguments are evaluated in parallel). This decrement is performed by each box that evaluates an argument. In addition, the box that converts such counter to zero will place a mark on the blackboard, meaning that this function, because it has a counter of zero, is now ready for evaluation.

The values simply replace (take the place of) the evaluated expression, in the grill. Eventually the whole tree of the original program will collapse to a data-object (a number, a list), this being the end of the evaluation of the expression (that is, the end of the execution of the program).

Example:

1. The expression to be evaluated is

```
(MULTIPLY X (SIN Y) Z).
    3     0   1  0  0
```

The numbers below are the counters of unevaluated arguments. Variables X, Y and Z are ready for evaluation.

2. Two variables boxes attach themselves to Y and Z, and (after some time) yield values:

```
(MULTIPLY X (SIN 30) 20)
    2     0   0   =   =
```

Notice that the variable box that evaluated Z places '20' instead of Z, and a 2 instead of a 3 as counter of MULTIPLY.

3. The variable box that evaluated Z now goes on to evaluate X, and a SIN box now comes to evaluate (SIN 30). After they finish, the expression is:

```
(Multiply 3 0.5 20)
    0     =  =   =
```

4. The zero under MULTIPLY triggers the visit of a MULTIPLY box; after its execution the expression becomes

completely replacing the original (MULTIPLY X (SIN Y) Z).

One has to notice that this process is being held in parallel; the grill is changing and the boxes are placing constant demands on the blackboard, where other boxes are continuously searching for operations to perform. The limiting factor in parallization is the amount of boxes. A process of reconfiguration (explained later) takes place whenever demands for certain type of boxes exceed its availability.

In order to easily incorporate concepts of high level languages, the grill's architecture is based on a tree structured memory [7] where the nodes fig. 1 are stacks, providing an environment to handle free variables, bindings and return's to the calling environment (static and dynamic links) and a set of pointers to the arguments.

The grill is organized in active nodes that are being processed, and a pool of free nodes. We may think that each node is a memory bank with asyncronous access to it. (Figure 2). Nodes may be exhausted causing a machine error.

Blackboard.- The blackboard is a linear memory divided into areas where boxes place marks pointing to nodes demanding evaluation. Areas are divided by box type; the division is with soft registers that can be moved as demands grow. The blackboard may be exhausted (eg. in an infinite recursion), causing a machine error. Boxes place marks in the first free slot available. The blackboard makes a "searcher" mechanism [33] unnecessary.

Memory.- Passive memory contains programs, data and results. The grill usually points to some of these. Certain boxes copy programs from memory to the grill for evaluation, and copy results from the grill. I/O transactions take place between memory and external devices.

The Lisp Machine

To pinpoint some of these ideas we propose a LISP machine, that is, a machine that processes LISP [24] programs residing in memory as S-expressions. We consider "pure LISP". S-expressions are copied into the grill by some boxes as evaluation take place.

We have boxes for the LISP primitives CAR, CDR, CONS, ATOM, EQ, plus some programming functions LAMBDA, IF, AND, etc. [31]. The following classes of boxes are needed:

- Primitive and programming operations CAR, LAMBDA,...
- Boxes that handle calls from users defined functions.
- Converter (special box. See RECONFIGURABILITY).
- I/O (special box, see INPUT-OUTPUT).

An s-expression is read into memory by the I/O box, and a node pointing to it is created in the grill; then execution begins.

The performance of the LISP machine is now explained.

- VARIABLES. A variable is evaluated by finding its value in the current a-list. In our case a stack is formed in each LAMBDA node, and a pointer is held to the calling environment.

- IF. The format of this expression is (IF P Q R). If P is true, Q is evaluated; otherwise R is evaluated [31]. Initally an IF node is created, having only the argument P and a pointer to the rest of the code in passive memory. P is evaluated leaving a TRUE or FALSE value. The IF box goes into action bringing Q or R and also setting the pointer to the code to nil, the counter of unevaluated arguments to 1 and then allows the evaluation of the argument to proceed. If the pointer to the code is nil this means that the argument to be evaluated is Q or R. Since the IF delivers values at various stages (first P and then Q or R), it frees nodes.

- LAMBDA. Binds variables in LISP and evaluates an expression (form). It can be used as a primitive: ((λ(X) (CDR X)) (QUOTE (A B C))) or in a function definition. The binding takes place in the stack built in the node (see figure 3). Notice that the access and control links both point to the calling frame. Bindings for free variables are traced up in the call structure chain. A special case is the FUNARG problem where access and control links do not point to the same places [8].

Whenever a LAMBDA is placed on the grill, the arguments are evaluated, producing a zero in its counter of unevaluated arguments. Then a LAMBDA box is demanded, which in turn binds the arguments with its variables, placing a 1 into its counter and having as argument a pointer to the form, which is now placed on the grill by the LAMBDA box. At the same time a zero is placed in the code link indicating that when evaluation of the form has ended, the LAMBDA box also should end and should leave the node transformed into an s-expression (a result). As soon as node is not used it gets into the pool of free nodes (figure 3).

- RECURSION is handled by expanding the tree in the grill, replacing the name of the function (each time is called) by its definition. We examine the case where user-defined functions are evaluated. Assume that the name of a function and its arguments are already on the grill, in some node. Its arguments are evaluated and a zero is placed in its counter. The box that decrements the counter and makes it zero also places a demand in the blackboard in the section of the DEFINED-FUNCTION boxes. Once this demand is attended such box erases the name node replacing it by a LAMBDA node; in its counter places a zero and the grill continues its evaluation as a regular LAMBDA. When it finds the recursive call, the process is repeated. Notice that the name node and the IF node in the example are replaced and the only thing that remains is a chain of TIMES nodes, that unwinds when n is zero, transmitting a result to higher nodes. (figures 4, 5).

Other LISP functions are computed in a similar manner. For instance, (AND F1 F2 F3 ...) places F1 on the grill and holds its other arguments in passive memory. If F1 is FALSE, the AND becomes FALSE itself; otherwise F2 is placed on the grill, etc.

Reconfigurability

Our machine consists of a mixture of boxes; this mixture may not be optimal for the whole execution of a program, hence the need to change it. It is through this change of composition that the machine reconfigures itself.

To make these changes, the machine needs:

1) To detect which type of boxes are idle most of the time t, where t is a small interval. Reconfiguration occurs at the end of each period t.

2) To detect which type of boxes are very busy during the same period t. These two actions (1) and (2) are performed with the help of a "laziness" counter attached to the blackboard of each type of boxes. It contains the number of times boxes of the type in question didn't find anything to do, in the period t.

3) To decide what box(es) to change into what other(s).

4) To make the change, that is, to reconfigure. A CONVERTER box seizes the (iddle) box chosen in step 3 for metamorphosis, and plants in it a new microprogram, converting it into one of the desired type.

5) To reset to zero the "LAZINESS" counters of (2). To initiate a new reconfiguration period t.

6) To have some statistics ready to answer the owner's question "what new boxes should I buy next?". This feature is optional.

INPUT OUTPUT

We can regard our LISP machine as sharing its passive memory with a general purpose computer, which performs the necessary I/O.

Other Properties of the Machine

(a) The machine can process any language expressed as a lambda calculus formalism.

(b) Unless the user explicitly wishes to do it, there is no way to specify sequential evaluation when it is not needed or desired.

(c) There are no GOTO's, LABELS or CALL EXIT (STOP. The machine never stops: when a program gets evaluated (reduced) to an s-expression, all the boxes become iddle, looking for work to do.

(d) The machine does not need a compiler or interpreter to execute the high level language.

(e) There is no central control, or program counter. There is no machine cycle. It is an asynchronous architecture.

(f) There are no interrupts, deadlocks or priorities. A box never interrupts its execution.

(g) Boxes of different speeds, configurations and complexity can coexist.

(h) The machine is modularly expandible. [2].

(i) The machine performs maximal parallelization: given a program, the machine tends to perform the algorithm in the fastest possible way. To show this, first observe that there are no waiting boxes: each box is either fully busy performing its function, or it is looking for something to do. Functions get evaluated as soon as their arguments become ready, except when all the boxes of its type are busy. Even this situation is detected at the next reconfiguration period, which may possibly create new boxes of the type most urgently needed.

   Of course, faster computation could be achieved if we: change the algorithm; make (some of) the boxes faster; add more boxes.

(j) The machine has self-checking capability. A TESTER box periodically checks the boxes.

Related Work

There is much activity in the field of parallel computation [5,14,29]. Most computers offer some kind of parallelism computation [1,20,40,43]. There have been recent meetings [35,36,38]. Several universities engage in this research [34,39, 43].

Lambda-calculus machines. In 1971, [7] formalizes the way to evaluate lambda-expressions in a machine. Of course, McCarthy's paper [25] is classical.

High level language machines. The idea to use a high-level language as a machine language is not new. A Fortran machine [4] has been proposed. Hewlett-Packard has a BASIC machine. APL machines also exist. B5500 and B6700 are inspired in Algol [18]. Deutch [12] presents an architecture specially suited to LISP.

Pipelines. The idea of many units jointly transforming a set of data is used in pipeline architectures [27,37]. We can regard the pipeline as a special type of grill, where flow of partial results follows rigid paths.

Tag machines. B6700 already has tag bits [18]. Feustel [15] generalizes this concept.

Asynchronous computation. Patil [34] studied the asynchronous evaluation of lambda-expressions. Rumbaugh [39] designed a highly parallel machine for programs expressed in a data flow language. He has dormant as well as active programs; his data structures are vectors of values. A structure is shared instead of copied, for use by several concurrent activations.

Register transfer modules. Bell [6] postulates boxes that can be easily connected and reconnected to perform arbitrary computations.

Architectures resembling ours. Miller [32,33] describes a configurable machine based on a searcher that feeds the operational units with new tasks to perform. It is not configurable in our sense. He uses an n by n interconnection network for reconfiguration. Glushkov [17] presents a recursive computer architecture that is similar to ours in the sense that all program elements for which operands are available are to be executed by boxes. As in our machine, his architecture allows the removal of interruption processing programs. Actors [21] are relevant. Kautz [26] also places logic into the memory. A theory which is also relevant is [28].

Current Troubles

We do not like the complexity surround ing each box. They are simple in themselves, but they have to search the backboard, subs tract one to the father of the node just evaluated, have a "laziness counter", etc. To support such overhead, the boxes will have to be big (perform large operations), thus increasing the probability that large parts of the hardware are not doing anything useful, because inside each box computation takes place most likely in a serial fashion. Said differently, I will rather place an adder in an ADDER box so as to have it free to look for work at the grill (any addition to perform), than place such adder inside a SINE box, because then such adder will work only when there are sines to compute; and even then only during part of the sine cycle. Hence this adder is iddle most of the time. The overhead associated with each box is large (in number of circuits, say) and forces us to use larger boxes. Hence, a cheaper overhead will allow more "elementary" boxes to access the blackboard directly, provoking a better utilization of hardware.

Conclussions and Recommendations for Future Work

We have proposed a computer structure where several boxes "cook" in parallel the program laid into a "grill" or work memory. There is no need to explicity synchronize such boxes; the program itself provides the time constraints.

In most machines, the operating system allocates resources to the different programs in execution, and controls execution times. In this machine, there is no such central administration: each box knows what to do, and it does it as soon as it is able.

We have described the behavior of such machine for programs written in LISP, although the approach is valid for computations expressed in the lambda notation.

This machine departs from the traditional architectures in a number of ways described in the paper. Also, the proposed architecture presents new problems, some of which also received attention.

Error recovery has to be taken into account: the machine needs changes to handle LISP errors.

Acknowledgments

To Mario Magidin, of Universidad Metropolitana.

References

1. Abramson, N. The ALOHA System. Computer Communication Networks. Prentice Hall.

2. Alvarez, Manuel. A LISP Growing Machine. M.S. Thesis, Moore School of E.E.U. of Pennsylvania. 1967.

3. Auerbach Computer Technology Reports. Auerbach Publishers Inc. Philadelphia, Penn.

4. Bashkow, T.R. Sasson, A., and Kronfeld, A. System design of a FORTRAN machine. Chapter 31, 363-381. In 5 .

5. Bell, C.G., and Newell, A. (Eds). Computer Structures: Readings and Examples. McGraw Hill. 1971.

6. Bell, C.G.; Eggert, J.L.; Grason, J., and Williams, P. The description and use of register transfer modules (RTM's), IEEE Trans. on Comp., C-21, 5, 495-500, May 72.

7. Berkling, Klaus. A computing machine based on tree structures. IEEE Trans. Computers, C-20, 4, April 71.

8. Bobrow, D.G., and Wegbreit, B.A. model for control structures for artificial intelligence programming languages. Proc. Third Intl. Jt. Conf. on Art. Intelligence. 1973. 246-253.

9. Control Data 6000 Series Computer Systems. Reference Manual. Control Data Co. Minneapolis, Minn.

10. Cornell, J.A. Parallel processing of ballistic missile defensive radar data with PEP. COMPCON 72, 69-72. We refer to the PFOR language of this machine.

11. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogramming

computations. Comm ACM, 9,3, March 66. 143-155.

12. Deutsch, L. Peter. A LISP machine with very compact programs. Proc. Third Intl. Jt. Conf. on Art. Int. 1973. 697-703.

13. Digital Equipment Co. PDP-11 Peripherals Handbook 1975.

14. Enslow, P.H. (ed). Multiprocessors and parallel processing. John Wiley. 1974.

15. Feustel, E.A., On the advantages of a tagged architecture. IEEE Trans. on Comp. C 22, 7, Jul 73, 644-656.

16. Gleary, J.G. Process Handling on Burroughs B6700. Proc. Fourth Australian Comp. Conf. 1969.

17. Glushkov V.M., et al. Recursive machines and computing technology. Proc IFIP 1974, North Holland, 65-70.

18. Hauck E.A., and Dent, B.A. Burroughs B6500/B7500 stack mechanism, Proc. SJCC 1968.

19. Guzmán, Adolfo, and McIntosh, H.V. CONVERT. Comm ACM 9, 8 Aug. 66.

20. Heart, F.E., et al. A new minicomputer/multiprocessor for the ARPA network. Proc. AFIPS 1973 Natl. Comp. Conf.

21. Hewitt, C., et al. A universal modular ACTOR formalism for artificial intelligence. Proc. Third Intl. Jt. Conf. on Art. Int., 1973, 235-245.

22. Higbie, L.C. The OMEN Computers: associative array processor. COMPCON 72, 287-290. We refer to Fortran and Basic of this machine.

23. IBM System 370 Principles of operation. GA22 7000 4.

24. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM 3, 4 April 60.

25. John McCarthy et al. LISP 1.5 Programmer's Manual. MIT Press. 1962.

26. Kautz, W.H., and Pease, M. C. Cellular logic-in-memory arrays. National Tech. Information Serv., Nov. 71, AD763710.

27. Keller, Robert E. Look Ahead processors. Computer Surveys 7, 4, Dec. 75.

28. Landin, P.J. A program machine symmetric automata theory. In Machine Intelligence 5, Edinburgh Univ. Press. 99-120.

29. Lorin, Harold. Parallelism in hardware and software: real and apparent concurrency. Prentice-Hall, 1972.

30. Lawrie, D.H., et al. GLYPNIR-A programming language for ILLIAC IV. Comm. ACM, March 75, 157-164.

31. Magidin, M., and Segovia, R. LISP B6700 Implementation. Technical Report 5, 70, Computer Sci. Dept., IIMAS, National University of Mexico. 1974.

32. Miller, R. A comparison of some theoretical models of parallel computation. IEEE Trans. Comp., Aug. 73, 710-717.

33. Miller, R., and Cocke, J. Configurable computers: a new class of general purpose machines. Intl. Symp. on Theoretical Programming. Ershov and Nepomniaschy (eds) Springer Verlag, New York 1974, 285-298.

34. Patil, S.S. An abstract parallel-processing system. S.M. Thesis, E.E. Dept. M.I.T. June 67.

35. Proc. Symp. on Computer Architecture Dec. 73 Avail from IEEE.

36. Proc. 1975 Sagamore Computer Conference on Parallel Processing, August 19-22. Available form IEEE.

37. Raytheon Data Systems Array Transform Processor. Raytheon Data Systems. Norwood, Mass.

38. Record of Project MAC Conference on concurrent systems and parallel computation. June 70. Available from ACM.

39. Rumbaugh, J.E. A parallel asynchronous computer architecture for data flow programs. Ph.D. Th., EE Dept., M.I.T.

40. Segovia, Raymundo. Computer networks for load sharing. To appear as a Technical Report, Computer Sci. Dept., IIMAS; National University of Mexico.

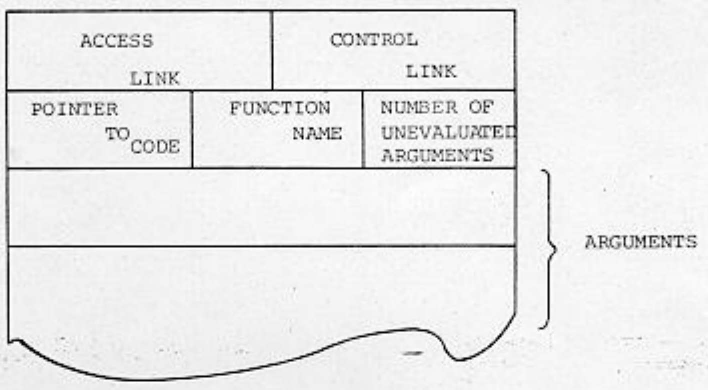
41. Thornton, J.E., Design of a computer: the Control Data 6600. Scott, Foreman & Co.

42. Rhurber, J.K., Associative and Parallel Processors. Computer Surveys (ACM) 7, 4, Dec. 75.

43. Wulf, W., and Bell, C.G. "C. mmp - a multi-mini processor ". Proc. AFIPS 1972 FJCC.

FIGURE 1.

THE NODE



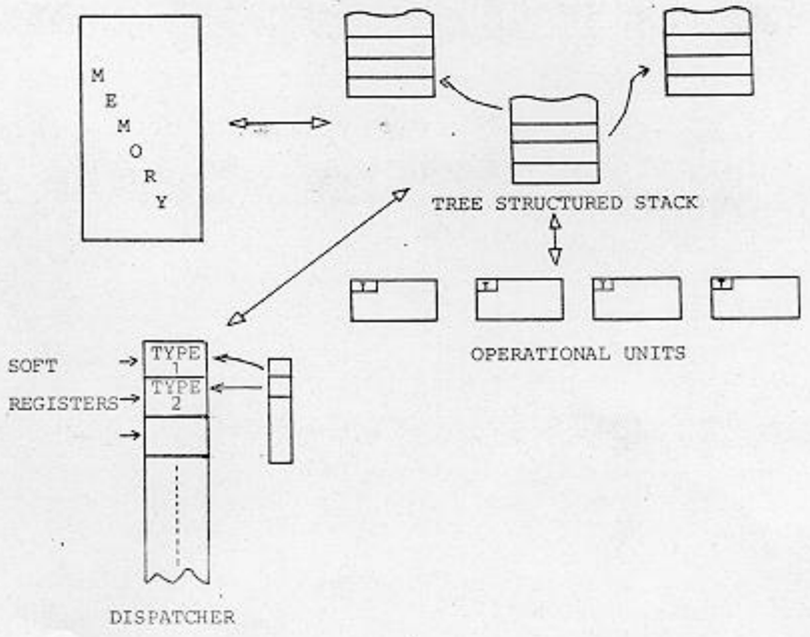


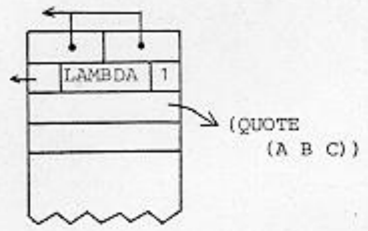
FIGURE 2.

THE CONFIGURABLE COMPUTER MODEL

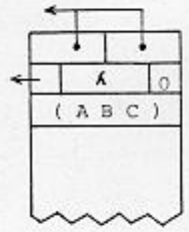
FIGURE 3.

EVALUATION OF A LAMBDA EXPRESSION.

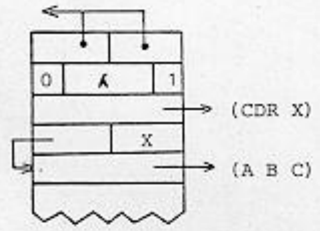
EXAMPLE:((LAMBDA (X) (CDR X)) (QUOTE (A B C)))



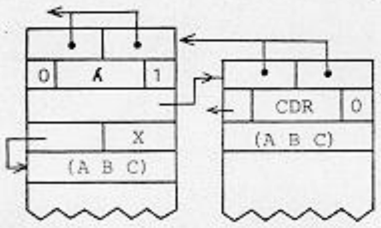
The argument has to be evaluated.



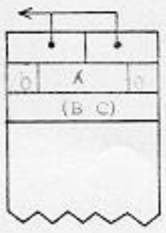
A call to the LAMBDA box is placed and binding of argument X will take place.



A zero is placed on the code link - and a 1 on the counter.



The X is replaced by its value.



The value (B C) is passed to a higher level.

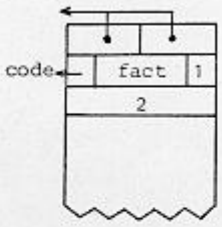
The LAMBDA node disappears.

FIGURE 4.

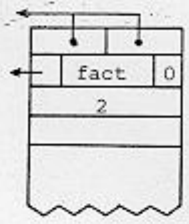
THE RECURSIVE FUNCTION FACT IS EVALUATED.

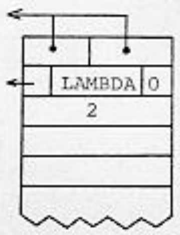
```
DEFINITION: (FACT (LAMBDA (N)
                  (IF (ZERO N) 1
                  (TIMES N (FACT (SUB N 1)))
                                          )))
```

EXAMPLE: (FACT 2).

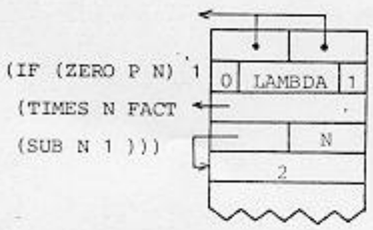


This is the initial node

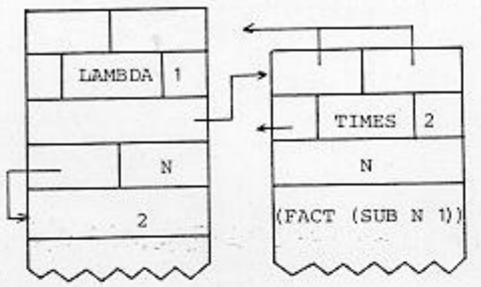




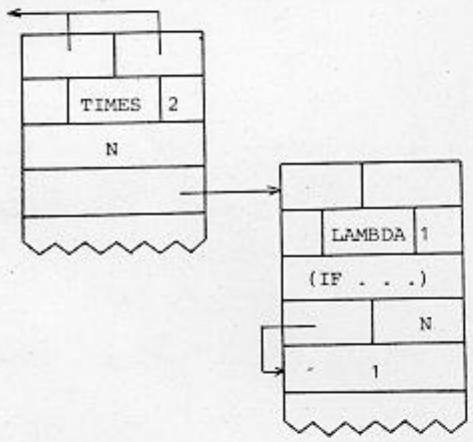
FACT disappears and is replaced by LAMBDA, - having a 0 on its counter. This zero triggers the visit of λ-box, that binds the argument - with the variable.



The LAMBDA BOX has finished; a 1 is placed and the body is passed as the argument of λ.



The only nodes that remain are the λ and the TIMES nodes.



Here we expand the (TIMES N ...). The process finishes when N=O.

FIGURE 5.

CONTINUATION OF THE EVALUATION OF THE RECURSIVE FUNCTION FACT.