M3L : A LIST-DIRECTED ARCHITECTURE

J.P. SANSONNET
M. CASTAN
C. PERCEBOIS

Laboratoire "Langages et Systèmes Informatiques"
Université Paul Sabatier
118 route de Narbonne 31077 TOULOUSE CEDEX - FRANCE

ABSTRACT

This paper describes the basic principles and the architecture of a general host machine based upon lists processing. Current works in this field are dealing with conventional direct execution schemes which use lineary structured Directly Executable Languages : prefixed languages with varying formats for operators and operands. If these languages are convenient for interpretation and provide an efficient execution scheme, on the other hand, they are very hard to generate.

Therefore, we propose here a new direct execution model based upon the definition of a class of Directly Executable Languages with a list oriented structure using LISP as model. The first part of the scheme is held by an editor which translates the high level source-text into the internal tree-structured form. The second part is held by an interpreter which executes this form on an appropriate machine.

In this paper we pursue the design of the list-structured intermediate form and we give the reasons of our choice. Once we have brought out the concepts and the functions required for the implementation of non-numerical processing and particularly for list-structured forms, we discuss the architecture of the lists-directed machine.

1 - A LIST-DIRECTED ARCHITECTURE

1. Direct execution

For ten years, many architectures have been developed in order to support one or more high level languages by using a new execution scheme called direct execution. This scheme differs from current implementations by the elimination of the machine-language. Various direct execution schemes were proposed. Many authors think as CHU [1], that the hardware structure can execute directly the external form of high level languages, but most of them suggest a less extreme solution which uses an internal form to represent the source-program. Typical direct execution schemes are made of two steps : The first one, software, analogous to the compilation in the conventional models, translates the source-text into an intermediate form which has to be directly executable. The second one, hardware, holds this Directly Executable Language (DEL) [2] and interprets it in a microprogrammed way.

Recent advances in technology of fast RAM memories have involved an important development of microprogramming and in particular of dynamic microprogramming. The ability to write powerful and swappable microinterpreters dictated the giving up of the machine-languages in favor of DELs.

Many DEL forms were suggested. The most advanced are the polish forms such as the P-code or I-PASCAL [3], but most of them are derived from the S-languages of the Burroughs B1700 [4]. In any cases, they have the main common feature which has prompted them the name of linear DELs : the text is always represented by a sequence of statements including op-codes and operands.

2. Criticism of linear DELs

The purpose of HLL-processors is to reduce, as much as possible, the semantic gap between the high level languages and the hardware structure. The semantic gap [5] is the measure of the difference between the concepts of the high level languages present on a machine and the concepts of its hardware structure. In a Von Neumann architecture there is no concept of the high level languages (e.g. arrays, structures, procedures, blocks, recursivity ...) which is directly connected with a concept of the hardware structure, and this process is bound to increase in the future.

Hence the DEL takes place in the semantic gap. It can be confounded neither with the high level language itself, nor with the machine-language. What are the criteria for choosing a good DEL ? The direct execution scheme is made of two environment transfers. The first one deals with the translation from the external form into the internal one : the translator has to process the external form so as to produce a language with the same semantic level but a simplified syntax. The second environment transfer reexamines it and has to interpret it in a hardware manner. Hence, an ideal DEL must be a good output environment for the translator and a good input environment for the interpreter.

Today's DELs are often rather good input environments for the microprogrammed interpreters. As they use to paraphrase the conventional machine-languages they are string related to the hardware structure. One of the aims of DEL designers was the compactness of the intermediate text [6]. At first, this factor seemed to be rewarding because of the saved memory space and the faster access to the whole text, but as long as environment transfer is concerned, it

CH1494-4/80/0000-0105 $00.75 © 1980 IEEE

appears to be a major drawback. As a matter of fact, the frequency-based encoding involves the intermediate form to be very complex, and then, hard to encode and decode. ORGANICK notices in [7] that, on the B1700, writting the access to the DEL is as difficult as writting the very interpreter. In our previous works on emulation, we defined an efficient hardware tool for bit pattern manipulation [8] in order to relieve the microprogrammer of the burden of accessing to the intermediate DEL forms, but such hardware mechanisms are heavy and unaesthetic. They result from a bottom/up approach and they overload the architecture.

Today's DELs are often bad output environments for the translator. Due to their sequential form, the semantic power of many DELs is far from that of the HLL text. In addition, the compactness involves the definable length formats which are very hard to generate. Thus, the precompilers turn out to be as complex as conventional compilers.

It is therefore necessary to redefine the DEL forms so as to make both compilation and interpretation steps as simple as possible.

3. The choice of a list-structured form

In advanced edition processors, the tree-structured form is the abstract representation of the source-texts which is the most developed. Here,with respect to the environment transfer, it exactly plays the role of an ideal DEL form. As a matter of fact, it gives us the expected advantages :

. It is a good output environment for the compiler : the source-text/tree-form translation can be achieved in a simple way by using the concepts of edition : top/down recursive parsers, syntactical edition ... If the source-text is maintained in the internal representation, the IN/OUT transformation is reversible and the compiler turns out to be an editor which involves direct benefits in man-machine communication.

. It is also a good input environment for the interpreter : without anticipating the exact structure of the internal form, the associated syntax will probably be trivial. Parsing is reduced to a simple tree exploration, i.e. a very simple recursive process which only requires few instructions.

With respect to the tree-structured internal form, we had to choose between a n-uple and a binary form. The object of the n-uple representation is to place the functions (operators) in the nodes of the tree and operands in the leaves. Hence, this form is dealing with two types of objects : on the one hand, nodes are n-uple cells including a function name and a varying list of pointers towards the arguments.On the second hand, leaves are primitive objects including a directly evaluable information. In the binary representation operators and operands are all placed in the leaves, nodes are only responsible for logical connections. Nodes are pair-cells consisting of a left pointer and a right one. Leaves include either functions to be executed or directly evaluable information.

Although the n-uple representation is today the most popular, especially in advanced edition systems, we have chosen the binary form for several reasons : the fixed size of the nodes eases the building and the traversing of trees, it is therefore a good environment for both translation and interpretation. As functions and operands are at the same level,the binary form is potentially more powerful than the n-uple representation. Lastly, it shows up many similarities with the internal representation of the LISP language.

We cannot present here the LISP language [9] nor discuss its implementation, but the great convenience of this language for non-numerical processing is well-known. This is not pure fortune,because the list-structured representation, used in LISP,is versatile and powerful. The main characteristic of LISP is that its external syntax is quite equal to its abstract syntax, represented within the computer by a binary tree. The LISP syntax is approximatively that of DELs, as specified above. It is therefore an excellent model for our execution scheme.

2 - THE 3L-MODEL

Consequently our direct execution scheme is based upon an internal list-structured form using LISP as model. For each high level language a LISP-Like-Language (3L) is defined (e.g. for PASCAL : P3L, for COBOL : C3L, etc...). For each 3L form an editor is written. It ensures a bidirectional communication between the external form (source-text) and the internal one (3L form). For each 3L form, an interpreter is written. It evaluates the associated 3L form on a hardware processor, specialized in list processing : the 3L-MACHINE (M3L). For any high level language the direct-execution scheme, described in the figure 1, is obtained.

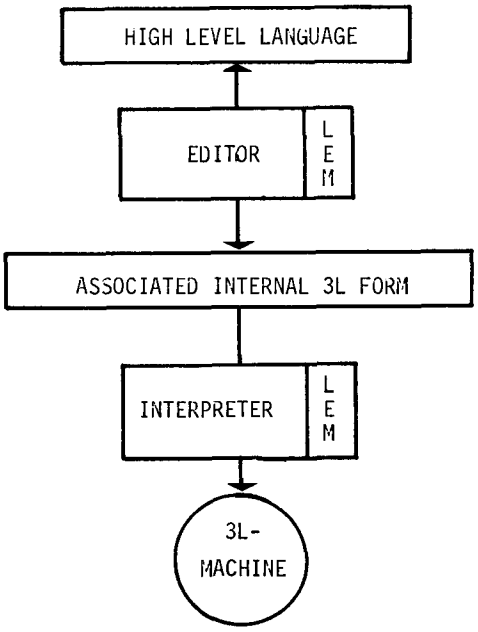


Figure 1 : THE 3L-MODEL

The LEM language

The 3L-model includes two processors which execute two different tasks. Edition processing is essentially non-numerical and consists in processors such as : lexical parsers, syntactical parsers, editors. Interpretation processing is mainly associated with the environment transfer. Thus, in the edition step as in the interpretative step, non-numerical processing is prevalent. It will therefore be expressed in a single language with the intend of supporting the emulation functions ; the Language for EMulation (LEM).

LEM is a high level microprogramming language. On the one hand, its goal is to support the emulation. On the second hand, the great symbolization of LEM makes its use easier for persons who do not have to be specialized in the hardware structure.

. The 3L-machine

The 3L-machine is an emulation which can deal with any language provided that before any operations it is put on a fully parenthetized prefixed notation.

LISP gives a full satisfaction to this definition, thus M3L can be regarded as a LISP-machine. Today, several studies on LISP-machines have been made, in software [10], in a microprogrammed way on the B1700 [11], or with microprocessors [12]. However, the project which is under study at MIT [13] is the most important one. This shows how attractive architectures based on λ-languages are. We will refer to them as λ-architectures [14].

Nevertheless, these plans are only dealing with LISP and its own concepts. As for the hardware realizations, the host processor is not necessarily a specialized machine ; sometimes it is equipped with hardware tools which are valuable but too much oriented towards LISP concepts. Rather than discussing the semantic features of LISP our aim was to show the common characteristics of every 3L forms and give them a direct expression through the hardware.

3- THE GENERAL ARCHITECTURE OF M3L

The M3L project was initiated in september 1977 with a systematic study of LISP interpretation. First, we defined a pseudo-machine and then we built a simulator on which a microprogrammed LISP interpreter was written. The simulation measures [15] prompted a new design for the prototype presently in phase of achievement. These measures showed how important some resources are, more precisely the pair-cells memory. From a functional point of view, the recursivity mechanism is the most critical. We shall hence develop these two points more lately.

1. The synoptic of M3L

The general organization of M3L is very simple. The resources (memories, registers, operators) are bussed on a single communication path, using very conventional tri-state connections.

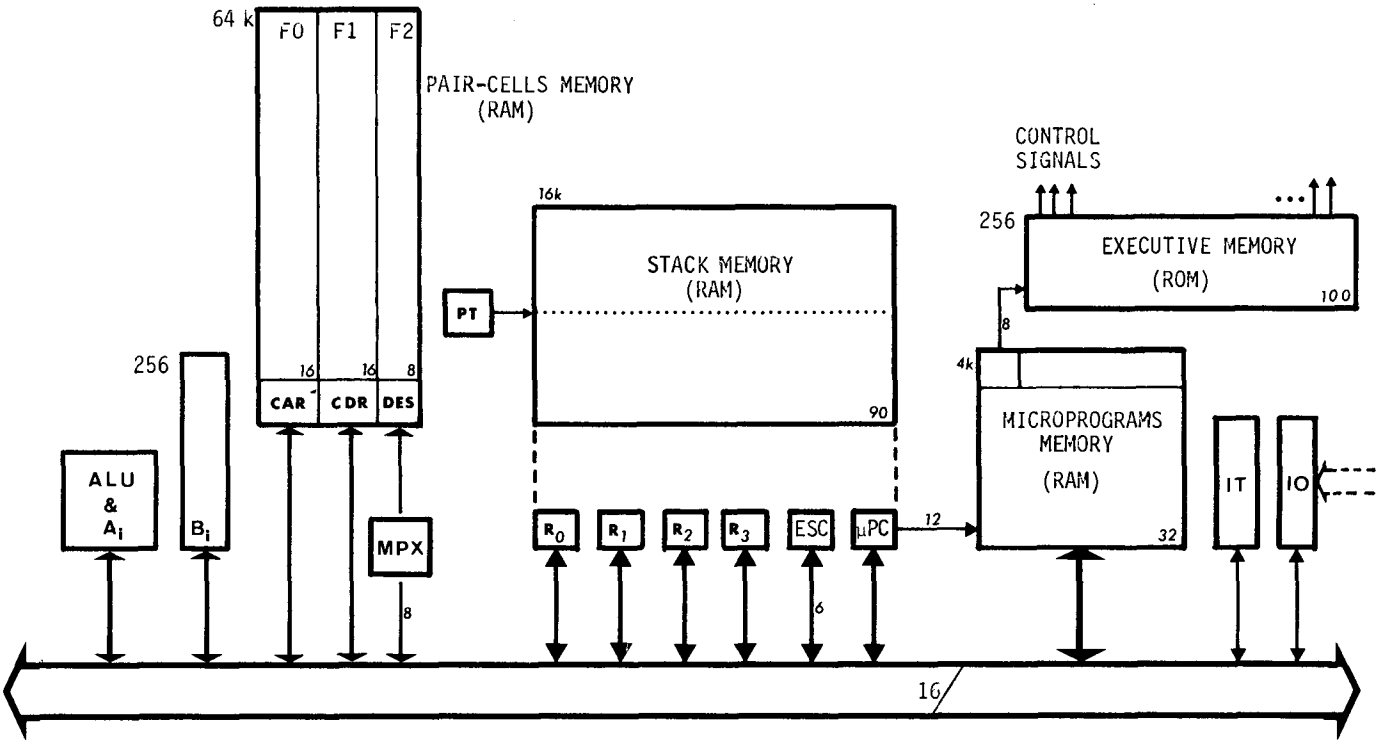


Figure 2 : ARCHITECTURE OF M3L

The figure 2 shows the synoptic of the M3L prototype with the parameters which were retained in order to validate the 3L-model.

The datapath is 16-bit wide, which corresponds to the maximal size of the pair-cells memory (64 K). The numerical processing is carried out by a bit-slice ALU which is relatively powerful (AMD 2903). A little arithmetical processor (AMD 9511), connected as a peripheral, allows the direct achievement, in hardware, of the more complex algebrical functions. Inputs/outputs are also managed by specialized chips of the AMD 2900 family. Once again we find the conventional functions of the Von Neumann architecture but, here, they take a marginal place.

The resources of M3L consist only in registers and constants. Registers are divided into four categories :

- A_i registers $i \in [0,15]$
  They are used for current tasks and information transfer between the microprocedures
- B_i registers $i \in [0,255]$
  Theyserve as global registers for every microprocedure, they contain the descriptors of the currently emulated system
- T_i registers $i \in [0,31]$
  Theyare flip-flops which give the current status of the system. They are global resources and some of them can be set or reset by the microprogrammer
- R_i registers $i \in [0,3]$
  Theymake the use of recursivity possible owing to their locality.

The table 1 shows the static occurrences of the registers in the LISP microinterpreter.

Registers \ i	0	1	2	3	4	5	6	7	8
A	265	37	9	8	6	0	0	0	0
B	49	6	2	0	0	0	0	0	0
T	0	0	0	0	0	0	0	0	0
R	141	68	26	7	0	0	0	0	0

Table 1 : Registers occurrences

In our LISP interpreter, only 5 A_i registers were necessary for supporting the parameter parsing. The small number of B_i registers resulted from the fact that, in the simulating environment, the parameters of the emulated system were quite non-existent. As LISP mainly deals with non-numerical processing and as the escape concept was intensively used for writting the interpreter (see section 5.1) T_i registers were put aside. Four R_i registers only were necessary to support the local environment of the microprocedures. We shall see that this consideration had an important impact on the stack memory organization.

3. The micro-control

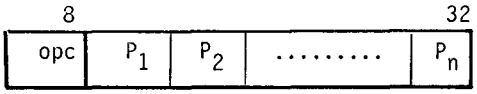
The microprograms, written in LEM, are compiled to produce fixed microcode. The great diversity of control signals to provide (in particular to control the tri-state bus) has led to a two-level microprogramming. Here, the used technique is different from the microprogramming with a real second level referred to as nanoprogramming. To execute a microinstruction through the datapath one must :

1. provide some parameters :
   - number of A_i , B_i ...
   - long, short constant
   - number of branch function, of ALU function...
2. define an action to be executed, i.e. state a particular data transfer through the datapath.

The first level of microprogramming, which corresponds to the first part, is vertical. Thus, the effort of the LEM compiler is less important and the size of the microinstructions can be shortened. In addition, this reduces the amount of microcode to be swapped during control switches.

The second part, fixed for a given action, still requires much more bits for the direct control of the gates. The repetition of such a long dead-bit sequence is cumbersome. Thus, the action to be executed is specified by the second level of microprogramming, in a single horizontal word where each bit directly controls the gates. It is called the executive.

The format of a microinstruction is



opc represents the code number of an executive and the P_i's are the arguments. The size of microinstructions is 32 bits. To the operation code (opc) can correspond up to 256 executives. Theoretically, a great number of executives can be defined, but practically, the facilities of a datapath are never completely put on use : our simulation of a LISP system required 60 executives only. The executives reside in fast PROM memory (t_A = 50 ns) with 256 words of 100-bit length.

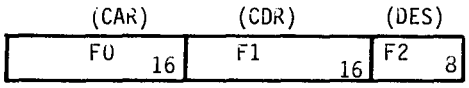
The cycle time of the microinstructions is fixed to 500 ns. It may seem to be long for a modern technology but with regard to the power of the microinstructions it is a good speed : the cycle starts with the microinstruction fetch (100 ns). Then, it includes some registers moves and always a main control phase (200 ns). As the case may be this phase performs:

- an access to the pair-cells memory
- an arithmetical operation on the ALU
- a context switch with an access to the stack memory
- a refresh cycle

Hence, the microinstruction cycle includes the access to the different memories. This results from the fact that the simulation has shown that most of the accesses to the pair-cells memory are performed, on an average, each three microinstructions. Thus, M3L can be viewed as a memory-to-memory architecture.

4 - THE PAIR-CELLS MEMORY

The pair-cells memory is the main memory of M3L. It contains 64 K of identical cells with the following format :



In the M3L prototype, the size of F0 (in LISP : CAR) and F1 (in LISP : CDR) fields is equal to the datapath size (16 bits). The field F2 (a byte) serves as a descriptor of the pair. These fields can be accessed to independently in the read/write mode. Thus, we have the equivalence between :

the LISP functions	the LEM microinstructions
(SETQ X (CAR 'Y))	X ← F0(Y)
(RPLACA 'X 'Y)	F0(X) ← Y
	(X and Y are M3L registers)

The measurement of the memory accesses has shown that a CAR access is often followed by a DES and then a CDR one. This corresponds to the "decoding" of the cell. Therefore, the read access to the different parts of the cells has been thoroughly elaborated. This fact is emphasized by the addition of the FETCH microinstruction :

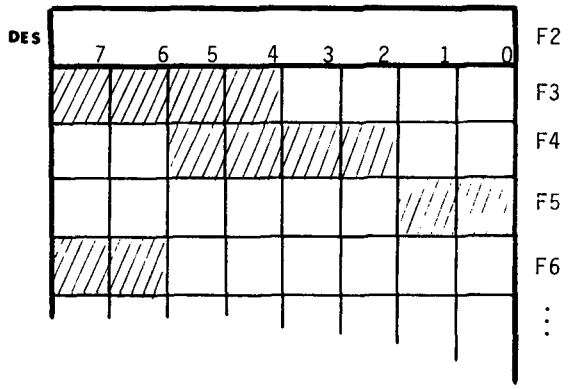
FETCH X INTO Y AND Z ∿
- Y ← F0(X)
- Z ← F1(X)
- DES ← F2(X)

DES is a pseudo-register devoted to the descriptor field.

At the interpretative step, this microinstruction, similar to the fetch-operation of the Von Neumann systems, really speeds up the decoding operations and the passage of arguments to the microinterpreter procedures. It permits 31 percent of the memory accesses to be avoided.

The pair-cells memory is block organized. The three blocks correspond to the CAR, CDR and DES fields respectively. As CAR and CDR blocks are 64 K words of 16 bits, it is clearly implied that the pointers deal with real addresses. As a matter of fact, we have implemented no CDR-coding mechanism because we think that the profit in memory space should be lower than 10 percent of the prototype cost, and it does not justify the complexity introduced by its management. As we wished to validate the 3L-model rapidly, the prototype only features a 16-bit addressing and a relatively small main memory. Presently, a new version of M3L with a 24-bit addressing is studied.

The DES block is 64 K words of 8-bits. Logically, each F2 field can be divided into contiguous or superposed sub-fields (F3,F4 ...). They can also be accessed to independently in the read/write mode.



The access functions to the pair-cells are generalized to :

$$X \leftarrow F_i(Y)_{i=0,n} \quad \text{or} \quad F_i(X)_{i=0,n} \leftarrow Y$$

The slicing capability of the descriptor byte results in a good optimization of descriptors encoding. It allows a direct access to any pair-cell field and yields a good output environment for the editor (DES encoding) and a good input environment for the interpreter (DES decoding) because, in these two steps, there are no shift, compacting or decompacting to perform.

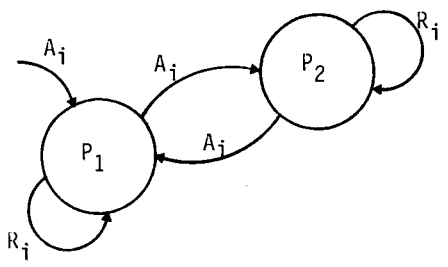
The access functions to the DES block are achieved in hardware by a gadget (MPX) whose basic principles are developed in [8]. It performs parallel justifications on the DES sub-fieds in a purely combinatory way. The sub-fields are defined by a mask sourced from the executives. We said above that such tools are unaesthetic but, as the operation field is only 8-bit wide, it can be achieved very easily and it offers an access to the DES sub-fields as fast as that of the F0 and F1 fields.

The technology used for the pair-cells memory is the dynamic-MOS RAM (TMS 4116). Its fast access time (t_A = 150 ns) has enabled the inclusion of the read/write cycle in the microinstruction cycle. The drawback of dynamic memory is that additional refresh operations are required. In M3L, they are mainly performed during register-to-register moves. Note that new technologies will eliminate this inconvenience rapidly and will strenghten the trend to directly access the main memory in the microprogramming environment.

5 - THE STACK MEMORY

1. The recursivity in the LEM language

LEM is a recursive language, this is absolutely necessary in tree processing. A LEM module is composed of little procedures which are independent and not ordered. They can refer to each other and even to themselves. In control switching from a microprocedure to another, the A_i global registers are used for parameters passing and the R_i local registers are automatically saved.



P_1 picks its input arguments into the A_i registers and outputs its results to P_2 via A_i's. The object of the R_i registers is to maintain the value of A_i registers in the environment of P_1. Thus, their value cannot be erased by the application of P_2.

To the recursivity an automatic escape mechanism was added. Writting top/down recursive parsers requires such tools. They are similar to the software interrupts (like ON conditions of PL/1). Escapes and recursivity are two concepts which are closely related, hence they were merged in order to offer a better systematization of the control transfer between the microprocedures. It is thus stated that in LEM:

- calls are recursive
- returns are escapes

2. The stack memory management

The object of the recursivity stack is to save the context of the calling microprocedure during a CALL instruction and to restore it on the corresponding return instruction. A context comprises mainly : the local registers (R_i), the microprogram counter (µPC) and the escape tag (ESC).

We first thought that the size of the context would be important because of the number of local registers required for tree processing. Therefore, our preliminary architecture included a complex memory stack processor for the management of CALLs and ESCAPEs. It consisted in a disk file, which simulated the virtual stack memory, and a circular queue which simulated the real stack memory and was implemented in a fast memory. In order to save a maximum of memory space, only the used R_i registers were saved when a switch occurred.

Fortunately this "gas-works" processor was only achieved in software. We were surprised when the first LISP interpreter was written. As a matter of fact, only four R_i registers were required (see table 1). The reason is that in tree processing the use of the recursivity results in very concise contextual information : generally, the current work (A_i) is done on the CAR and we only have to save the CDR (see also the section 6.1). This fact was confirmed by the dynamic measures ; the average number of pushed R_i registers is 1.3.

Finally, we chose the simplest solution. The number of R_i registers is stated to four and the context is located in a single pseudo register where $R_{i=0,3}$, µPC, ESC can be accessed to. Therefore, the status register is 90-bit wide. During CALL and ESCAPE operations it is simply stored into (or restored from) the stack memory. In one microinstruction cycle the micro-context is pushed or popped.

The stack memory consists of 16 K words which are 90-bit wide. This allows the performing of 16 000 recursive CALL successively. Our dynamic measures, upon little LISP programs, have shown that the number of contexts increases rapidly and then seems to become stable after 100. However, because of the low cost and the fast access time of the dynamic-MOS memory we used the same technology in the pair-cells memory and in the stack memory, thus dictating the 16 K size. Because of tail-recursivities elimination and of the small size of the pair-cells memory, the stack is certainly too large for this version of M3L. This fact will have to be confirmed in further measurements.

3. The CALL microinstruction

The CALL mechanism of M3L is illustrated in the figure 3.

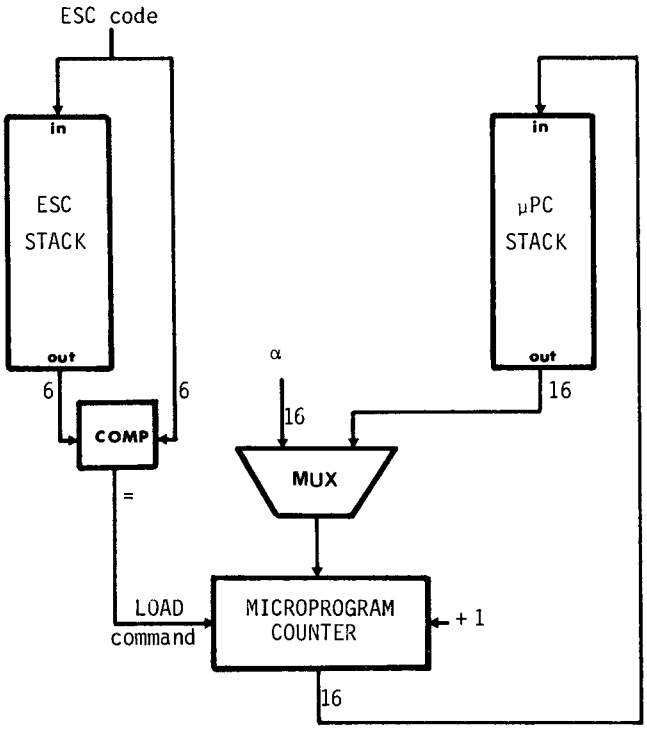
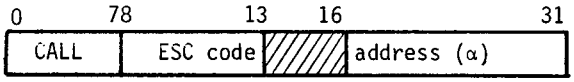


Figure 3 : THE MICRO-CONTROL UNIT

The call microinstruction has the following format :



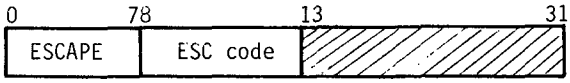
when a call microinstruction is executed, both the escape number (passed as argument) and the return address (µPC) are saved and a branch operation is done according to the address specified as second argument (α). The execution goes along with three phases :

- return address computation : incrementation of μPC
- context saving : ESC code ESC stack ;
  μPC → μPC stack
- initialisation of μPC : α → μPC

which correspond to two register moves and an access cycle to the stack memory.

4. The escape microinstruction

The object of the escape microinstruction is to make a branch to the last microprocedure of the current process which has previously set up the same escape code. Its format is :



The execution cycle is divided into two phases :

- fetch of the preceding context from the stack memory
- if the ESC code, which is passed as argument, corresponds to the restored ESC code then initialize μPC with the return address.

Notice that with this algorithm, the same escape microinstruction is performed as many times as the ESC code of the microinstruction is different from the fetched one. This process enables the stack to be scanned until the searched context is found.

6 - EXAMPLES OF MICROPROGRAMMING

1. Interpretation of a HLL control construct

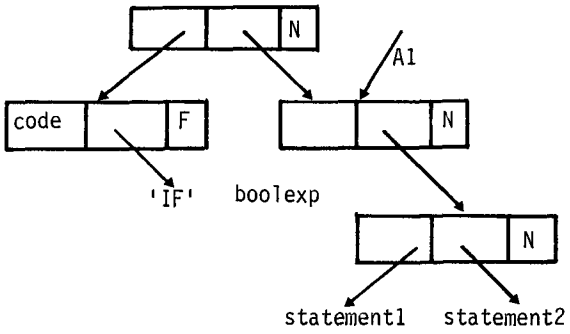
When the 3L-model is applied to a HLL language, the first task is to design the associated 3L form. For example, let us consider the following HLL control construct :

if boolexp then statement1 else statement2

The corresponding 3L form might be :

(IF boolexp statement1 . statement2)

which gives, for example, the internal form :



N : node

F : function

When it is called, the IF procedure which interprets the 'if' function receives the pointer to its argument list in the A1 register. In LEM :

```
procedure if ;
begin
fetch A1 into A1 and R1 ; % to save the CDR in R1 %
call exec ;
if A1 # 'nil' then A1 ← F0 (R1) % CAR %
              else A1 ← F1 (R1);% CDR %
call exec ;
escape
end
```

- EXEC : is the general evaluator of the interpreter. It picks in A1 the pointer to the sub-tree it has to evaluate, and returns it in A1 again to the result
- checking of boolexp is done like in LISP.

Seven microinstructions only are required for the fixed representation of this microprocedure. On M3L, its execution time is 3 μs to which we must add the evaluation time corresponding to the two calls of exec. They strictly depend on the complexity of the sub-trees to be evaluated.

2. The interpretation of LISP

Presently, a little LISP interpreter is operational on the simulator of the M3L prototype. This interpreter requires 670 32-bit microinstructions. It includes the 21 principal standard functions of LISP.

. The EVAL procedure : the role of this procedure, in our LISP interpreter, is to decode the objects and pass the argument list to the functions. When it is applied to the different kinds of LISP objects its response-time is :

numbers : 2 μs
atoms : 2.5 μs
functions : 6.5 μs
quote : 5 μs

. The standard functions : the following table gives the response-time for the typical LISP functions. This time does not include the inner EVAL sub-calls.

LISP function	μs
CAR	2
CDR	2
CONS	10
QUOTE	1
RPLACA	4

LISP function	μs
RPLACD	4
LISTP	3
ATOM	3
NULL	2.5
ENDLISP	0.5

The measurement of some standard functions is dependent from the environment in which they are executed:

EQ	if equal and non-num.	5 µs
	if non-equal and 1th non-num.	5,5 µs
	if non-equal and 2th non-num.	6,5 µs
	if non-equal and non-num.	9 µs
	if equal and num.	9 µs
GT	starting with false	5,5 µs
	false at the pth level	4+[(p-1)*5]µs
	true with n arguments	n*5 µs
ADD1	if numerical	10,5 µs
	else	2,5 µs
SETQ	with n affectations	3+[(n-1)*4]µs
IF	with selection true	3 µs
	with selection false and n statements in the ELSE clause	4,5+2+((n-1)*3,5)µs
COND	without clause	1,5 µs
	with n clauses whose pth is selected and involves the execution of n S-expressions	2,5+((7+5)*(p-1)) +7+((n-1)*7)µs

These times do not include the inner EVAL sub-calls, because their value depend on the complexity of the sub-trees which are evaluated. Here are three complete examples :

The execution of :	requires :
(SETQ X (CONS (CAR X)(CDR X)))	48 µs
(RPLACA 'X (CAR 'X))	29 µs
(IF (EQ X 'A)(SETQ X 'B)(SETQ X 'A))	X = 'A : 43 µs X ≠ 'A : 50.5 µs

In the three examples studied, advantages of about 14 times speed improvement are realized over the CII-HB LISP which is running on an IRIS 80 computer.

CONCLUSION

The architecture of M3L is not spectacular.This machine is equipped with no black boxes performing super-functions, nor complicated processors.It quite consists of memories ; the pair-cells memory and the stack memory deal with 70 percent of the prototype chips. However, we think that this architecture is innovative and that its efficiency in list processing is great. The reason for it is that the top/down strategy used here has resulted in a good distribution of the emulation functions at every step of the model : 3L form, LEM, hardware structure.

Rather than developing a LISP-machine our aim was to investigate emulation in a more general context. Today, the LEM micro-software for the interpretation of PASCAL is in the phase of achievement.Following the LISP example, we focus in edition processing so as to improve interactivity. Thus, the simplicity and the versatility of M3L lead this machine towards the range of interactive applications. In this area, M3L and in a more general fashion λ-architectures can improve highly the man/machine communication.

ACKNOWLEDGEMENTS

This work was done at Paul Sabatier University in the laboratory of Professor R. BEAUFILS who encouraged our research in this way and is sponsored by the french IRIA under grant # 79-027 for the achievement and evaluation of a prototype offering PASCAL and LISP capabilities.

REFERENCES

[1] Y. CHU
Direct-execution computer architecture
IFIP Congress - Montreal - 1977

[2] L.W. HOEVEL
"IDEAL" Directly Executable Languages. An analytical argument for emulation
IEEE Trans.on Computer - Vol.C-23 n°8 - 1974

[3] J.P. SCHOELLKOPF
A tutorial on high level language machine for PASCAL
ENS-IMAG Report 131 - Grenoble - Oct.1978

[4] W.T. WILNER
Design of the Burroughs B1700
FJCC AFIPS Montvale,New Jersey - 1972

[5] G.J. MYERS
Advances in Computer Architecture
John Wiley & Sons Interscience Pub. 1978

[6] W.T. WILNER
B1700 Memory Utilization
FJCC AFIPS Montvale, New Jersey - 1972

[7] E.J. ORGANICK, J.A. HINDS
Interpreting machines : Architecture and programming of the B1700/B1800 series
Elsevier North Holland - 1978

[8] D.LITAIZE, B.LECUSSAN, J.P.SANSONNET, J.PETIT
An efficient hardware tool for bit pattern manipulation
EUROMICRO Congress - Venise 1976

[9] J. Mc CARTHY
LISP 1.5 Programmer's Manual
MIT Press - Cambridge - 1962

[10] A. LUX
Etude d'un modèle abstrait pour une machine LISP et de son implantation
Thèse de 3ème Cycle - USMG - Mars 1975

[11] M.L. GRISS, M.R. SWANSON
A microprogrammed LISP-machine for the Burroughs B1726 SIGMICRO NEWSLETTER Vol.8 n°3 - 1977

[12] G.L. STEELE Jr., G.J. SUSSMAN
Design of LISP-based processors or,...
MIT AI MEMO n°514 - March 1979

[13] A. BAWDEN, R. GREENBLATT, J. HOLOWAY
LISP machine progress report
MIT Report n°444 - August 1977

[14] E.J. ORGANICK
New directions in computer systems architecture
EUROMICRO JOURNAL - September 1979

[15] J.P. SANSONNET, M. CASTAN
Un exemple d'émulateur : M3L
Progress Report LSI # 131 Toulouse - June 1978