DIALISP - A LISP MACHINE

G.Stefan
A.Paun
V.Bistriceanu
A.Birnbaum

Functional Electronics Laboratory
Polytechnical Institute of Bucharest
ROMANIA

A B S T R A C T

High performance facilities to interpret LISP represent an ever increasing request even for minis.

This paper presents a LISP hardware structure conceived to be implemented in a general purpose mini system called DIAGRAM.

The LISP structure had to be adapted to the system technological requirements and size.

The data structure and the instruction set concerning the basic machine are also presented.

1.Introduction

The system comprising the LISP machine is shown in Fig.1, where:

- IOM is a microcomputer controlling the system input-output devices;
- MPM1 is a minicomputer on a PCB, operating as the system central unit, running high level languages (Fortran, Basic, a.s.o.). It operates with a general purpose arithmetic processor;
- MPM0 is a physical structure identical with MPM1, controlling the alphanumeric and graphic display on a black and white or color CRT monitor. It can operate with a numerical processor specialized in bi- and tridimensional graphic transformations;
- DIALISP, the topic of this paper, is the LISP hardware interpreter.

2.General structure of LISP Hardware Interpreter (DIALISP)

2.1.Structural Options

The acces time of the available memory devices used in high capacity memory ar - rays is 300-500 ns.

Using TTL devices, processing structures (RALU, CROM,...) having cycle time between 150 and 300 ns can be obtained.

Hence, using a cache memory the effici - ency may be rather poor even when the processes associated to the LISP interpreta - tion frequently access the memory.

Hardware facilities offered by bit-slices controlled by a stack state-machine (SSM) have been used to optimize DIALISP cost and size. Thus the whole structure is built on a single PCB, but the operating speed is limited due to the small number of fast registers and to the use of arithmetic comparing functions instead of logic ones, as in a dedicated structure.

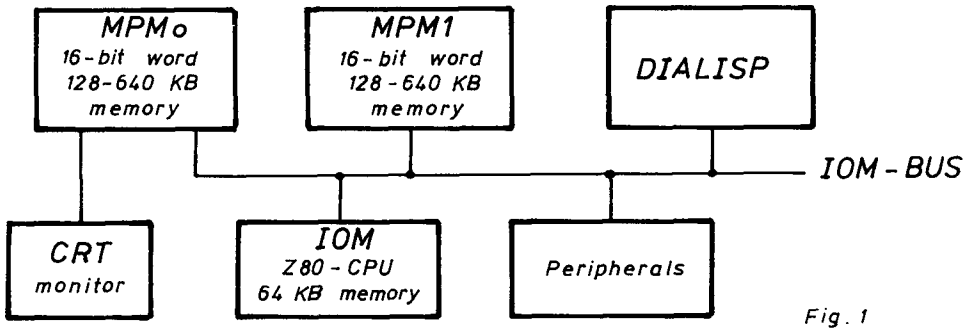

Fig. 1

Using a SSM to control the structure instead of a typical CROM configuration permits higher speed and the implementation of a large micro-stack.

2.2.Duality

An important option for DIALISP takes

account of the following facts:

- the memory access time is approximately two times larger than the execution cycle time;
- the memory accesses are frequent, re - quiring waiting states for the data proce - ssing unit;
- LISP implementation may imply parallel processes to the main evaluation one (e.g. GC, stack memory, tree structure functions etc.).

Consequently, a physical configuration a- llowing parallel running of two processes has been conceived, so that the speed of either process is only slightly affected.

The waveforms of Fig.2 represent the time sharing of the physical resources between both processes.

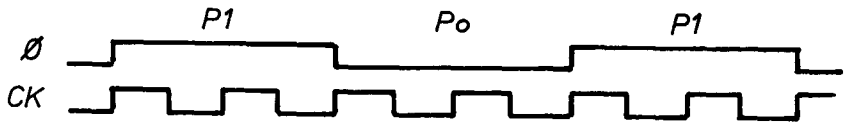

Fig. 2

The clock cycle is 250 ns.

When Ø = 0 the PO process uses the pro - cessing unit (RALU and SSM), and the P1 process reads its own memory M1.

When Ø = 1 the P1 process uses the pro - cessing unit, and Po reads Mo.

2.3.The Stack State-Machine (SSM)

The SSM block-diagram is shown in Fig.4, where:

- SM (Stack Memory) closes the state- -machine loop along with the LATCH;
- SPo and SP1 are counters used as stack pointers for Po and P1;
- MUXS selects the stack pointer (SPo or SP1) according to Ø;
- MUXF multiplexes the flag that deter - mines the state-machine transition.

The state-machine can thus control two independent processes switching to Po or P1, relative to Ø. The trace of either process is stored in the SM location indi- cated by SPo and SP1.

LATCH+SM form a master-slave structure with a master (LATCH) and several slaves (SM). The current slave is indicated by SPo or SP1. The succesive number or CALLs may thus significantly increase.

2.4.Functional Distribution in DIALISP

The DIALISP structure of Fig.3 is equi- valent to two 32 bits minicomputers, acce- sing o.25 MB of memory each. The two structures may operate independently; they communicate by interrupts and some inter - nal registers of RALU.

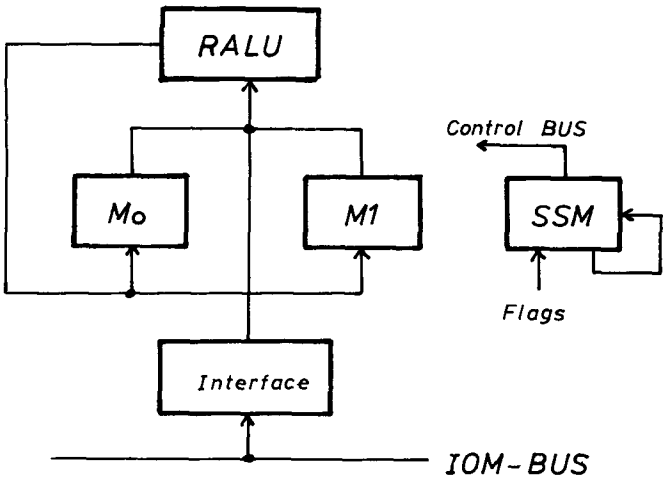

Fig. 3

During one memory access time two RALU microoperations are performed, allowing the preparation of a new memory cycle or the end of a read-modify-write cycle.

Therefore, two processes may be run by the structure: the access to the RALU and SSM is made while the other process waits during the memory access time.

Under these conditions, the DIALISP block diagram is that of Fig.3. The RALU is implemented with 16 Intel 3oo2 ICs; Mo and M1 have 256 KB each and use Intel 2164-25 chips. The RALU registers are shared by both processes.

Several ways of task distribution bet - ween Po and P1 are considered in interpre- ting LISP.

A possible solution, now implemented, is when Po is the main evaluation process and P1 is a large stack with very short access time. P1 ensures the stack extension, if necessary, over 256 KB, on the external magnetic memory.

This solution has the disadvantage of the unbalanced use of the structure especially when P1 controls a common stack.

Another more balanced solution is when P1 is a complex stack.

Because Po and P1 processes are not al - ways symetrical a hardware facility allows "cycle stealing" when Po or P1 is time dominant.

The present implementation uses 1 bit for garbage collector (GC) and the other 7 for identifying up to 128 data types.

Hardware data types (a subset in the current version) are shown in Fig.5.

There are only four data types, but later they will be extended with vectors, arrays, long integers, short and long floating-point numbers.

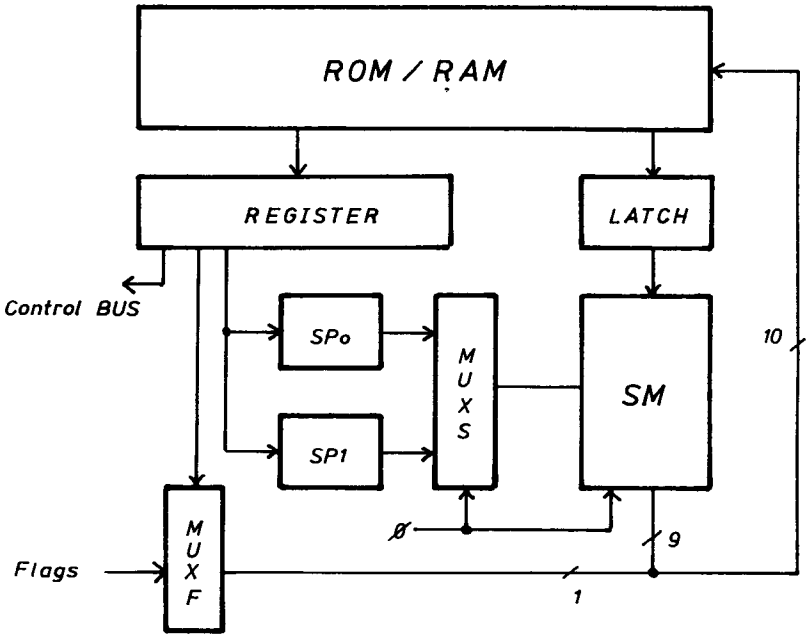

Fig. 4

Such an example is evaluating a LISP va - riable when Po is working searching on the stack while Po is waiting the value. Thus P1 is "stealing" Po's time which otherwise would be lost.

3.Data Structures and Address Space

The standard word structure consists of 32 bits:

- 1B (8 bits) used as tag;
- 3B (24 bits) used for addresses and for short integers.

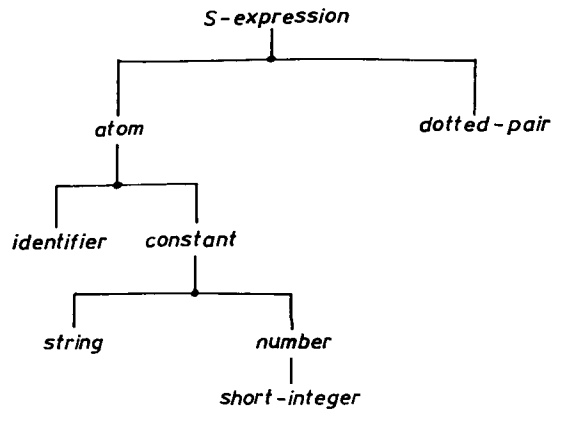

Fig. 5

4.The Instruction Set

Software is three-level structured as follows:

1.microprogramming level;
2.assembly level;
3.LISP level.

4.1.Microprogramming Level

Taking into account the position of the present machine in the range of LISP ma - chines, a 1Kx72 bit ROM has been chosen;it contains aproximately 80 microprogrammed functions. They are:

a) initializing functions
b) I/O functions
c) pure LISP functions
d) stack functions
e) all assembly functions many of them being supported by b), c) and d).

A micro-translator has been written in LISP on a PDP 11/45; it generates microcode in three passes. The micro-translator may be used later, on the present machine for new microprogrammed functions.

The address space is 16 Mwords, the memory being extended on a cartridge disk. The virtual space is logically divided into three subspaces:

- binary code space (BS), used for a - ssembled functions;
- cell space (CS).

In the version now implemented Po process

is driving BS and CS and P1 the stack space.

4.2.Assembly Level

On this level there are those functions belonging to binary space (BS), which implement other useful LISP functions (hand-coded or compiled ones). READ, EVAL, PRINT and most of LISP system functions are hand-coded using the assembler. The assembly language instructions have 8 bit op-code and o up to 4 24-bit operands.

The operands may be constants, registers (up to five Ro-R4) or jump addresses.

Because the assembler-code provides the control of evaluation choosing a good instruction set improves LISP speed.

Thus we have considered two methods:

a) DIALISP is register-like oriented
b) DIALISP is stack-like.

The first option results in a small set of pure LISP functions which must be micro-coded and hence an economical use of control memory. However this solution is slower than b) regarding evaluation time.

Option b) is dealing with an asymetrical and redundand set of instructions. Most of them are stack-oriented hence there are less pure microprogrammed LISP functions.

On the other hand some of LISP functions such CAR, CDR, C...R send explicitly their values on the stack but others like CONS, NCONS implicitly.

Type cheking is done on the stack or using LISP values which have not been sent yet to their destination slot.

Method b) does not allow a good use of the small microprogrammed space but is a fast evaluating one.

There are made the following assumptions regarding the assembly instructions:

<op> - denotes the mnemonic of the instruction
<Rj> - denotes any register ($o \leq j \leq 4$)
<addr> - denotes any jump address in the range $-2^{24}+1$; $2^{24}-1$
<dec> - denotes any decimal constant in the range $-2^{15}+1$; $2^{15}-1$
#<hex> - denotes any hexadecimal constant in the range $-2^{15}+1$; $2^{15}-1$
@<const> - denotes any LISP constant like NIL, QUOTE,...
<offset> - denotes any slot from the top of the stack.

4.2.1.Basic instructions. Method a)

The register-like machine instruction set consists of five classes o to 4.

Class o includes instructions without operands:

<op>

Such functions are: TERPRI, SAV, RES (saves/restores stack pointers) etc.

Class 1 includes instructions like:

<op> <Rj>
<op> <addr>
<op> <dec>
<op> #<hex>
<op> @<const>

Such functions are: stack ones (PUSHN , PUSHV, POPV, RET) or control ones like JUMP input/output (OPENI, OPENO, PATOM) and finally error-handling primitives (FATALT, ERRDT).

Class 2 includes instructions like:

<op> <Rj>, <Ri>
<op> @<const>,<Rj>
<op> <Rj>, <addr>

Such functions are: LISP functions (CAR, CDR, C...R), transfer ones (MOVE), type cheking (JLIS, JATOM, JNSTR).

Class 3 includes instructions like:

<op> <Rj> , <Ri> , <Rk>
<op> <Rj> , @<const>, <Rk>
<op> @<const>, <Ri> , <Rk>
<op> @<const>, @<const>, <Rk>
<op> <Rj> , <Ri> , <addr>
<op> @<const>, <Ri> , <addr>
<op> <Rj> , @<const>, <addr>.

Examples: LISP functions (ASSOC, RPLACA, RPLACD, CONS, EQ) and control ones like JNEQ.

Class 4 includes instructions like

<op> <Rj> , <Ri> , <Rk> , <Rl>.

Such functions are: GET, PAIRLIS.

4.2.2.Basic instructions. Method b)

The stack-like machine instructions set consist of four classes o to 3.

There are many LISP functions without explicit operands (the arguments being on top of the stack) and there are also type-cheking ones with explicit operands (they are considered by the offset from the top of the stack) or without operands (they are usually LISP function values being deposited in an internal register reserved for this special purpose).

Class o includes operations like:

<op>

Such operations are: LISP functions (CAR, CDR, RPLACA, RPLACD, C...R etc.), stack functions (LOOK, SEND, NEXT, SAVE-ED, LINK-D, RET, RETS, SEND etc.).

Class 1 includes operations like:

<op> <offset>
<op> <addr>
<op> @<const>
<op> #<hex>.

Such operations are: stack functions (SEND, CALL, CALLI), LISP functions (CAR, CDR, C...R, SET-CAR, SET-CDR, SET-C...R). control functions (JNREC), type cheking ones(JLIS, JATOM, JNSTR, JNNUM) or error handling (FATALT, ERRDT etc.).

Class 2 includes operations like:

<op> <offset>, <offset>
<op> <dec> , @<const>
<op> <offset>, <addr>.

Such operations are: stack functions (ALLOCN), LISP functions (RPLACA, RPLACD, CONS), type-cheking ones (JLIS, JATOM, JNSTR, JNNUM) etc.

Class 3 includes operations like:

<op> <offset>, <offset>, <addr>
<op> <offset>, @<const>, <addr>
<op> @<const>, <offset>, <addr>.

Examples: RPLACA, RPLACD, CONS etc.

The assembler is one-pass and has also been implemented in LISP on a PDP 11/45.

5.Estimating performances

On a machine with a little main memory (2x64 Kw) we had to provide compatibility with MacLisp since there are many AI applications using this LISP dialect.

Our measurments suggest the use of a larger main memory reducing the paging overhead.

Data structures do not implement CDR-coding; our machine is based upon a simple design to be carried out with minimum of microcode.

On the other hand DIALISP has a small number of fast register (3002's) so our stack-like design (method b)) is better than register-like one.

The conclusions listed above are showing the same problems as with Alto's design.

Finally DIALISP is using Baker's rerooting scheme for method a) and a deep binding one for method b).

The dual processor was good idea for speeding LISP evaluation with a full use of hardware resources.

The table below shows some execution times for pure LISP functions:

Function	Execution time microsec.
CAR	4.5
CDR	5
RPLACA	12
RPLACD	14
CAAR	7
CADR	7.5

A variable evaluation takes (10+9*n) microsec. where n represents the depth from the current top context.

Next there is the LAST LISP function written in assembly:

Method a)

Recursion

```
LAST   CDR       R1,        Ro
       JT        Ro,        LST1
       CAR       R1,        Ro
       PUSHV     Ro
       RET       1
LST1   PUSHN     1
       CDR       R1,        R1
       CALL      LAST
       RET       1
```

Iteration

```
LAST   CDR       R1,        Ro
       JT        Ro,        LST1
       CAR       R1,        Ro
       PUSHV     Ro
       RET       1
LST1   CDR       R1,        R1
       JUMP      LAST
```

Method b)

Recursion

```
LAST   CDR
       JNREC     @NIL.      LST1
       CAR
       SEND
       RET
LST1   SAVE-END
       ALLOCN    1,         @LAST
       CDR
       SEND
       LINK-D
       CALL      LAST
       RET
```

Iteration

```
LAST   CDR
       JNREC     @NIL,      LST1
       CAR
       SEND
       RET
LST1   SET-CDR   4
       JUMP      LAST
```

6.Further Development

Two possible development directions are considered by the authors.

Firstly, the machine can be improved starting from the structure presented above, by using more efficient functional blocks, e.g.:

- a purely logic structure having a larger number of registers may be used instead RALU. This would permit a clock cycle shorter than 200 ns;
- the memory is to be implemented on several PCBs, each having 1-4 MB, thus allowing a maximum internal memory capacity of 16 MB;
- a large Stack-Memory (SM) (up to 4 Mlevels) allowing microprogramed recursion
- a larger microprogram memory (at least 8 Kwords).

Secondly a RISC LISP oriented processor can be conceived. This option allow software level to be reduced to:

- RISC implemented level;
- LISP level.

This permits LISP level implementation using a large but not very fast memory (e.g. 64K DRAM).

7.References

1) Henry G. Baker, Jr., "Shallow Binding in LISP 1.5", CACM, Vol.21, No.7, 1978
2) Henry G. Baker, Jr., "List Processing in Real Time on a Serial Computer", CACM, Vol.21, No.4, 1978
3) R.R. Burton et al., "Overview and Status of Dorado LISP" Conf. Record of the 198o Lisp Conference, 1980
4) E. Goto et a., "Design of a Lisp Machine-Flats", Conf.Record of the 1982 Lisp Conference, 1982

5) R. Greenblatt, "The LISP Machine", Working Paper 79, MIT Artificial Intelligence Lab., Camb., Mass., 1979
6) D. Moon, "Maclisp Reference Manual", Revision O., Proj. MAC, MIT, Artificial Intelligence Lab., Camb., Mass., 1974
7) G. Stefan, "Unbalanced Structures in Information Processing Systems", Artificial Intelligence and Robotics, Ed. Academiei 1983, Buch.
8) D. Weinreb et al., "Lisp Machine Manual", MIT Artificial Intelligence Lab., Camb., Mass., 1981