Scptcmbcr 1987

Report No. STAN-CS-87- 1184
Also numbered KSL-87-57

Firmware Approach to Fast Lisp Interpreter

by

H. G. Okuno, N. Osato, and I. Takeuchi

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems **Laboratory**
Report No. KSL 87-57

September 1987

Firmware Approach to Fast Lisp Interpreter

by

Hiroshi G. Okuno, Nobuyasu Osato and Ikuo Takeuchi

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

and

Electrical Communications Laboratories
Nippon Telegraph and Telephone Corporation
3-9-11 Midori-cho, Musashino
Tokyo 180 Japan

To Appear in Proceedings of Twentieth Annual Workshop on Microprogramming (MICRO-20).

Table of Contents

Abstract

The approach to speed up a Lisp interpreter by implementing it in firmware seems promising. A microcoded Lisp interpreter shows good performance for very simple benchmarks, while it often fails to provide good performance for larger benchmarks and applications unless speedup techniques are devised for it. This was the case for the TAO/ELIS system. This paper describes various techniques devised for the TAO/ELIS system in order to speed up the interpreter of the TAO language implemented on the ELIS Lisp machine. The techniques include data type dispatch, variable access, function call and so on. TAO is not only upward compatible with Common Lisp, but also incorporates logic programming, object-oriented programming and Fortran/C-like programming into Lisp programming. TAO also provides concurrent programming and supports multiple users (up to eight users). The TAO interpreter for those programming paradigms is coded fully in microcodes. In spite of rich functionalities, the speed of ***interpreted*** codes of TAO is comparable to that of ***compiled*** codes of commercial Lisp machines. Furthermore, the speeds of the interpreted codes of the same program written in various programming paradigms in TAO does not differ so much. This speed balance is very important for the user.

Another outstanding feature of the TAO/ELIS system is its firmware development environments. Micro Assembler and Linker are written in TAO, which enables the user to use the capability of TAO in microcodes. Since debugging tools are also written in a mini-Lisp, many new tools were developed in parallel to debugging of microcodes. This high level approach to firmware development environments is very important to provide high productivity of development.

1. Introduction

The TAO/ELIS system is the first milestone of the New Unified Environment (NUE) project at NTT Software Laboratories. ELTS [5] is a Lisp machine family; one is a breadboard machine and the other is a VLSI machine [14]. TAO [7, 11, 12, 13] is a superset of Common Lisp and designed as a kernel language for NUE on the ELTS machine. However, TAO is not a simple Lisp system, but a multi-paradigm language which incorporates logic programming, object-oriented programming and FortranK-like programming into Lisp programming.

We consider that Lisp interpreter is essential from the following three points-of-view.

- [Application] Interpretive execution of programs is required by some application programs. For example, many expert system building tools support sophisticated programming environments, while they often lack a rule compiler and execute user-specified Lisp programs interpretedly.
- [Programming Environments] The interpreter is considered as an important component of interactive programming environment such as stepper, editor, tracer, and error break.
- [Debugging tool] One of the best debuggers for Lisp programs is the interpreter. And the interpreter is the easiest and clearest tool for the user.

These are our motivations to design and implement a fast Lisp interpreter with full-fledged facilities. Furthermore, the speed of each programming paradigm should be balanced so that the user can implement his idea naturally by using multiple paradigms which is suitable to his conceptualization of applications.

Our approach to speed up the interpreter is to implement it in microcodes. Microcoded Lisp interpreter shows a good performance for very simple benchmarks, but it often fails to provide a good performance for some benchmarks and applications unless speedup techniques are incorporated into it. This was the case for the TAO/ELIS system and we have been developing various techniques of speedup for several years. In this paper, we discuss various speedup techniques adopted in the TAO/ELIS system, their evaluation and applicability to other systems. In Section 2, the background on the ELIS Lisp machine and the TAO language is presented. Firmware development environments of the TAO/ELIS system is also discussed in this section. They are written in TAO or a mini-Lisp, which raises the expressibility of microcodes as well as gives flexibility and customizability to tools. The bottlenecks of interpreted execution of the Lisp system are presented in Section 3, and their solutions are given in Section 4. In Section 5, the TAO interpreter is evaluated.

2. Background - the TAO/ELIS system

This section gives an overview of the ELIS Lisp machine and the TAO language. Firmware development environments are also discussed.

2.1. Overview of the ELIS Lisp machine

The ELTS family has two types of Lisp machines: breadboard machine and VLSI chip machine. The cycle time of each machine is 200nsec and 180nsec, respectively. VLSI chip is manufactured by 2μm CMOS technology [14]. Both machines are compatible at the level of microcodes. The block diagram of CPU is shown in Fig. 2-1. All data given in this paper are measured on a VLSI ELIS machine. The features of ELIS which influence the design and implementation of TAO are summarized below:

- [Tagged architecture] Pointer is 32-bit wide with 8 bit tag included (Fig. 2-2). Tags are used to specify various data types and speed up the interpreter. Various combinations of tag branches are provided by the ELIS hardware.
- [Hardware stack] ELIS has 32K words stack and three stack pointers. Stack

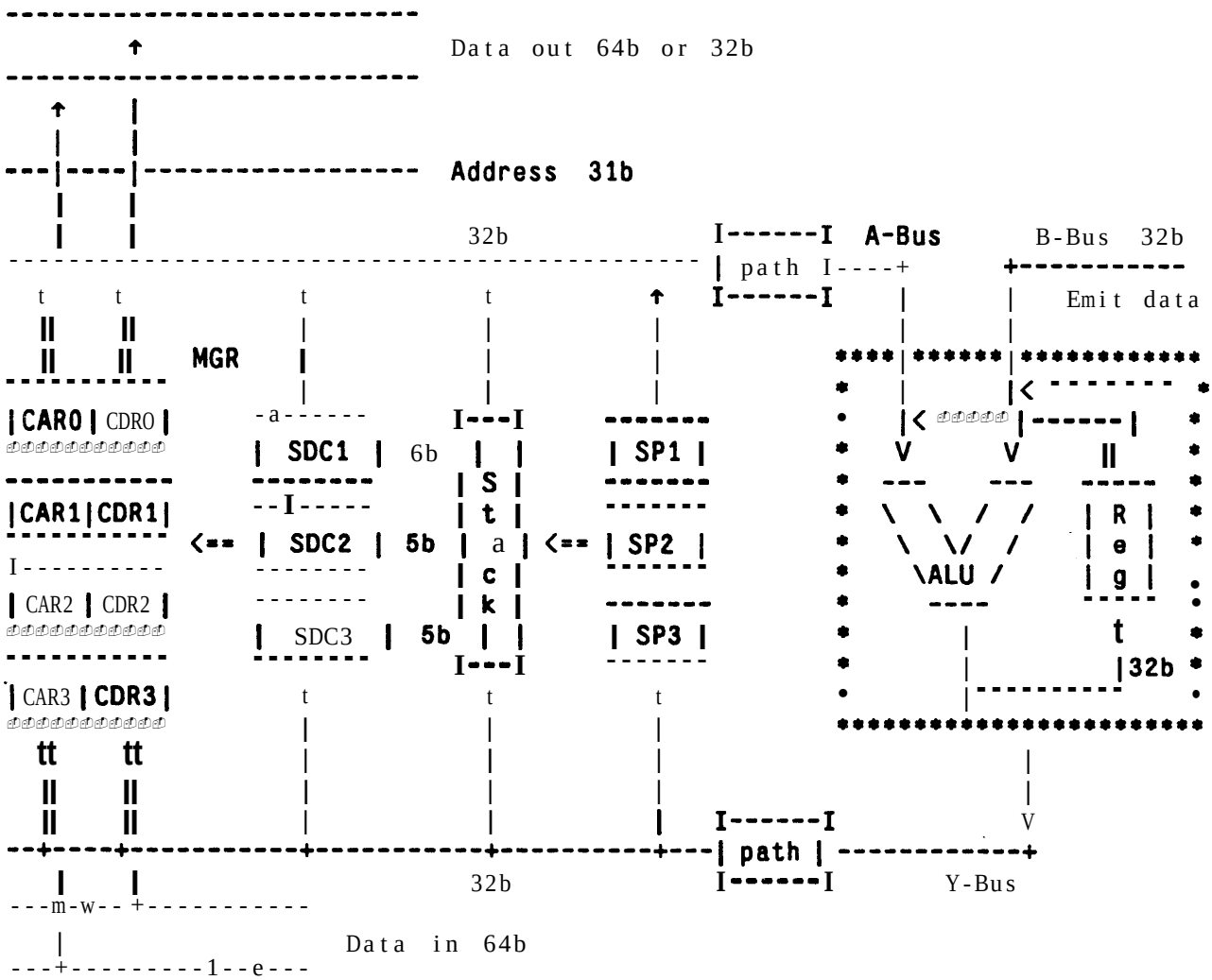
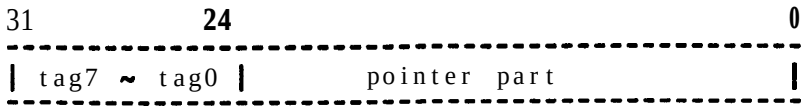


Figure 2-1: Block Diagram of the ELIS CPU



Assignment of tag bits in TAO
tag7 -- for garbage collector
tag6 -- auxiliary use
tag5 -- atomic data if 0, non-atomic data otherwise
tag5 ~ tag0 --- data type

Figure 2-2: Structure of Lisp pointer

overflow and underflow are checked by hardware and if such an overflow occurs, a bit of processor status word (PSW) is set. However, microcode should check the overflow by testing the bit. There is no hardwired interrupt. **Stack** operation is performed in one microcycle.

- [Large **Writable** Control Store (WCS) for microprogramming] The capacity of **Writable** Control Store is 64K 64-bit words so that the TAO interpreter and most of system functions are coded in microcodes. For example, some primitives for EMACS-like editor, **TCP/IP** software and Japanese text processing are coded in microcodes.
- [Memory General Registers (MGR)] Four sets of 64-bit memory interface registers called Memory General Register (MGR) are provided with three index registers called Source Destination Counter (SDC) which points to any byte of MGR. Car and cdr field of each MGR can be used **as** a memory address register or memory data register. They also can be accessed by ALU as a source or destination operand. Note that a **64-bit** word (one Lisp cell) can be read or written between MGR and memory. MGR with SDC can be used as byte manipulation buffers (for strings, compiled codes, etc.)
- [Hardware check of memory access] If a memory operation is initiated to an illegal pointer, that is, a memory address register (say, car or cdr field of some MGR) points to a non-CAR-CDR-able address, the memory operation will be aborted automatically. Tag-5 of a pointer specifies whether the pointer is CAR-CDR-able or **not (see** Fig. 2-2). Therefore, a memory operation can be initiated without checking the validity of CAR-CDR-ability. Since it takes three microcycles to complete a memory operation, this hardware checking capability is very important because it enables the programmer to fetch a data in advance without performing such a check at the microcode level. This memory operation is called **boc,** which is used in the body of the car function shown in Appendix III.

Microinstructions are divided into four types shown in Appendix I. The type IV is reserved **for** floating operations, but the current system implements IEEE standard floating operations by microcodes. One of the powerful instructions is a set of tag branches (see Table 2-l). Note that since there is no address field in the type III instructions, the linker should be intelligent to handle the combination of a type III instruction and branch instruction. Consider the following code:

```
(!lr8   (- r0 #15) (br gel (lrn lnull lreof)))

(!lrn   (- r0 #12) (br z (lr0 lrl)))

(!lr1   (jsr no store-byte))
(!lr1a  (mov r14)
        (brc tag7 (lr1' lr1'')) )
```

The instructions labeled by 1 r0 and 1 r 1 should be allocated to a consecutive address with starting an even address. In addition, since the instruction labeled by 1 rl is of type III, the next instruction labeled by 1 r la should be allocated to the consecutive address. The three instructions labeled by 1 rn, 1 null and 1 reof should be allocated to three consecutive addresses and the address of 1 rn should be a multiply of four. The linker considers these constraints of addressing and allocates instructions within the ***narrowest*** possible address range.

Table 2-l: Branch conditions on Tag field

Condition (mnemonic)	Meaning
tag7	branch if tag7 is set
tag6, tage	branch if tag6 is set
tag5, tagcadbl	branch if tag5 is set
tag5-0	**64-way** branch according to **tag5-0** bit
tagh5-0	**33-way** branch; branch to 33rd offset if **tag5=1**
tagl5-0	**33-way** branch; branch to 33rd offset if **tag5=0**
tag4-0	**32-way** branch according to **tag4-0** bit
tagf**il**	branch if **tag5-0** is not zero
tagnil	branch if **tag5-0** is zero

2.2. Firmware Development Environment

The Micro Assembler and Linker are implemented in TAO itself. Therefore, the syntax of microcodes is expressed in S-expression[1]. For example, Appendix III shows the microcode of car function. The argument of car is given on the stack and the returned value is pushed oh the stack. The microcode of binary search function is shown in Appendices II. Since the Micro Assembler and Linker are written in TAO, the user can use the power of TAO in microcodes. For example,

```
(       (mov ↑(** 2 16) r0))
```

is the same as

```
(       (mov 65536 r0))
```

That is, a form prefixed by t is evaluated before assembling. This evaluation may be postponed till linking or global linking. In the following operation,

```
(!1b1   (mov ↑(+ #10400000000 (getsym '1b1))
             -<sp> ))
```

the address of the instruction can be given as an operand at the time of linking.

The source of microcodes for the TAO interpreter consists of 112 files and its total size is about 2.7M bytes. It takes about one and a half hour for the micro assembler and linker to assemble and link all source files. The total size of used Writable Control Store is about 48K words. Needless to say, microcodes are being developed to support new functions. It takes

[1] ***S-expression*** consists of a sequence of alphanumeric characters or a sequence of S-expressions enclosed by a pair of paren theses.

about three minutes to create a binary image of WCS, which is down loaded to WCS from the front-end processor (FEP).

A mini-Lisp system is implemented on various FEP's such as PDP-11, VAX and NTT's DPE and it provides primitives to access various hardware resources of ELIS -such as WCS, sequencer, Y-bus, and processor status word. Therefore, the loader and debugger of microcodes are written in this mini-Lisp system. Since the user can inspect the status of ELIS interactively via this mini-Lisp system, the productivity of the development of microcodes was very high. The debugging tools was also being developed during the debugging of the m icrocodes.

2.3. Language aspect of TAO

TAO is a Lisp dialect and upward compatible with Common Lisp [10]. However, it is not a simple Lisp dialect but a very powerful language. TAO supports various programming paradigms within Lisp world; logic programming, object-oriented programming, Fortran/C-like programming and concurrent programming. The logic programming is embedded in Lisp by extending function types to support the primitives of logic programming; pattern matching (unification) and choice function types. The object-oriented programming is embedded in Lisp by extending ***eval***. That is, Common Lisp signals an error for a form whose **car** is not a function, while TAO treats it as a message passing form. For example, (1 + 3) is a message passing form which expresses that a message + is sent to an object 1 with an argument 3. This is an implicit message passing form whose car should be checked whether it is a function. Explicit message passing form is represented by [1 + 3], whose meaning is the same as (1 + 3). The factorial function can be defined as follows:

```
(defun fact (n)
    (if (n = 0)
        1
        (n * (fact (n - 1))) ))
```

In object-oriented programming, a factorial can be defined as a method for the class ***integer.*** The program is

```
(defmethod (integer fact) ()
     (if [self = 0]
         1
         [self * [[self - 1] fact]] ))
```

and [10 fact] calculates the value of factorial of 10. TAO provides a powerful set of concurrent primitives and its operating system is implemented on these primitives. Therefore, the TAO system supports multi-user/multi-task environments and up-to eight users can login the same ELIS at the same time.

In this paper, we will focus our attention only on Lisp programming and object-oriented programming for the simplicity of discussion. The concept of logic programming and concurrent programming in TAO will be discussed in [13].

3. Bottlenecks of interpreted execution

The execution of Lisp programs is divided into four categories, variable search, function call, type checking and real computation. In each phase, speed up is needed to provide a fast interpreter.

3.1. Variable search

Common Lisp has two kinds of variables: lexical (local) and special (non-local) variables. In the factorial program, a variable n is a lexical variable. Since lexical variables can be looked up statically, they can be accessed directly in compiled codes. However, it is one of the main problems for interpreter to speed up the access of lexical variables.

Special variables are looked up dynamically in the context of computation. For example, a built-in function, print, refers a special variable *print-pretty*. Consider the following program:

```
(def'un f (x)
   (let ( (*print-pretty*  t))
      (h x) ))                          ; {1}

(defun g (x)
   (let ((*print-pretty* nil))
      (h x)                             ; {2}
      (f x) ))                          ; {3}

(defun h (x)
   (print "banner")
   (print x) )                          ; {4}
```

The values of *print-pretty* in executing the print are t for {1} and {3}, ni l for. {2}. The value for (4) is decided on the context. Special variables may be implemented by shallow-binding or deep-binding technique. In shallow-binding, the value of a special variable is stored in the value cell of each variable. Thus, no search of special variables is needed in shallow-binding. New context for special variables is established when entering a function which contains the definition of special variables and old context is recovered when exiting the function. In other words, an old value of special variables is saved and a new value is stored in the value cell of special variables. In deep-binding, special variables and local variables are stored in a function frame or on the stack and to access a variable, the function frame chain or the stack is traversed. Therefore, shallow-binding provides faster variable lookup than deep-binding. However, the former is more expensive under concurrent programming, because process switch requires saving and restoring a context for special variables.

The implementation of TAO on ELIS adopts deep binding for special variables. This is because the cost of process switch is smaller in deep-binding implementation than in shallow-binding implementation. Furthermore, debugging tools are easy to construct in deep-binding implementation, because all information on context of computation is pushed on the stack in the manner that their stored position is directly associated with the corresponding activation frame. Therefore, for example, the backtrace function is quite easy to implement.

3.2. function call

Since Common Lisp provides a rich variety of lambda bindings such as optional arguments with/without default values, rest arguments and keyword arguments, the function call is quite heavy,- especially for interpreter. Consider the following example:

```
(defun
   foo (a b
        &optional (c 30) d (e 123 exist-p)
        &rest x
        &key start (end 10)
        &aux index (result 3) )
   ... )
```

An indicator of &optional indicates optional arguments and paired list specifies a default value. &rest indicates arbitrary number of arguments and &key indicates keyword arguments.

&aux declares local variables. In some cases, the actual computation may be done while processing function call. Macro function also introduces overheads for interpreter, because macro form is expanded before its evaluation.

3.3. type checking

Since Lisp is one of the languages which has the richest data types, type checking is very important to provide the validation of computation. In addition, some data types are very complex and their manipulation functions are overloaded. For example, number type in Common Lisp contains rational, float and complex: rational contains integer (fixnum and bignum - integer of infinite precision) and ratio, float contains short float, single-float, double-float, and long-float. A function + should work well for any type of numbers and any combination of types. Therefore, number functions should dispatch an appropriate subfunction to do the calculation. Since Common Lisp provides more than 20 data types, checking of arguments is extensively performed to validate the correctness of the computation.

3.4. real computation

Actual computations of Lisp programs are data manipulations such as list handling, numerical computation, infinite precision computation, string manipulation and vector handling. In other words, almost all kinds of computations provided by other languages may appear in Lisp programs. In the TAO/ELIS system, most of Common Lisp functions are implemented in microcodes to speed up the execution. In addition, some functions which are critical to the speed of applications such as a screen editor and networking programs are implemented in microcodes. Since this phase is a general problem for compiled codes and interpreter, we will not discuss it any further here.

4. Speedup techniques for Lisp interpreter

4.1. Usage of Tag

The implementation of TAO on the ELIS machine uses the tag in four ways.

1. To represent data types and internal data types
2. To speed up the interpreter and decrease the memory consumption
3. To make S-expression more readable to human
4. To realize new computation mechanisms such as message passing

The tag is used as a pointer tag not a self-descriptive flag in the TAO/ELIS system. That is, a. pointer includes a tag which indicates the property of the data pointed by the pointer. Invisible pointer is originally introduced to implement logic programming, but is used extensively to speed up the interpreter. Some data types and invisible pointers are listed in Table 4-1.

Checking data types is performed very efficiently in multiple branch of microcodes. If the data is given to the Y-bus at the previous instruction, branch occurs after executing the current instructions. In the microcode of car function shown in Appendix III, the branch instruction (br tag4-0 al) is performed by the Y-bus result yielded by the instruction labeled by a7. However, it is neither possible nor practical to do 64-way, 33-way or 32-way branch in each function body to check data types because of limit of WCS. Therefore, data types are first encoded to smaller groups of data. Note that the overheads introduced by this subgrouping are only one or two microinstructions.

Table 4-l: Some data types and invisible pointers

Data types or invisible pointers	meaning
nil	nil and () are discriminated to give more readable form to human.
shortnum	24 bit integer
bignum	integer of infinite precision
ratio	ratio, e.g., 2/3
float	floating-point number
complex	complex number
id	symbol
key id	keyword symbol
sysid	special symbol
logic	logical variable for logic programming
char	character
str	string
fatstr	string with font information
f ilstr	string with fill pointer
vector	vector
applobj	function object
cell	cell
namcell	named cell, e.g. table(i j k) for I/O, but the same as (table i j k).
bra	bracket
nambra	named bracket, e.g., window[move 10 20]
quoted	quote, 'foo is output as 'foo, not (quote foo)
backq	backquote macro expander
eval	comma in backquote or evaluation before unification
icar	invisible pointer to car of cell (Cdr of cell is invisible)
icdr	invisible pointer to cdr of cell (Car of cell is invisible)
splvar	special variable or closed variable
evalvar	preprocessed variable, a kind of icar
evallogic	preprocessed logical variable, a kind of icar
eval inst	preprocessed instance variable, a kind of icar
evalcdr	macro expanded form, a kind of icar
shadow	preprocessed result for let, prog, a kind of icar
comment	comment, comment is stored by using an invisible pointer, a kind of icdr

4.2. Variables in TAO

The variables in **TAO** are classified into lexical variables, special variables, semi-global variables and global variables. Semi-global variables are process-wide global, while global variables are system-wide global. Semi-global variables are introduced to provide the same mechanism as global variables to each process, because some variables in a process must be stable against accidental process reset. For example, a variable, `*history-obj*`, which holds the top-level loop conversation history, is declared as a semi-global variable attached only to the user main process.

The order of variable lookup is (1) lexical variables, (2) special variables, (3) semi-global variables, and (4) global variables. If the current environment is a message passing form, instance variables are checked before special variables. Access to an instance variable of an object will be discussed in the section of instance variables.

4.2.1. Mechanism of variable search

Since TAO uses a single stack, function frames and values are pushed on the stack. A function frame consists of chain pointers to access and control frames, function objects, arguments, and other information such as lexical scope limit and a flag which indicates whether special variables are contained or not.

The value of a lexical variable is pushed on the **stack** as an element of a function frame, while its name is not pushed. The variable names are stored in the vector, called ***how-to-bind vector*** which can be accessed via function object. To get the value of a lexical variable, the interpreter searches for the name in the how-to-bind vector to know the relative position of the variable in the frame. The interpreter repeats this lookup till it finds the variable or up to the limit of lexical scoping frame. If the variable is found, its value is returned. If the variable is not found and is declared as special, special variables are sought. Otherwise, an error is signalled. Special variables are pushed on the stack as a pair of variable name and value with a special invisible tag, called splvar. Since a frame has a flag which indicates whether special variables are contained in it, a frame without special variables are skipped and all frames are not traversed in searching for a special variable. If no special variable is found in the frame chain, semi-global variables are sought. If no semi-global variable is found, then the value of global variable is returned. However, if the value is unbound, an error is signalled.

If a variable is accessed in the body of a method, instance variables are sought before checking special variables. That is, the order of variable lookup in the body of a method is (1) lexical variables, (2) instance variables, (3) special variables, (4) semi-global variables, and (5) global variables.

4.2.2. Preprocess of lexical variables

The lexical variables are preprocessed at **the** time of definition. That is, a lexical variable is converted to a pair of variable name and its ***variable*** position on the **stack** with a tag evalvar. ***Variable position*** consists of ***fchain*** and ***offset. Fchain*** is a count for access frame chain and ***offset*** is a deviation from the target frame. This preprocess may be considered as a very Simple compilation. Figure 4-1 shows a preprocessed form of the `tarai` function. In the figure, **{evalvar}(x. #x200)** indicates that the position of a variable x in the stack is specified ***as fchain*** is 0 and ***offset*** is 2.

4.2.3. Variable cache

Variable cache is used for special variable, semi-globals and **globals** in order to speed up the search of these non-local variables. Variable cache is attached to each process. When a new function frame is created and if it contains special variables, the special variables are registered to the variable cache. When exiting a function, entries corresponding to the special variables are cleared whether they hold exactly the special variable bindings or not. Cache entries for semi-global and global variables are set only when they are' accessed. Note that no anomaly will occur even if there exist a special variable and a semi-global or global varible with the name name declared in a program. The variable cache is stored in each process. To search

```
(defun tarai (x y z)
    (if (> x y)
        (tarai (tarai (1- x) y z)
               (tarai (1- y) z x)
               (tarai (1- z) x y) )
        y ))
```

is preprocessed and converted to

```
(defun tarai (x y z)
    (if (> {evalvar}(x . #x200)
           {evalvar}(y . #x300) )
        (tarai
            (tarai  1- {evalvar}( x . #x200))
                     evalvar}(y . #x300)
                    evalvar}( z . #x400) )
            (tarai  (1- {evalvar}(y . #x300))
                    {evalvar}(z . #x400)
                    {evalvar}(x . #x200) )
            (tarai  (1- {evalvar}(z . #x400))
                    {evalvar}( x . #x200)
                    {evalvar) ( y . #x300) ))
        {evalvar) ( y . #x300) ))
```

Note that **#x200** reads 200 in hexadecimal.

Figure 4-1: Preprocessing of lexical variables

for a non-local variable, the interpreter checks the cache and return the value if found. If the cache entry is void or holds other variable binding, that is, cache doesn't hit, the frame chain is traversed to search for the variable as described before. If the cache hits, the performance of this cache mechanism is quite similar to that of shallow-binding technique. Note that the variable cache is automatically write-through, because cache entries hold a binding cell tagged with splvar. Note that the tag is used as a pointer tag, any data can be carried out to anywhere.

TAO provides direct access methods to global and semi-global variables. (Value expression) and (sg-value expression) are used to access a global and semi-global variable directly, respectively. Semi-global variables are sought by a binary search. If global or semi-global variables are used as a means of communications between several functions, value or sg-value will give a direct and fast access method to the user.

4.2.4; Preprocess of Instance variables

Instance variables are not stored on the stack but in an instance vector. Instance vector is held& a value of a variable **self**, which is a kind of lexical variable and pushed on the stack as the first argument.

Since object-oriented system in TAO provides a hierarchical decomposition of data and programs, each class has only its own definitions of instance variables for data and methods for programs. Each class has several superclasses whose instance variables and methods are inherited to it.

Each class has its all instance variables including the inherited ones from superclasses and, thus, the offset of the same instance variable in the instance vector may vary among classes. If inherited methods are copied to subclasses, the offset can be determined. This copying technique is not adopted in the TAO/ELIS system by considering the tradeoff among memory waste and efficiency. Instead of copying, instance variables are preprocessed to point to **self**, not to themselves. This preprocess is the same as that of lexical variables, except the tag.

That is, an instance variable is converted to a pair of the variable name and the variable position with a tag evalinst. After getting an instance vector, instance variable is sought by a simple linear hashing.

4.3. Function calls

4.3.1. Function invocation

Symbols in **TAO** has one of four tags; sysid, id, **keyid** and logic (see Table 4-l). The latter two tags are for **speedup** to check a keyword and logic variable, respectively. Symbols with sysid tag are microcoded primitive functions such as car, cdr, cons and so on. The entry address **of sysid** function in microcodes is the same address of a sysid symbol. That is, if the address of `car` is **#143** (in octal) in memory, the entry address of microcodes of `car` is **#143** in WCS (see Appendix TIT). Furthermore, checking the number of arguments is embedded in the body. Therefore, to lookup a function definition is not needed to check the TV shows the control flow **in** evaluating `(car . . .)`.

Every function has a function definition table which contains information on arguments and function body. Common Lisp provides various kinds of arguments of functions such as obligatory, optional and **rest** arguments. However, if a function has only obligatory arguments, it suffice to check only the number of arguments. Such a function is called expr-simple or subr-simple and its invocation is faster than expr (interpreted function) or subr (microcoded function), because checking arguments in the former is much simpler.

4.3.2. Special dispatch of built-in message

In TAO, primitive data types such as integer, list, or symbol, can be treated as a class. These classes have several built-in messages such as **+, <.** The method corresponding to these **built-in** message is **invoked** directly without searching the method table. The key idea is quite similar to sysid functions. **There are 14 reserved** built-in messages; that is, **+, -, *, **, /, >, <, =, >=, =<, /=, ., . .** and `belongs-to` for the moment. These built-in messages have a sysid tag and the entry address of the corresponding method is calculated by adding the offset unique to the class to the address of a message symbol. Micro assembler and linker supports absolute addressing as well as symbolic addressing for this purpose.

4.3.3. Fast lookup of message-method table

Object-oriented programming in TAO **[9]** is quite similar to the ***original*** FLAVOR system **[15].** All methods defined to a class including inherited ones are registered in the **message-**method table associated to the class. The table is sorted by the address of message, and a method is sought by binary search. The microcode of binary search is shown in Appendix **II.** The cost of method lookup is $\log_2 n$ **μsecond,** where n is the total number of methods defined in the class including inherited ones.

5. Evaluation of the TAO interpreter

5.1. Benchmark results

The data shown in Table **5-1** except for TAO is an excerpt from **[8].** Symbolics-3600 with Instruction Fetch Unit (IFU) and 8 Mbytes memory is used to compare the performance with the **TAO/ELIS** system, because it is the fastest commercial Lisp machine. Symbolics-3600 without TFU is about 30 ~ 40% slower that one with TFU. Roughly speaking, the interpreter of the **TAO/ELIS** system runs much faster than that of Symbolic-3600 but we cannot say which is faster, the interpreter of the **TAO/ELIS** system or the compiler of Symbolics-3600. **It** depends on benchmarks.

The definition of `tarai-5` is shown in Fig. 4-l with arguments 10, 5, 0. The tak is a

Table 5-1: Benchmark results according to [8]

benchmark	TAO *interpreted*	Sym bolics *interpreted**	*compiled*[2]
Tarai-5	1.0 3	***44.9***	0.17
Tak-18-12-6	1.0 3	41.8	0.15
List-tarai-4	1.0 3	36.8	2.52
String-tarai-4	1.0 3	26.0	3.50
Bignum-tarai-4	1.0 3	40.8	2.48
Flonum-tarai-4	1.0 3	30.3	0.26
Bit-A-6	1.0 3	21.4	0.69
TPU-3	1.0 3	21.4	1.20
TPU-4	1.0 3	21.0	1.32
Boyer	1.0 3	33.8	0.28

1 Release 5.0 without Instruction Fetch Unit
2 Release 6.0 with Instruction Fetch Unit and scheduler off

modified tarai, which is well-known in the American Lisp community. String-tarai, list-tarai, bignum-tarai, flonum-tarai is a modified tarai for various data types. For example,

```
(defun list-tarai (x y z)
    (if (< (car x) (car y))
        (list-tarai
           (list-tarai (copy (cdr x)) y z)
           (list-tarai (copy (cdr y)) z x)
           (list-tarai (copy (cdr z)) x y) )
        y ))
```

is the definition of list-tarai and the the speed is measured by

```
(list-tarai '(1 2 3 4 5 6 7 8 9 10)
            ' 5 6 7 8 9 10)
            ' 9 10) )))
```

which is a variation of (tarai 8 4 0). These data shows that TAO provides efficient data type manipulations except for floating point operations. This is because 64-bit IEEE floating point number is manipulated by microcodes. These operations will be implemented by hardware in the future. The bit produces all permutations of a list of length 6 by a mapping function. The TPU is a theorem prover by Unit resolution and its program size is about 400 lines. The Boyer is a well-known benchmarks, but the size of program is smaller than that of TPU and it uses property lists extensively.

The process switching takes about 40 μsec. Although logic programming is not discussed here, the speed of logic programming in TAO is about 11.5 KLTPS.

5.2. Speedup of variable access

5.2.1. Lexical variables

Table 5-2: Execution time ratio between non-/preprocess

	tak-18-12-6
preprocessed	1.00
no preprocess	1.62

The typical time to access a lexical variable is 1.7 μsec, while the compiled code takes 0.6 μsec. Table 5-2 shows that speed-up factor by preprocessing lexical variables to evalvar is 1.62 for the tak function.

5.2.2. Special variables

The programs shown in Fig. 5-l proves that benefit of variable cache will be gained if the same special variable is accessed more than twice, that is, for all n where $n > 2$. **Of** course, the cost includes cleanup time to remove the entry of x from variable cache as well as setup time. Note that Gabriel's **stak [3]** (tak function with special variables) runs slower with variable cache than one without it, because every special variable is accessed only once. Since an expert building tools called KRTNE **[6]** uses many special variables, KRINE runs two to seven times faster with variable cache. **Its** resulting speed is comparable to compiled codes of KRINE on Symbolics-3600.

```
(defun f (x)
    (declare (special x))
    (g) )

(defun g () x1 x2 . . . xn)
```

where x_1 is x.

Figure 5-1: Benchmark to evaluate variable cache for special variables

5.2.3. Instance variables

Table 5-3 shows the speed to access some instance variables of an object which has 50 instance variables. Instance variables are accessed in two ways; as a name and by a message passing. Consider the following object.

```
(defclass ship () (x-pos y-pos) ()
          :gettable :settable )

(defmethod (ship distance) ()
    (sqrt [[x-pos ** 2 +
          [[self y-pos3 ** 2]] ))
```

The class ship has two instance variables and these variables are accessed by its name. In the

`distance` method, the value of x-pos is accessed by its name, while the value of `y-pos` is accessed by a message passing, `[self y-pos]`. The **speedup** factor by preprocessing is from 1.5 to 5.8 and 1.4 for a name access and a message passing, respectively. Name access for the last instance variable in an instance vector is the most time consuming because the search is linear from the first instance variable to the last one.

Table 5-3: Speed of instance variable access

Instance variable position	ELIS[1] *interpreter*	ELIS[2] *compiled*	Sym bolics *compiled*	TI/Explorer
first	1.36	2.47	0.47	0.91
by MP[3]	1.94	2.32	2.53	5.87
last	1.36	9.23	0.45	0.94
by MP[3]	1.83	2.38	2.53	5.88

The unit time is microsecond.
[1] Preprocessed [2] Not-preprocessed
[3] MP = message passing

The CARE system [1] is an instrumented multiprocessor simulation system developed at Knowledge Systems Laboratory, Stanford University. The CARE system is a large system (the siie of source codes is about 600K byte) implemented in object-oriented programming. That is, it is written in **ZetaLisp** and Flavors system [15] and uses only a few special variables. We ported the CARE to TAO **(CommonLisp)** with TAO's object-oriented system. The interpreted codes of the CARE system runs on the **TAO/ELIS** system nearly as fast as the compiled codes of the TT/EXPLORER with 8 Mbytes memory system.

5.3. **Speedup** of function invocations

5.3.1. Function invocation

Table 5-4 shows that the **speedup** by ***expr-simple*** function is about 1.12 for **tak-18-12-6. The** `tak` function uses three arguments. The more the number of arguments of ***expr-simple*** is, the faster a function is invoked.

Table 5-4: Execution time ratio for exper-simple

	tak-18-12-6
expr-simple	***1.00***
expr	1.12

5.3.2. Method search

Table 5-5 shows that sending a built-in message is executed almost as fast as Lisp functions. Note that a bracket form such as [x + y] is treated as a message passing form without checking a normal form, while a form (x + y) is first checked whether x is a function or not. This overhead for the latter is not negligible if the real computations is nbt small like + or =. As a consequence, the user is not recommended to use a parenthesized form such as (x + y) as an overloading means to a message passing, although this overloading is a new interpretation of Lisp forms proposed by the TAO/ELIS system.

Table 5-5: Speed comparison between prefix notation and infix notation

form	time	form	time
(+ x y)	12.92	(= x y)	11.74
(x + y)	18.10	(x = y)	18.17
[x + y]	12.06	[x = y]	12.09

unit: microsecond

```
(defun fib (n)
       (if (< n 2)
           1
           (+ (fib (- n 1)) (fib (- n 2))) ))
```

Figure 5-2: Lisp style Fibonacci function

```
(defmethod (integer fib) ()
       (if [self < 2)
           1
           [[[self - 1) fib] +
            [[self - 2] fib] ]))
```

Figure 5-3: Object-oriented style Fibonacci function

Table 5-6 shows the results of Fibonacci function written in Lisp and object-oriented programming (Fig.52 and Fig.93) and gives two conclusions. First, the method search is only a 596 overhead to Lisp function call. Second, if the method is found in the worst case by binary search, the execution is slow down by 7% and 10% for 30 and 100 user-defined messages, respectively. Since the overhead is small, we can say that the merit of object-oriented programming is not be subsumed by the overhead of execution. In fact, many applications are implemented in object-oriented programming in the TAO/ELIS system, examples being an Emacs-like editor, TCP/IP and network application programs, operating system.

6. Discussion

The experience of implementing the TAO/ELIS system proves that a naive implementation of Lisp interpreter in firmware cannot provide high performance and that microcoded interpreter should incorporate many speedup techniques. With various techniques discussed in this paper such as data dispatch, variable search, function invocation, method search, the resulting TAO/ELIS system provides a very fast interpreter of which speed is comparable to the compiled codes of commercial Lisp machines.

Table 5-6: Speed comparison
between Lisp and Object-Oriented programming

Lisp style	time in μsec		
(fib 19)	783		
(fib 22)	3,394		
(fib 25)	14,376		
Object-oriented style	***size of method table***		
	1	30	***100***
[2219 fib fib]	3,364 795	3,610 853	3,730 880
[25 fib]	14,246	15,294	15,800

These techniques presented in this paper can be applied to any (compiler-bases) deep-binding Lisp system as well as any Lisp interpreter. Much attention is recently paid to implementation of Lisp by deep-binding mechanism, because parallel Lisp system forces such an implementation [2, 4]. In parallel or concurrent Lisp system, many processes are spawned and process switching is critical to the performance. If the variable binding mechanism is implemented by deep-binding mechanism, process switch is very easy because all information on computations is stored in the stack. This is the criteria why the TAO/ELIS system adopts deep-binding mechanism. Although the TAO/ELIS system is a Lisp machine system, it works as a multi-user system like Unix.

The TAO/ELIS system proves that the high level approach to firmware development environment is very important. That is, micro assembler and linker are written in TAO itself and micro loader and debugger are written in mini-Lisp system running on the FEP. As a consequence, any simulator, either hardware level or software level, was not used to design and develop the breadboard ELIS and the TAO interpreter. Note that the TAO/ELIS system has no machine instructions as convensional machines. The system uses the bytecode interpreter to execute compiled codes, but most computations are executed by microcoded Lisp functions. Byte codes manipulates only function calling and exiting. If a set of machine instructions is fixed, it is very difficult to incorporate new functionalities to the system. As Lao-Tsu said 'The TAO named TAO is not the true TAO", the TAO/ELIS system is ever evolving. In fact, the TAO/ELIS system supports object-oriented programming, logic programming, Fortran/C-like programming, concurrent programming and database management capabilities as well as Lisp. -We believe that firmware approach gives this flexibility to language design.

The current status of the TAO/ELIS is that Japanese word processing system, window system, Emacs-like editor, network system, C programming environment (C is compiled to TAO) and other utilities are developed for the TAO/ELIS system. Even if the TAO/ELIS system is an Interpreter-centered system, compiler is useful for memory economy and further speedup. The development of compiler for Lisp and object-oriented programming is almost completed and that for logic programming is under development.

It will be an interesting research theme to use the ELIS machine to implement other high-level language such as Smalltalk, because the ELIS machine is not dedicated to Lisp but a general-purpose stack machine. In addition, powerful firmware developing environments are provided by the TAO/ELIS system. This approach will be in a striking contrast to RISC approach.

Acknowledgments

The authors thank Yasushi Hibino and Kazufumi Watanabe, NTT Human Interface Laboratories, who designed ELIS and VLSI ELIS and made the prototype ELIS. They express thanks to their colleagues of NTT Software Labs for developing various application softwares and evaluating the TAO/ELIS system. They also express thanks to the members of NTT Human Interface Labs, to design and develop VLSI ELIS. They thank Dr. Katsuji Tsukamoto for his continuous support to the NUE project. They also thank Prof. Edward Feigenbaum for giving two of them a chance to write this paper and to evaluate the TAO/ELIS system at Knowledge Systems Lab, Stanford University. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim and by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875 to Advanced Architectures Project at KSL.

References

1. Delagi, B.A., Saraiya, N.P., Nishimura, S., and Byrd, G. An Instrumented Multiprocessor Simulation System. Report KSL 86-35, Knowledge Systems Laboratory, Stanford University, Palo Alto, CA, January, 1987.

2. Gabriel, R.P. and McCarthy, J. Queue-based multiprocessor Lisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984.

3. Gabriel, R.P.. ***Performance and Evaluation of LISP Systems.*** MIT Press, Cambridge, MA, 1985.

4. Halstead, R. MultiLisp. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984.

5. Hibino, Y., Watanabe, K., and Osato, N. The architecture of Lisp machine ELIS ***(in Japanese).*** Report of WGSYM 24, IPSJ, June, 1983.

6. Ogawa, Y., Shima, K., Sugawara, T. and Takagi, S. Knowledge Representation and Inference Environment: KRINE, --- An Approach to Integration of Frame, Prolog and Graphics. Proceedings of the international conference on Fifth Generation Computer Systems (FGCS '84), ICOT, Tokyo, Japan, October, 1984, pp. 643-651.

7. Okuno, H.G., Takeuchi, I., Osato, N., Hibino, Y. and Watanabe, K. TAO : A Fast Interpreter-Centered System on the Lisp Machine ELIS. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August, 1984, pp. ‘140-149.

8. Okuno, H.G. The Report of The Third Lisp Contest and The First Prolog Contest. Report of WGSYM 33-4, IPSJ, September, 1985.

9. Osato, N., Takeuchi, I. and Okuno, H.G. Object-Oriented Programming in TAO ***(in Japanese).*** In Suzuki, N., Ed., ***Object-Oriented System,*** Kyoritsu Publishing Inc., Tokyo, Japan, 1985.

10. Steele, G.L.. ***COMMON LISP : The Language.*** Digital Press, Burlington Massachusetts, 1984.

11. Takeuchi, I., Okuno, H.G. and Osato, N. “TAO - A harmonic mean of Lisp, Prolog and Smalltalk.” ***SIGPLAN Notices 18, 7*** (July 1983).

12. Takeuchi, I., Okuno, H.G. and Osato, N. “A List Processing Language TAO with Multiple Programming Paradigm.” ***New Generation Computing*** **4, 4** (1986).

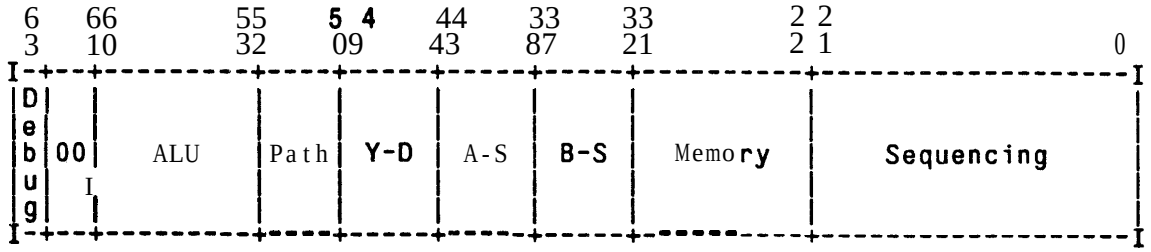
13. Takeuchi, I., Okuno, H.G., Osato, N., Kamio, M. and Yamazaki K. A concurrent multi-paradigm list processor TAO/ELIS. Proceedings of Fall Joint Computer Conference (FJCC'87), ACM & IEEE, Dallas, Texas, October, 1987. ***to appear***

14. Watanabe, K., Ishikawa, A., Yamada, Y. and Hibino, H. A 32b LISP Processor. Proc. of IEEE International Solid-State Circuits Conference (ISSCC '87), IEEE, New York City, February, 1987, pp. 200-201, 394.

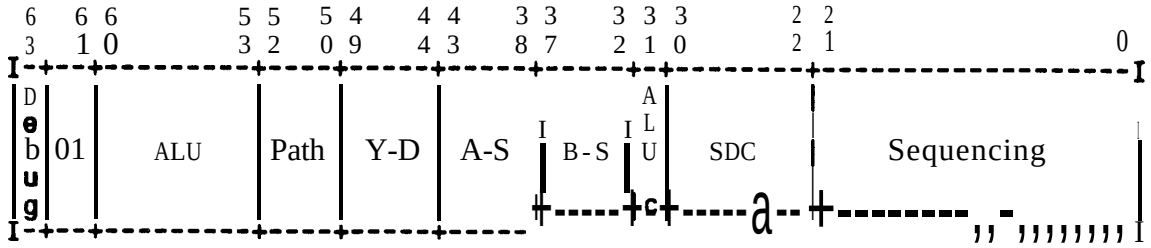
15. Weinreb, D., Moon, D. and Stallman, R.M. ***Lisp Machine Manual.*** LMT, 1983.

I. Microinstruction Format

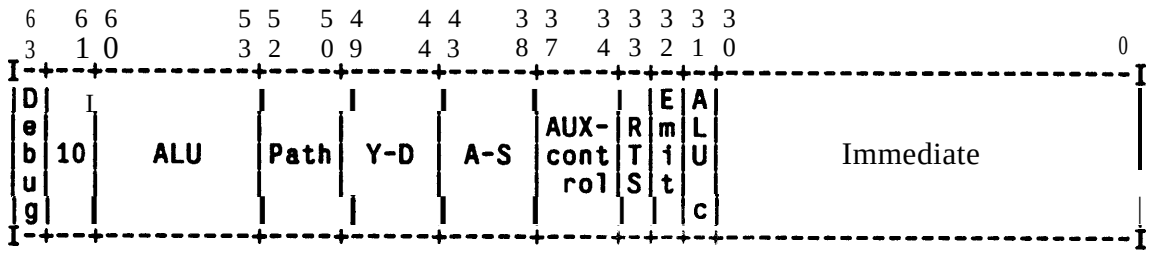
<< Type I >> Memory Reference type



<< Type II >> SDC Control type



<< Type III >> Immediate type



<< Type IV >> Reserved for floating point operations

• ** legend • **

ALU c	ALU carry control
Emit	MSB of Immediate data

II. Micro code of binary search for id-message

```
; binary search for id-message
; r6 = the lowest position of the current table
; r2 = the highest position of it
; car0 = position to be examined
; r0 = key
; all relevant addresses are nilnum!
; returns car1 = corresponding method if not nil

(!!bins (- r6 r2 car0) sra (bo car0 mdrl))      : car0 = middle point
(!lp    (- r2 r6))                              ; something remains?
(       (- car0 1 r3)                           ; r3 = next possible highest pos
        (br n (cont not-found)) )               ;   yes/nothing remains.

(!cont  (- cdr1 r0))                            ; compare key and contents
(       (+ car0 1 r4)                           ; r4 = next possible lowest pos
        (br gel (big found small)) )            ;   bigger/found/smaller

(!found (and car1 #17777777777 car1) rts)       ; return method without gcmark

(!big   (+ r6 r3 car0) sra (bo car0 mdrl))      ; bigger, get the middle point
(       (mov r3 r2) (got0 lp))                  ; update the highest position

(!small (+ r4 r2 car0) sra (bo car0 mdrl))      ; smaller, get the middle point
(       (mov r4 r6) (goto lp))                  ; update the lowest position

(!not-found
        (mov #20000000000) rts)                 ;  return a not-found code
```

III. Microcode of the body of the car function

```
(.local *nil-car 4)                                  ; car-nil error flag
(.local :car ↑(sysid #143))                          ; address of symbol car

; entry point of car body --  its argument is on sp

(!!car  (and <sp>+ gmc car0) (boc car0 mdrl)         ; read car0 to mdrl
        (br nhap (a8 a7)) )                          ; check special condition
                                                     ;   something-happened/*

; entry point of car --  its argument is on car0

(!!car.s (mov car0) (boc car0 mdrl)
        (br nhap (a8 a7)) )                          ; check special condition
                                                     ;   something-happened/*
(!a7     mov car0 rpr)
        (br tagcadbl (a3 a4)) )                      ; branch on cadbl data type
                                                     ;   error?/ok
(!a4    (mov 1 rpf) (br tag4-0 al))                  ; cadble, invisible?
                                                     ; rpf 1 means rplaca assign
(.case al cadr#
 (inv-a  mov car1 car0) (got0 car.s))                ; invisible in car
 (inv-d  mov cdr1 car0) (got0 car.s))                ; invisible in cdr
 (t      mov <sp>) (got0 a2)) )                      ; (car1 . cdr1) is founded.
                                                     ; yield return addr on Y-bus
(!a2    (and car1 gmc <sp>) return)                  ; push return value and return

(!a3    (and sysmode *nil-car)                       ; car-nil error?
        (br tagnil (a5 a6)) )                        ; is it nil? no/yes

(!a6    (clr rpf) (br z (a9 a10)))                   ; should be car-nil error ?
                                                     ; nil is not rplacable
(!a10   (mov <sp>) (got0 rtnnil'))                   ; not error, returns nil

(!a9    (clr r8))                                    ; car-nil is error
(!!ae1  (mov :car r7))                               ; errored fn is car
        (mov :illarg rl))                            ; set error message
        (got0 err))
(!a5    (mov car0 r8) (goto ael))                    ; non-car-cdrable thing
(!!a8   (mov car0 -<sp>))                            ; store back arg
        (mov car' -<sp>))
        (mov sbr) (br spover hap))                   ; stack overflow?
(!car'  (mov <sp>+ car0) (boc car0 mdrl)             ; resume car operation
        (got0 a7) )
```

IV. Evaluation of a form (car . ..)

Entry of Eval

```
; <sp> => form
; <sp+1> => return address

(!!eval (and <sp>+ gmc car0) (boc car0 mdrl) (br nhap (evi ev)))
(!!ev   (br tag5-0 eval-disp) (corn sysmode))      ; check evalhook
(!!ev-nohook                                       ; eval without hooking
        (br tag5-0 eval-disp) (mov -1) )

(.case teval-disp dtyp#
 ((list dnil keyid shortnum bignum ratio float codnum undef
   bigfloat str char fatstr filstr complex shortfloat)
  (mov <sp>) (br y6 (hevconst evconst)) )
 ((list sysid id)
  (and car0 gmc rl)
  (br y6 (hlispv lispv)) )                         ; rl = variable to be searched
 ((list cell namcell)                              ;For the case of (car . ..)
  (mov car0 -<sp>)
  (br y6 (hform form)) )                           ; push form onto stack
 ... )
```

Analyze a form

```
(!form  (asrc car1) (aluh zero) (ydes rpf) 24bw ; clear rpf every time for form
        (boc cdr1 mdr0) )                       ; get arg list in mdr0
                                                ; (first-arg  arg-tail)
(!form1 (br tag5-0 car-form)                    ; dispatch car of form
        (mov  car1 rl) )                        ;  rl = car of form

(.case car-form dtyp#
 (sysid (and <sp> gmc r2) (br ybr 0))           ; sysid, jump by its addr
                                                ; r2 = the form
                                                ; gm clear for indicating that
                                                ; this sysid call is from eval
 ((list id logic keyid)
        + car1 1 car0) (bo car0 mdr0)           ; mdr0 <- (applobj . prop)
        (goto idf) )
 ... )
```

Entry of car

```
(!#143  (mov car <sp>))                         ; Symbol-car's address is #143
(       (mov cdr1) (goto sysl))                 ; car is a label of microcodes
```

Arguments check

```
; 1 arg sys subentry
; Upon entering, r2 = the form

(!! sys1 (mov cdr0) (br tagh5-0 s0))            ; 1 arg sysid subentry
                                                ; branch on previous result
(.case s0 dtyp#
 (inv-a (mov car0 cdr1) (bo cdr1 mdr0) (goto sysl))
 (inv-d (mov cdr0 cdr1 (bo cdr1 mdr0 (goto sysl))
  dnil  (mov <sp>+) (nua sel))                  ; no arg, r2 contains the form
 ((list cell namcell bra nambra quoted eval# backq assign usym
        selfass assignee )
        (and car0 gmc car0) (boc car0 mdrl)     ; copy of eval head
        (br tagcadbl (s2 s3)) ))                ; is there excess arg? no/yes
```