ISE-TR-80-19

ASSOCIATIVE DESCRIPTOR SCHEME - FOR THE EXPLOITATION OF ADDRESS ARITHMETIC IN LISP

by

Tetsuo Ida

Kozo Itano

December 15 , 1980

INSTITUTE
OF
INFORMATION SCIENCES AND ELECTRONICS

UNIVERSITY OF TSUKUBA

Associative Descriptor Scheme - for the exploitation of address arithmetic in Lisp

Tetsuo Ida* and Kozo Itano**

* Information Science Laboratory
Institute of Physical and Chemical Research
2-1, Hirosawa, Wako-shi, Saitama 351, Japan

** Institute of Information Science and Electronics
University of Tsukuba
Sakura-mura, Niihari-gun, Ibaraki 305, Japan

Abstract

We present an associative descriptor scheme to take advantage of address arithmetic in accessing lists and vectors. The scheme employs an associative table for storing descriptors for lists and vectors. In parallel with each access to the structure with its base and index, the associative descriptor is searched for the size of the structure and hardware range check is performed. This operation hence can be done with little or no overhead in accessing the structure. This scheme is straightforwardly applied to fast indexed access to vectors and also to fast random access of any list element when combined with CDR-coding. This fact further suggests a method for uniform handling of a list and a vector. Implementation aspect of the scheme using parallel hashing is also discussed.

Key Words and Phrases

Lisp, architecture for Lisp, range check hardware, associative memory, hashing

1. Introduction

Exploitation of address arithmetic capability to access a word in linear structure has long been a common practise. It has been limited in Lisp environment due to the overhead of range check and to the storage structure of a list.

'Array feature' does exist since the earliest implementation of Lisp [1], but a need for efficient implementation of 'indexed access*' was fully recongnized only recently by developers of a Lisp-based large scale formula manipulation system [2]. Along this line Fitch and Norman implemented Lisp in which vectors (one-dimensional arrays) play essential role in speeding up 'big-num' arithmetic [3]. Efforts have been made to establish a vector as a basic and well-defined standard data type in Lisp [4].

On the other hand, a technique of accessing a random element of a list was suggested, although not seriously considered so far, in the researches on microprogrammed Lisp machines [5]. In recent implementations of Lisp on such machines, elements of a list are arranged in adjacent locations, wherever possible, by replacing the link pointer (called CDR-link in Lisp) by a few bit tag for economizing the storage. This method, known as

---

* By "indexed addressing (or access)" in this paper we mean memory addressing (or access) by the address obtained from the addition of head address of a linear contiguous area in the address space and a positional positive index from the head. It does not necessarily imply the mode of addressing specified in the instruction words, as is often found in conventional computer architecture.

'Halfword Lisp' [6] or 'CDR-coding'[5], shows the possibility of use of indexed addressing to obtain a random element of a list.

Keller also noted the possibility of random access to the linear structure of a list in the CDR-coding scheme [7], but discarding the CDR-coding scheme, proposed new data structures tuple and conc; tuples for representing what is essentially a fixed linear list for taking advantage of indexed random access and conc for avoiding long threaded list structures (or more positively exploiting the resulting structure in parallel computing).

Observing the developments towards incorporating indexed access capability to Lisp in the arena of software, we investigate in this paper the issue of the indexed addressing from architectural point of view. Because the inefficiency involved in the indexed access to either lists or vectors is attibuted to the range check of whether the computed address is in the range of the intended linear contiguous area of the storage, we first focus on range check and propose the use of hardware range checker.

In section 2 we consider the case for speeding up the reference of an element of a vector and present an associative descriptor scheme for the range check. Then in section 3 we discuss how the scheme can be applied to speeding up the reference of a random element of a list. Further in section 4 we discuss implementation aspect of the scheme.

2. Speed-up of vector element referencing

Speeding up the element reference and complete range check on conventional systems is a conflicting requirement and often the latter is sacrificed for the former. However, range check is indispensable in the programming environment where vectors (possibly many and small) are created and deleted dynamically and their range is generally undetermined at compile time, such as in the case of Lisp. Illegal access to unintended area could result in total disruption of storage management mechanism.

The check should be made prior to or in parallel with the memory access. With pure software means, the code for the range check is inserted before actual memory access. The check requires time longer than the actual memory access on conventional architecture, even when the vector is statically allocated. More overhead for the range check of dynamic vectors will be expected. Program verifiers [8] and some optimizing compilers [9, for example] can remove range check code in some restricted cases of static vectors, but their usefulness is very limited in general cases.

A typical and general method to cope with the dynamic vectors is to provide a descriptor of a vector as shown in Fig. 1-a (method A). In this case, besides code fetch, three memory accesses are necessary; for the upper limit, the base and the element itself. Another method (method B) is to provide a header in the vector body and to place the upper limit in the header. In method B, a vector is represented by a pointer to

the head of the vector body (Fig. 1-b), rather than by a pointer to the descriptor in method A. Method B eliminates the indirection to get to the head of the vector body but loses the flexibility in handling; in particular sharing the part of the vector body is difficult. In both cases the memory access to obtain the limit and subsequent check of the index with the limit must precede the memory access to a vector element.

The above defects can be removed by providing descriptors in an associative table and by representing a vector by a pointer to the vector body, as shown in Fig. 2. We call this scheme 'associative descriptor scheme'. Descriptors disappear from the software scene, in that only vector bodies and the pointers to them are present in the main memory. Thus we can obviate the indirection via a descriptor to get to the vector body without losing the flexibility.

In accessing the i-th element of the vector, we require that

(1) access path to the associative memory (to be called AM hereafter) is different from the one to random access main memory (to be called RM hereafter) where the vector body is stored,

and that

(2) the sum of the access time to AM and the time for comparison of limit u and index i is less than or comparable to the sum of address addition time (base b plus index i) and the access time to RM.

Descriptor (b u) which consists of base b and upper limit u is stored in AM. The vector base is a key to interrogate AM and limit u associated with the key is retrieved as a result. In reading, access to AM and RM are made in parallel, and by the end of RM read cycle the result of range check is reported. In writing, access to AM should precede the RM write cycle. In Table 1 we give basic operations on AM.

Operations for the following basic Lisp functions*) in our scheme are now described below:

GETV[v,i]: Get the i-th element of a vector v.
v points to the vector body.
1. Compute v+i and initiate SerAM[v] (cf. Table 1) simultaneously.
2. Initiate memory read cycle with address v+i.
3. Obtain the result, u of SerAM[v].
4. If i>u then obey interrupt sequence**) else return the read-out value when the read cycle is completed.

PUTV[v,i,w]: Place the value w in the i-th element of a vector v.
1. Compute v+i and initiate SerAM[v] simultaneously.
2. Obtain the result, u of SerAM[v]
3 If i>u then obey interrupt sequence else initiate memory write cycle with address v+i and value w.

MKVECT[u]: Create a new vector whose upper limit is u.
1. Set v <- alloc(u+1).
{alloc is a function which allocates storage for consecutive u+1 elements and return the head address of the allocated storage.}
2. CreAM[v,u] and return v as the result.

---

* We use names of the functions on vector handling given in Standard Lisp Report[4].
** The interrupt sequence is not elaborated in this paper.

To delete vector v whose descriptor is (v u), DelAM[v] (cf. Table 1) is performed.

In list processing sharing the structure or sub-structure is customary and important. The simplest functions which effect the sharing are CDR, CDDR, ... etc.. It is highly desiarable not only for the sake of more flexible handling of a vector but for the sake of the notional compatibility of a list and a vector, that the sharing of the structure can be achieved with maximum efficiency, like CDR on a list. Function SUBVECT[v,m] which corresponds to CAD$\underbrace{\ldots}_{m}$DR performs that function.

SUBVECT[v,m]: Take a sub-vector of vector v, that is the vector excluding the first m elements of the vector v.

1. SerAM[v] and obtain limit u of vector v.
2. If m > u then obey interrupt sequence.
3. CreAM[v+m,u-m] (cf. Table 1) and return v+m as the result.

A problem on sharing sub-structures by modifying the already built structures is discussed in section 3 in conjunction with RPLACD in the associative descriptor scheme.

3. Linearized lists and fast look-up of a random element of a list

In the CDR-coding scheme mentioned in section 2, 2 bit tag is used in each list word to distinguish the four cases of adjacency [5]:

1. CDR is the next word (CDR-NEXT).
2. The next word is used as CDR-link (CDR-NORMAL).
3. The word is the end of list (CDR-NIL).
4. The word is used for indirection (INDIRECT).

When n element list is structured in a way that each of the first n-1 consecutive words is tagged as CDR-NEXT and the last word is tagged as CDR-NIL, it is called linear. A linear list exhibits the same data structure as a vector body and each element of which can in principle be accessed by using the index from the head of the list. That is, the i-th element can be accessed by computed address of p+i where p points to the head of the list.

This simplified version of list element referencing suffers from two defects:

(1) List structures change dynamically; for example when RPLACD is operated on them, the linearity of the structures is generally lost.

(2) Because the number of elements of the list is not readily known in many cases, upperbound range check will be difficult.

We recall that in our scheme of section 2 the descriptors are used for range check of accesses to any linear contiguous area. Hence, provision of the descriptor to each linear list or

linear part of lists (partially linear lists) such as shown in Fig. 3 is sufficient to check whether the addressed word is a part of a list. That is, whenever linear lists which are meant to be accessed by an index, a descriptor of a list, consisting of the head address and the upper limit is created dynamically in AM using the primitives of Table 1, as in vectors. In the case of a partially linear list, a descriptor for that particular linear part is created.

To further elaborate our discussion, we present three basic functions to be used in the associative descriptor scheme;

VECTL[r]	to create a descriptor of list r in AM, thereby enabling the use of indexed access capability on r,
DEVECT[r]	to remove a descriptor of r from AM, if exists, thereby disabling the use of indexed access capability on r,
SELECTN[r,i]	to select the i-th element of list r, using the descriptor(s) of r stored in AM, if any.

VECTL[r]: Let r be a list (r0 r1 r2 ... rk) .
After the execution of VECTL[r], random access to an element of list r can utilize the address arithmetic.

1. Set u <- SerAM[r].
2. If u $\geq$ 0 then goto step 6.
3. {r is not in AM.} If the tag of the list head is INDIRECT then terminate the algorithm returning -1.
4. Scan the list from the head until finding CDR-NIL, CDR-NORMAL or INDIRECT, and obtain u' ($\leq$k) of the upper limit of the linear part of the list.
5. Enter (r u') in AM, i.e. CreAM[r,u'] and terminate the algorithm, returning u' as a result.

6. {r is linear upto the u-th element.}
   Access directly the u-th element by address r+u.
   Scan the list from the u-th element until finding
   CDR-NIL, CDR-NORMAL or INDIRECT and
   obtain i ($\leq$ k - u) of the additional
   linear length of the remaining list.
7. Enter (r u+i) in AM, i.e. RepAM[r,u+i] (cf. Table 1) and
   terminate the algorithm, returning u+i as a result.

(Figure 4 shows the process of the execution of VECTL.)

DEVECT[r]: This function is to remove the descriptor for r
from AM, if any.

DelAM[r] does exactly the function of DEVECT[r].

SELECTN[r,i]: This function selects the i-th element of list r if i is less than the length of the list.

1. Set u <- SerAM[r].
2. If u=-1 then goto step 5.
3. If u<i, goto step 6.
4. Read the word at address r+i and terminate the algorithm. {Steps 3 and 4 are performed in parallel.}
5. Scan the list from the head upto the i-th element and if CDR-NIL is encountered on the way then terminate the algorithm with error else terminate the algorithm, returning the i-th element.
6. {r is partially linear.} Access the u-th element.
7. If the tag of the u-th element is CDR-NIL then terminate the algorithm with error.
8. {Get to the next node.} Set t<- the content of the word at address r+u+1.
9. Call recursively SELECTN[t,i-u-1] and terminate the algorithm, returning the result of SELECTN[t,i-u-1].

Note that (i) even when the descriptor for r is removed, r can be accessed as an ordinary list scanning each element of the list and that (ii) the execution of VECTL does not affect the data structure in RM. Hence, CAR, CADR, CDR etc. can be performed as before.

In effect, the descriptor scheme for lists is a device for speeding up the element referencing. Programmers can have entire control over the use of VECTL and DEVECTL on lists except that DEVECT is also forcibly performed by the system's garbage collector and by the functions which modify the structures, notably RPLACD.

When RPLACD is perfomed on the linear list, the related descriptor(s), if any, must be updated to reflect the fact that 'tail' of the list is cut. The following is the algorithm of RPLACD in our scheme. It is noted, however, that the use of RPLACD should at best be avoided, since the execution time of

RPLACD is proportional to the length of linear part of the list on which RPLACD is operated on, not to mention harmful side effects it would cause.

RPLACD[r,s]: This function replaces the CDR part of r by s.

1. If the tag of the word w pointed by r is CDR-NEXT or CDR-NIL then goto step 2,
   else if the tag is INDIRECT then perform RPLACD[w,s]
   else perform RPLACD[r,s] in the non-CDR-coding (traditional) scheme.
   Terminate the algorithm, returning the result of RPLACD operated in this step.
2. Change the tag and the data field of w into INDIRECT and CONS[CAR[r],s], respectively.
3. DelAM[r].
4. Set i <- 1.
5. If the tag of the word at address r-i is CDR-NEXT then goto step 6 else terminate the algorithm.
6. If SerAM[r-i] $\neq$ -1 then RepAM[r-i,i-1]
7. Set i <- i+1 and goto step 5.

4. Considerations for implementation of the associative descriptor scheme

We shall assume that the values of index i and base b are provided by the external logic (possibly registers in CPU). The essential part of our scheme is AM. A practical method for realizing large scale AM which satisfies the requirement given in section 2 is hashing for the following reasons:

(i) Hardware hashing can be performed very fast, mostly with single access to hash table memory (which is realized by conventional random access memory), if the memory is configured to be multi-banks which are accessed in parallel [10].

(ii) AM need only be a single-hit associative memory, which is suited to hashing.

In case that the number of descriptors to be entered to AM exceeds the capacity of AM, we have following choices;

(a) to abort the computation if the load factor of AM exceeds the prespecified value; say, 0.85 - 0.95,

(b) to make overflow area in main memory (RM) and prepare for possible decrease in efficiency if AM is overflowed,

(c) to trigger garbage collection and reclaim unused descriptors.

Method (c) can be combined with either method (a) or (b). In conjunction with (c), we should note that all the descriptors of lists may be deleted, if necessary, since they only decrease the efficiency of element referencing.

None of these methods poses difficulties to hash associative memories. It is of course necessary that certain number of look-ups of AM should give rise to the condition to trigger the garbage collection before the hash search turns into a practically exhaustive search on AM (or RM) if method (b) is used.

5. Concluding remarks

We showed that in the associative descriptor scheme a memory access with complete range check can be performed without incurring overhead. We applied the scheme to the speed-up of referencing lists and vectors by taking advantage of the address arithmetic.

Our discussion also suggests a method for handling lists and vectors uniformly when a tag for CDR-coding is provided in each element of a vector. Traditionally, there is distinct disciplines on the use of lists and vectors. We can either follow the tradition and use consciously lists and vectors in a different way, or pursue a way for integrated use of lists and vectors. In the latter approach, following considerations are due:

(1) Facilities are necessary for reserving a certain amount of storage for initial 'CONS' in addition to node-wise allocation by repeated application of 'CONS'.

(2) We need to check the tag of the boundary word of a vector to ensure that the structure is not extended by CDR-link, because condition i>u does not always imply actual range violation.

However, once linear structure is constructed and the size is fixed, there is no reason to distinguish between lists and vectors with regards to random access capabilities and the data structures.

As a natural extension we consider following themes for further research; efficient implementation of the tagged architecture and development of a Lisp system based on the associative descriptor scheme.

References

[1] McCarthy, et. al. Lisp 1.5 Programmer's Manual
Cambridge, Mass., MIT Press, 1962

[2] Cambell, J.A. and Fitch J.P. Symbolic Computing with and without Lisp, Conference Record of the 1980 Lisp Conference Stanford, U.S.A., 1980

[3] Fitch, J.P. and Norman, A.C. Implementing Lisp in a high-level language, Software practise and Experience, 1977

[4] Marti, J. B., Hearn, A. C. and Griss, M. L. and Griss, C. Standard LISP Report, UUCS-78-101, University of Utah, 1979

[5] Greenblatt, R., Knight, T., Holloway, J. and Moon, D. The LISP Machine, Report of Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1979

[6] Van Der Poel, W. L. A Manual of HISP for the PDP-9, Technical U., Delft, Netherlands

[7] Keller, R. Divide and Concer: Data Structuring in Applicative Multiprocessing Systems, Conference Record of the 1980 LISP Conference, Stanford, U.S.A., 1980

[8] Suzuki, N. and Ishihata, K. Implementation of array bound checker Proc. 4th ACM Symposium on principles of programming languages, 1977

[9] Cocke. J. and Markstein, P.W. Measurement of program improvement algorithms, Proc. IFIP 80, Tokyo 1980

[10] Ida, T. and Goto, E. Performance of a Parallel Hash Hardware with Key Deletion, Proc. IFIP 77, Toronto 1977

Table 1 Basic operations on AM

SerAM[b]	searches a descriptor with base b. If the descriptor (b u) is found then upper limit u is returned else -1 is returned.
DelAM[b]	removes a descriptor with base b from AM, it exists in AM.
CreAM[b,u]	creates a new descriptor (b u) in AM. When base b already exists, the descriptor is not created.
RepAM[b,u]	creates a descriptor (b u) in AM. When base b already exists the associated upper limit is replaced with u.

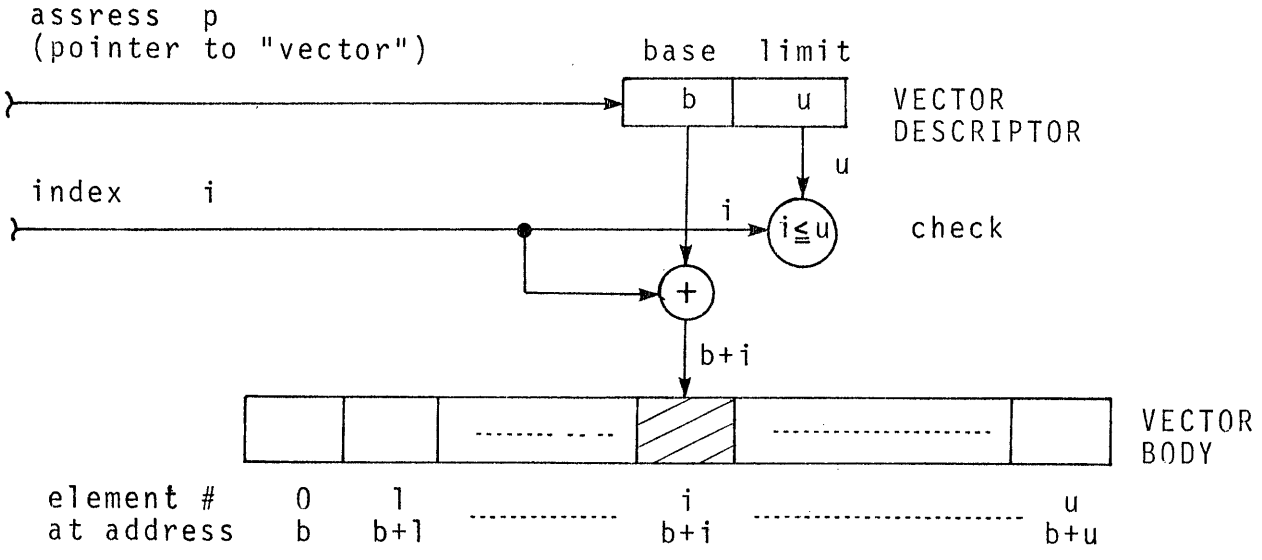

Figure 1-a. Representation of a Vector Using a Descriptor

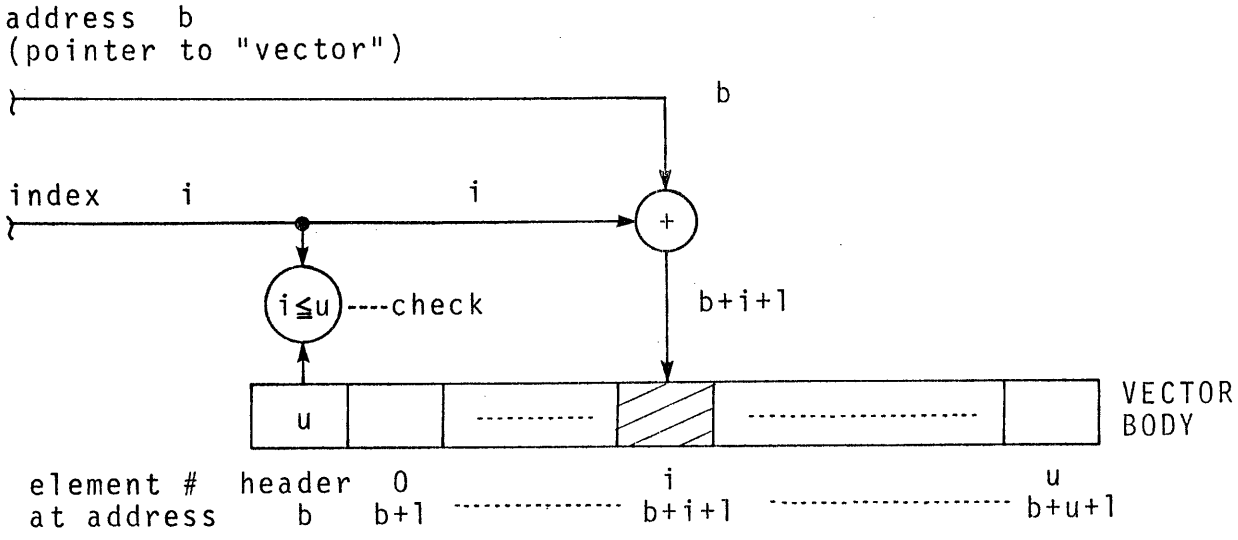

Figure 1-b. Representation of a Vector Using a Header

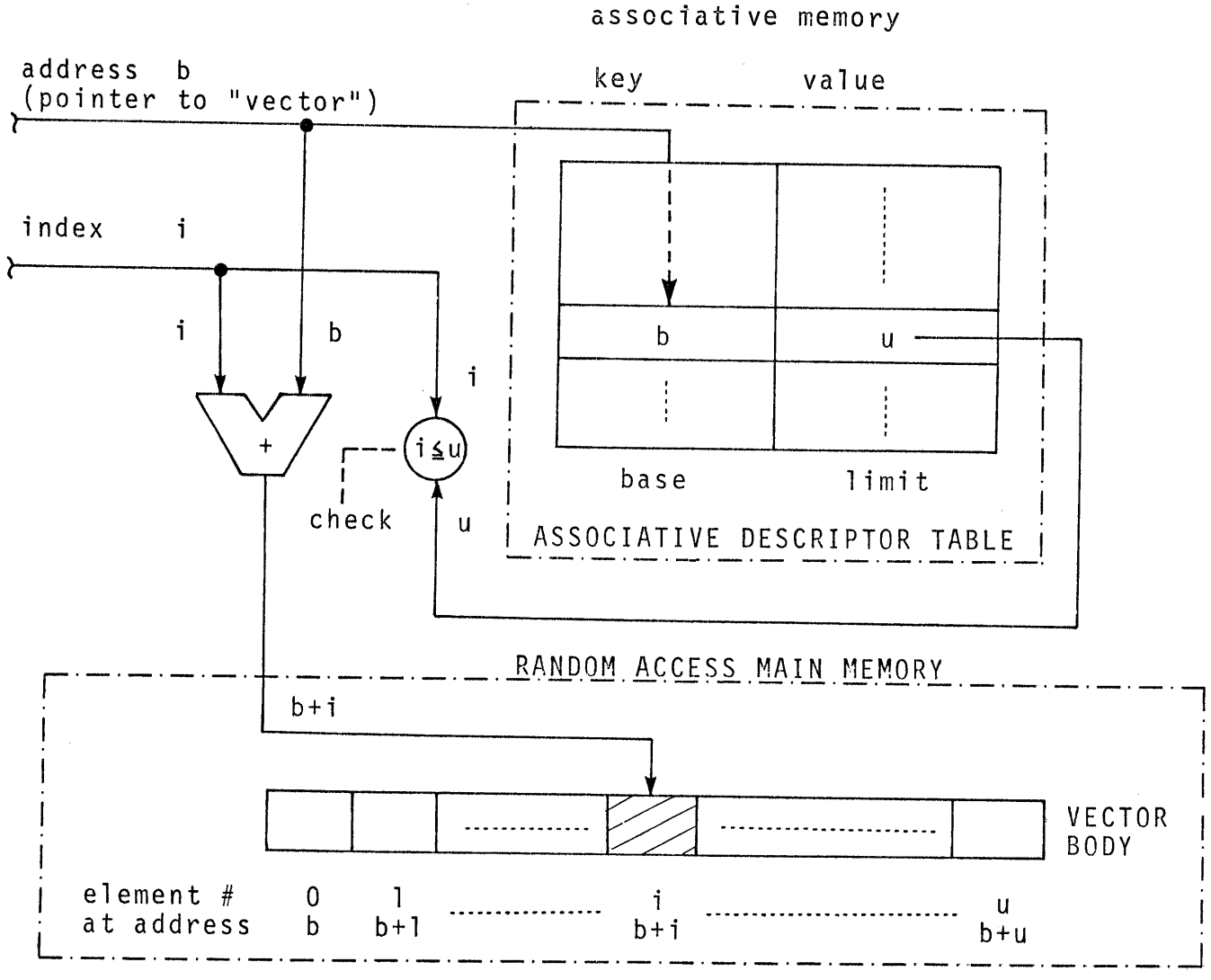

Figure 2. Associative Descriptor Scheme

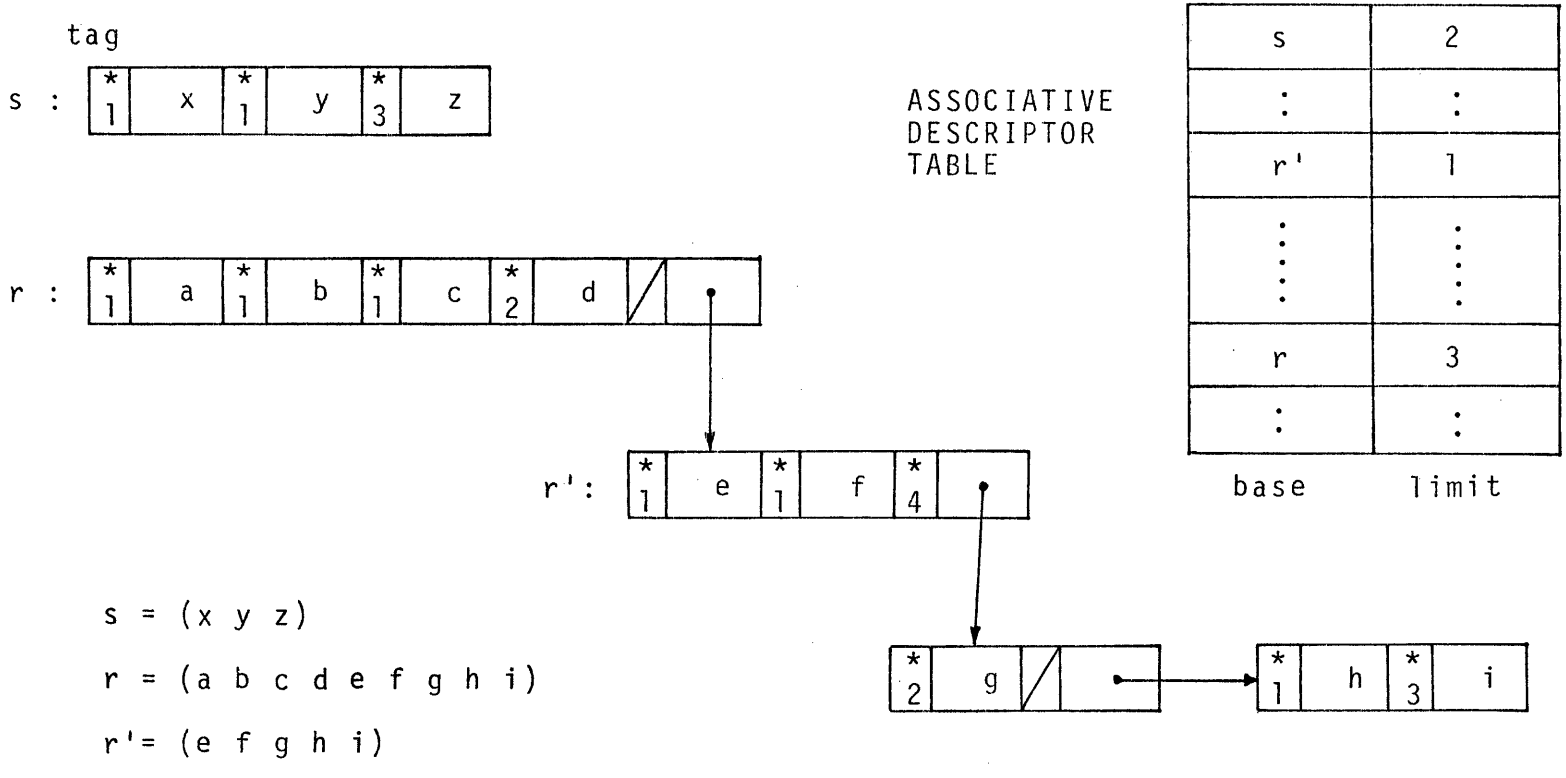

Figure 3. Descriptors for Lists in the Associative Descriptor Scheme

Tag *1 - *4 respectively denote the kind of tags 1 - 4 described at the beginning of section 3.

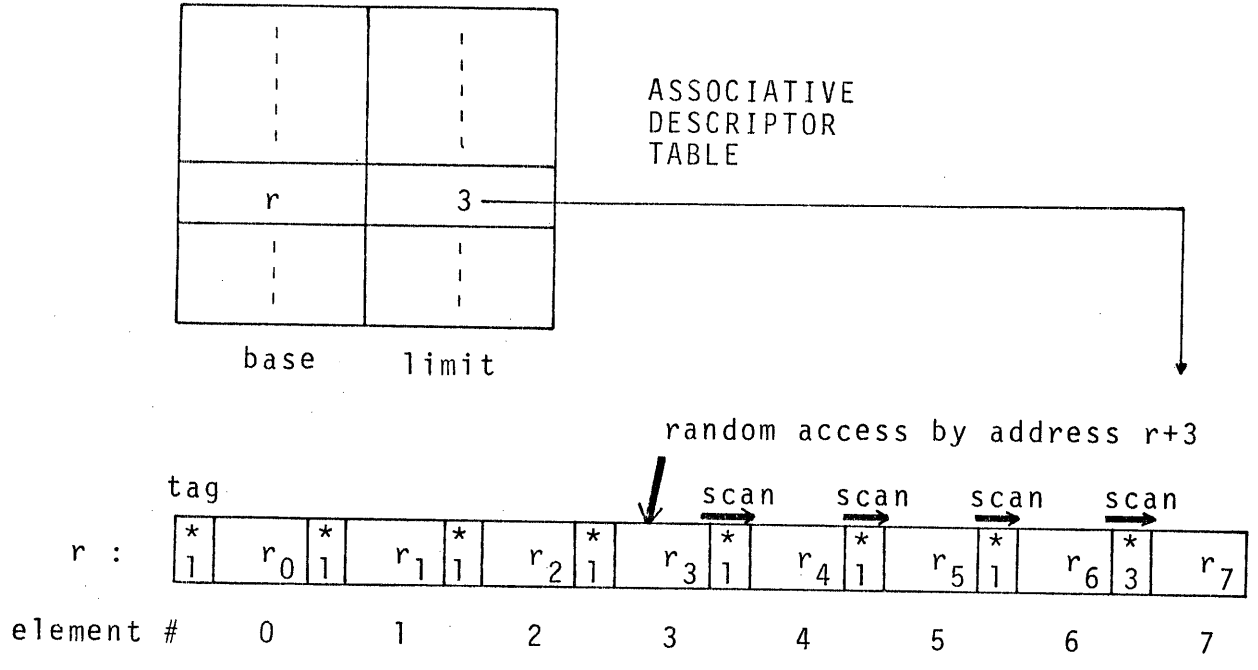

Figure 4. Process of Execution of VECTL[r]

This figure shows the case of k=7 and u=3 in the description of VECTL[r]. The value of i in step 6 gets 4, and after the execution of VECTL[r] the limit of the descriptor is chaged to 7.

Figure captions

Figure 1-a Representation of a vector using a descriptor
Figure 1-b Representation of a vector using a header
Figure 2 Associative descriptor scheme
Figure 3 Descriptors for lists in the associative descriptor scheme
Figure 4 Process of execution of VECTL[r]

Note to the figure 3
Tags *1 - *4 respectively denote the kind of tags 1 - 4 described at the beginning of section 3.

Note to the figure 4
This figure shows the case of k=7 and n=3 in the description of VECTL[r].
The value of i in step 4 becomes 4, and after the execution of VECTL[r] the size of the descriptor is changed to 7.

Acknowledgement

The authors thank Prof. E. Goto and the members of the staff at Information Science Laboratory, Institute of Physical and Chemical Research for helpfull discussions with them on the subjects of Lisp and architectural requirements for Lisp machines.

INSTITUTE OF INFORMATION SCIENCES AND ELECTRONICS
UNIVERSITY OF TSUKUBA
SAKURA-MURA, NIIHARI-GUN, IBARAKI 305 JAPAN

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-80-19

TITLE

Associative descriptor Scheme - for the exploitation of address arithmetic in LISP

AUTHOR(s)

Tetsuo Ida

Kozo Itano

REPORT DATE	NUMBER OF PAGES
December 15, 1980	25
MAIN CATEGORY: computer architecture	CR CATEGORIES: 6.1, 6.34, 3.74, 4.22

KEY WORDS

Lisp, architecture for Lisp, range check hardware, associative memory, hashing

ABSTRACT

We present an associative descriptor scheme to take advantage of address arithmetic in accessing lists and vectors. The scheme employs an associative table for storing descriptors for lists and vectors. In parallel with each access to the structure with its base and index, the associative descriptor is searched for the size of the structure and hardware range check is performed. This operation hence can be done with little or no overhead in accessing the structure. This scheme is straightforwardly applied to fast indexed access to vectors and also to fast rondom access of any list element when combined with CDR-coding. This fact further suggests a method for uniform handling of a list and a vector. Implementation aspect of the scheme using parallel hashing is also discussed.

SUPPLEMENTARY NOTES