CMU-CS-71-103

C.ai -- A LISP Processor for C.ai

by

M. Barbacci
H. Goldberg
M. Knudsen

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

August 9, 1971

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

A special microprogram controlled process designed for efficient interpretation of the LISP language is described. The processor has a fairly large, fast scratchpad memory and uses two cache memories: for the LISP program and data being interpreted; and for the LISP interpreter. Several special purpose registers, small function units, and general byte manipulation capabilities are present.

The approach taken has been to avoid unorthodox implementation schemes and employs little in the way of unusually new (and untried) hardware. Such a conservative approach should enable an implementation in a reasonable length of time.

One of the places where efficiency in list processing (and in most programming applications) can be enhanced is in the ratio of instruction fetches to data fetches. To that end two unusual features were required: writable (up-datable) microcode and recursive control of microcode. With them, it is possible to implement the language interpreter as close as possible to the real hardware machine. Such a machine could also be a "shell" language processor. However, this was not a goal of the design, but a by-product.

The microprogrammed processes include a storage-compacting garbage-collector, which can be made to operate incrementally in parallel with user-program execution. This option avoids interruptions in LISP execution for garbage collection.

TABLE OF CONTENTS

P.LISP DESIGN PHILOSOPHY

In considering a LISP processor for C.ai,[*] two goals were observed in addition to the constraints imposed by the design of the large, overall memory resource and operating system of C.ai. These were: the processor should be implementable in an obvious fashion; and the speed-up in processing of LISP may be borne by a loss in generality of the processor's order code. Although these seem to be either contradictory, or unrelated, the trade-off involved covers the spectrum between a conventional LISP system on a conventional computer and a hard-wired read-eval-print loop. We hope that the following design will lead to a single processor which employs little in the way of unusually new (and untried) hardware, provides a speed-up in the range of 1.5 orders of magnitude over conventional LISP systems, and is not as strictly bound to doing LISP as one might expect. The processor is micro-programmable, with numerous register transfers, byte manipulations, and arithmetic and logical functions (which has been left relatively openended in the design).

In most machines, even those that make extensive use of microprogramming, the mapping from the hardware functions to the order code retains a level of generality unnecessary in C.ai (since different processors are dedicated to different specialized languages or tasks). Hence, the best and cheapest place to begin designing our processor is at the microprogramming level. To that end, two features are necessary for LISP which are not usually available: writable (up-datable) micro-code, and recursive control of micro-code.

[*] It is assumed the reader is familiar with both the C.ai computer (Bell, et al, 1971) and the LISP language (McCarthey, et al, 1962); see Appendix 7 for a brief description of C.ai.

Some implications of such a machine might be that it could also serve as a basis for a processor suited to string manipulation languages and to "shell" languages of manifold capabilities. This, however, was not a goal of the design, but rather a by-product, resulting from these facts:

> The hardware and memory configurations of most computers, and by extension, many assumptions inherent in existing programming languages, are essentially the same.
>
> The capabilities required by a processor to evaluate LISP expressions are powerful enough to handle a large number of other programming language expressions and control structures.

P.LISP: STRUCTURE OF THE PROCESSOR

The first characteristic of the processor is the use of a dual cache system, one of them as the front end of Mp ('Primary Memory' in the PMS Notation, see Bell and Newell, 1971) for the program being interpreted and the other to hold the (most frequently used) microcode of the interpreter. The trade-off between cost and speed here has been discussed in Bell, et al, 1971.

All elementary data items (words) contain local type information permitting data dependent operation. This characteristic is even forced upon microcode words. This is one of the few truly fixed features of the system, the others being:

1. Microcode word interpretation is hardwired and a mixed interpretation of bits in the microword is used: some bits are direct control functions and some are encoded. The distinction is itself dependent on the micro-operation code.

2. Microcode addressing is via a microprogram counter containing the displacement in the microcode addressing space. (See Appendix 1.)

The PMS Diagram of the processor is shown in Figures 1 and 2.

BUSSES

To achieve parallelism, two data transfer busses are provided, L(1) and L(2), controlled by K(1) and K(2). In order to provide the most general byte manipulation capabilities (i.e., a string of contiguous bits located anywhere inside a word) a pair of byte transfer matrix-like switches are provided for each bus. These are described in Appendix 2. The busses, their controllers and switches are identical. However, for economic reasons one of the busses might not have the byte manipulation matrices, forcing us to use the other one for this kind of operation with the subsequent loss of generality.

ALU

The Arithmetic Logic Unit operates asynchronously with the rest of the processor. Its operation is independent of the microprogram speed and is invoked by selecting two operands from the local memory to be loaded into busses L(3) and L(4) and by selecting a function via K(3). After this is done, the micro processor proceeds in parallel while the selected function is being carried out. The output of the ALU may be transferred to the local memory via S(5) and any of the busses, and this transfer is done after a 'safe' number of microcode steps have taken place. Unfortunately this requires a minimum of two microsteps for even the simplest operation.

Each bus latches under the control of a microcommand. The reason for this is obvious in the case of the ALU busses L(3) and L(4) and may become clear when we consider some of the specialized registers in the local memory and their asynchronous operation.

PROCESSOR STATE

This is the set of registers used as scratchpad by the microprocessor. Some are full-word registers holding temporary data. Some are dedicated to frequently used data and some others are actually very specialized units (e.g., SAV CONTROL REGISTER) as described in Appendix 3.

P.LISP: ISP

The main criterion for specialized list processor design is to minimize the ratio: instruction fetches/data fetches. According to this criterion, the machine has two basic modes:

Plex Mode - This corresponds to our intuitive concept of a microprogrammed language, i.e., the execution of microcoded routines selected by the OP Code field in the macroinstruction.

Lisp Mode - This is equivalent to the EVAL function in LISP, where the interpretation of the input string is carried out by successive (possibly recursive) invocations of more primitive operations, without the overhead of instruction fetches and interpretation. In other words, we place the interpreter mechanism directly in the hardware.

The idea then is to provide a set of microroutines that implement enough primitives for plex processing and provide them with the possibility of calling each other using the stack mechanism; in other words, the microcode is recursive.

A set of microroutines, written in a register transfer language, can be found in Appendix 4, together with a detailed description of some of the data primitives that may be useful for our purposes. Appendix 5 presents the microcommands in detail.

Although the register transfer language and the microcode selected are not in a one-to-one relationship, the mapping is direct. For instance, the RT language does not show parallelism, which is achieved in the processor.

THE LISP WORLD

FUNCTIONS OF THE P.LISP OPERATING SYSTEM, OS.LISP

1. Acts as liason between the LISP interpreter (microcoded EVAL) and AMOS (A Minimal Operating System, see C.ai, Bell et al, 1971).

2. Handles multiprogramming of P.LISP (swapping, paging, relocation, memory allocation, etc., in cooperation and/or contention with AMOS).

3. Provides _choice_ of LISP version for each user, i.e., gives options of compacting or parallel garbage collection, compact lists, etc., in cooperation with AMOS.

4. Handles I/O, vis-a-vis the local file system and the ARPA network.

5. Controls allocation and "flipping" of semi-spaces, when compacting garbage collection is used. See Appendix 6.

6. Controls alternation of user and garbage collector, compute optimal length at time slices, and provides recovery procedures, when parallel garbage collection is used (see below).

GARBAGE COLLECTION -- SPECIAL METHODS

Garbage collection procedures will be microcoded. Conventional garbage collection (GC) is possible, using one dedicated bit in each word. However, if either paging or swapping is used (which is probably unavoidable, with the projected number of users), _storage-compacting_ GC (CGC) is preferable (Appendix 6). Fenichel and Yochelson (1969) give a simple recursive algorithm and several reasons for CGC: reduction of page faults, less core image to swap, and no time-consuming conventional linear sweep through large memory space. We have also adapted Cheney's

(1970) non-recursive scheme to LISP. Both divide data memory into two equal "semi-spaces", which doubles the (virtual) memory required, although a special paging scheme we designed reduces "real" memory needs (see Appendix 6, sec. 2.5). Some other advantages of CGC are:

1. Cache memory is more likely to give look-ahead, since CDR(L) is usually the next memory word.

2. Free-storage list is a linear block of words, so any subsequently-formed lists will likely be localized in memory.

3. Parallel GC is made possible -- see below.

PARALLEL GARBAGE COLLECTION

Delays of a second or more (much more, with large memories) while LISP garbage-collects may be intolerable in real-time applications such as speech or robotics. An alternative is an incremental garbage collector, time-shared with the user's program so that his job runs a bit slower, but never stops completely for more than a small time quantum. Time-slices on the order of this quantum are alternated between user computation and data-salvaging (garbage collection).

That is, free storage in the current semispace is alternately used to create new S-expression (user) and to copy still-active structures from the previously-used-up semispace (GC). If copying is complete when current semispace runs out, then semispaces are "flipped" as in non- parallel CGC (see Fenichel and Yochelson, 1969). Otherwise some special "bail-out" procedure must be executed to finish copying from the old semispace so it can be used as free storage again. Bailing-out causes a conventional GC waiting period.

While we have a bail-out procedure, the P.LISP Operating System tries to avoid the latter situation by balancing the relative times allotted to user computation and GC. We propose some adaptive heuristics for this in Appendix 6.

Some LISP primitives must be slightly modified to avoid strange effects on lists which have parts in both semispaces, not yet having been copied over into current semispace. Since these modifications reduce efficiency of the system, use of the parallel-GC version of LISP should be an option of each user.

COMPACT-LIST STRUCTURE

With storage-compacting garbage collection, we could eliminate the CDR field and assume the CDR of a list cell is the next word in memory (see Hansen, 1969 and Cheney, 1970). A special LINK word pointing to the true CDR must be inserted whenever this is not the case.

Such a scheme has two advantages -- look-ahead and saved space through elimination of the CDR and AM2 fields (26 bits). (See Appendix 4 for cell description.) We already have the former if CGC is used on conventional LISP structure. The latter advantage is realized if we can fit one compact-list cell into a halfword; such an implementation might be a worthwhile option.

SPECIAL HARDWARE/MICROPROGRAMMED SPEED-UP FEATURES

Property-list (PLIST) search speeds may be increased as follows:

Frequently-referenced property names are assigned integer indices according to their location in the processor's property descriptor table (PDT, Appendix 3 and Fig. 3). Attribute-value pairs which would otherwise be kept on PLIST in conventional two-word form (Fig. 4) may, if the attribute is in the PDT, be reduced to the single type-PLIST word of Fig. 5, where only the 6-bit PDT index of the attribute is needed. The separate type on such cells allows conventional dotted-pair cells to be intermixed with them in a PLIST. Retrieval of a value from PLIST is as follows: Given attribute "Att",

(a) If Att is type immediate integer, assume it's a PDT index, skip to (c). (Note that the system refers to certain properties (pname, fexpr, apval, etc.) directly by their fixed PDT numbers.)

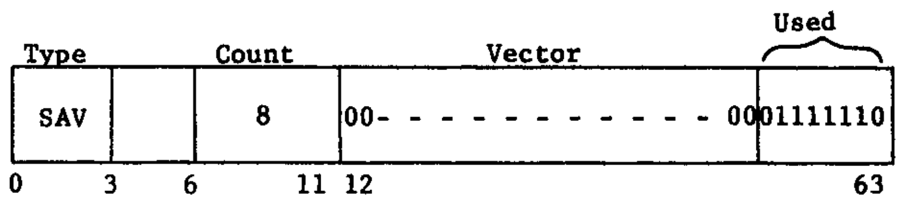
(b) If Att is type atom, search PDT for same atom. If found, set Att to its table index; else go to (d).

(c) Search PLIST, checking only cells of special type PLIST, for PN field equal to Att. If found, return CAR of this cell; else NIL for "not in PLIST".

(d) Search PLIST, checking only conventional (type-LIST) cells in usual manner; return value found, or NIL.

Likewise, when putting a new pair on a PLIST, the PDT is consulted to determine whether a type-PLIST or conventional structure is added. This can halve data-memory fetches.

Often one accesses an element in a complex S-expression whose position in that structure is <u>known</u>, e.g., by "(CADDR (CDDDDR (CAR L)))" for the 7th item in (CAR L). We can encode such a succession of CAR's and CDR's into a Boolean vector (0 for CAR, 1 for CDR), along with a count of how many bits are meaningful, into a structure-access vector (SAV), e.g.,



for the above example. Note that Nth-element access is a special case of this. LISP could now allow up to 52 A's and D's between C and R! We provide a dedicated hardware register for SAV interpretation, as described in Appendix 3.

Function-call arguments are passed to the function in the A registers, one per register, in order of appearance in the calling expression. This is not new, but the fact that we can work with the A registers at microprogram speed might give us some gain.

EXTENSIONS TO LISP

The generality of our processor hardware, plus the writable microcode memory, allows extra data types (structures) and their appropriate operations to be added at no cost (aside from writing additional microcode).

While these data structures could be used to extend LISP (as in LISP 2), they can just as well be embedded in the conventional LISP 1.5 linguistic framework. LISP's function-oriented syntax can easily support the predicates, selectors, and constructors needed to create and use such types.

Note that such additions affect neither the syntactic elegance nor the execution speed of LISP, in no way penalizing those who don't use them. LISP users have often created their own specific data forms and written LISP functions to handle them. But a sophisticated user of our P.LISP could translate these functions into microcode, for large speed advantages.

Some data types of general interest might be strings and arrays or plexes (blocks of arbitrary number of contiguous storage words). String processing is aided by our general byte-transfer operation. SNOBOL4's variable-binding semantics are identical to LISP's. Plexes are possible if we use storage-compacting garbage collection -- these could lead to an extensible data-type facility akin to SNOBOL4's DATA statement. Users might even be able to specify subfields of the plex words, e.g., L^6 (Knolton, 1966) thus getting more use out of our byte-transfer operation.

Note that a "LISP compiler" in this system might generate microcode for insertion into control memory, under user commands. Of course, the kernel LISP 1.5 code would reside in a write-protected section (from user's viewpoint).

A <u>compact-list</u> (no CDR fields; CDR of a word is the next storage location) version of LISP can also be microcoded, where data storage space is tight.

Floating-point (real) arithmetic is important for image processing, etc., and we would initially microcode the basic operations. However, hardware floating-point is faster than microcode and should not be ruled out at this point. Trig, log, hyperbolic, and other special functions can be added where demand exists.

PERFORMANCE CONSIDERATIONS

A basic ratio of x25 speed-up in memory fetch from cache versus core will be assumed here. (It has been suggested by the Stanford AI processor that a 2K cache is 95% effective for conventional LISP 1.6.) Our micro-program cache should have a high hit ratio since programs are mostly sequential and will not fragment very badly. With compacted lists, our data-cache should have a high hit ratio as well.

The speed of the microprocessor will not (short of extremely expensive and possibly unreliably new hardware) keep up with a 40 nsec. cache cycle time, but will certainly fall in the range of 100 nsec. per instruction cycle. This alone will give microprogram-implemented LISP primitives a speed-up of x15 to x40 over good conventional systems like LISP 1.6* (assuming a single processor).

The organization of the microprogrammed primitives, compacted lists, parallel garbage collection, and tighter property lists for atoms should yield further speed-up. To specify, with any accuracy, a factor of increase is rather difficult since the profile, over execution, of a typical LISP program varies considerably from program to program. A significant amount of processing time is spent in binding formal to actual parameters of functions. (This is where compiled functions pay off.) But this varies, as is indicated by a range of compiler speed-ups of x1 to x15 (in our experience). Even though we have compared our design to interpreted rather than compiled LISP, we have left open the possibility of doing the same sort of binding decisions that conventional LISP compilers perform.

*LISP 1.6 is the LISP implementation on the PDP-10.

Moreover, the user can compile some functions into microcoded routines.

Preliminary coding of some important LISP primitives indicates that we can gain very little over LISP 1.6 (PDP-10) by trickier programming of the evaluation algorithm (given our conventional approach). It is our opinion that the implementation of LISP 1.6 is good, and the PDP-10 order code is appropriate enough for LISP, so that we cannot do much better than the order of magnitude improvement provided by the microprogramming. It is our hope that the flexibility and reasonable simplicity of this design will more than outweigh the factor of not having a single LISP processor one hundred times faster. Multiple P.LISP's, of course, easily allow us to reach the goal.

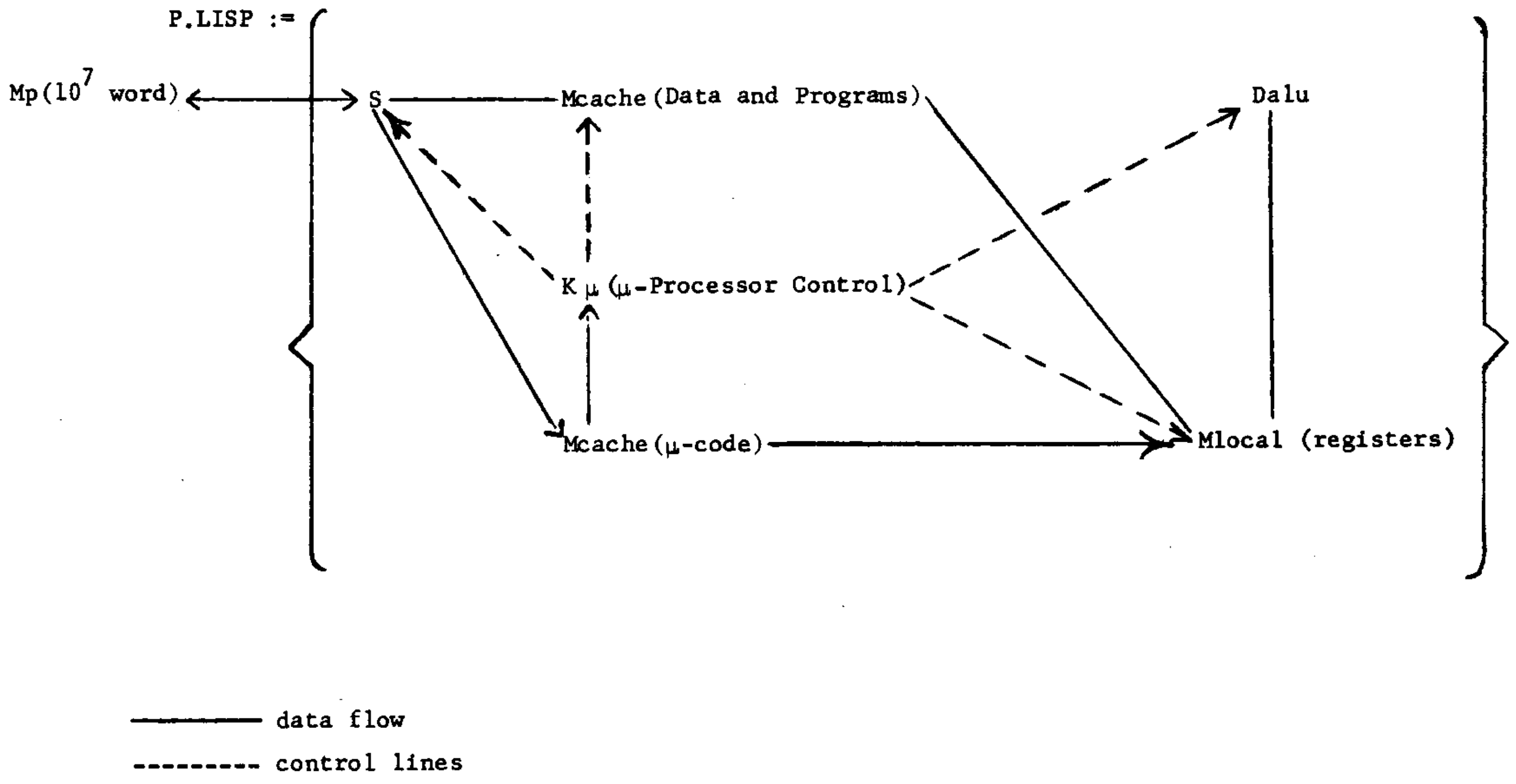
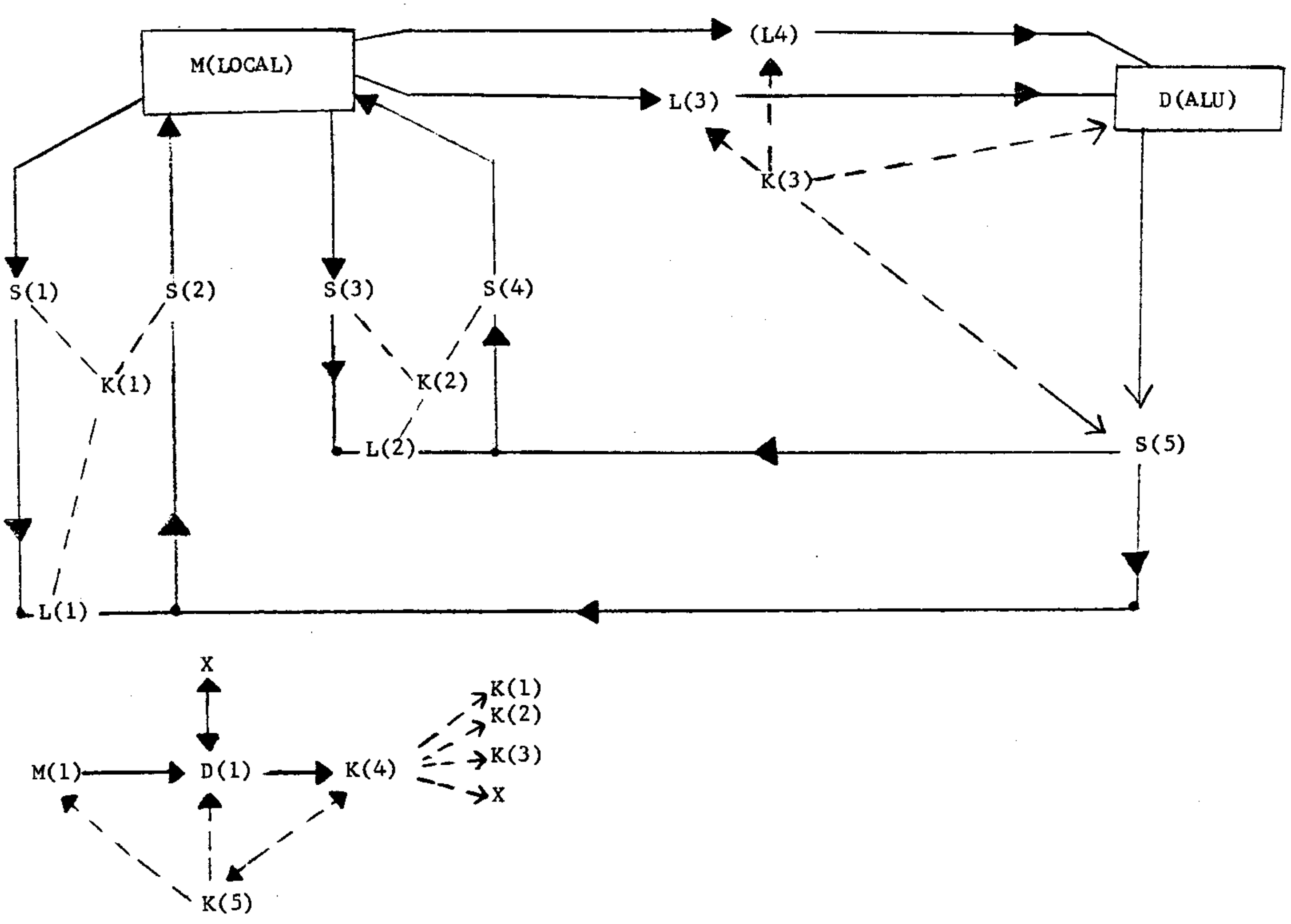


Figure 1: PMS diagram of P.LISP and its caches



S(1) input byte-transfer matrix; S(2) output byte-transfer matrix; K(1) byte-transfer controller.
S(3), S(4) and K(2) as S(1), S(2), K(1).
L(1) bus #1; L(2) bus #2.
L(3) ALU input bus #1; L(4) ALU input bus #2; S(5) ALU output-bus selector;
K(3) ALU controller.
M(1) microprogram cache; D(1) micro-word decoder; K(4) micro-program controller;
K(5) micro-program sequence controller (clock).

Figure 2: PMS diagram of P.LISP

Figure 3: Property Descriptor Table (PDT)

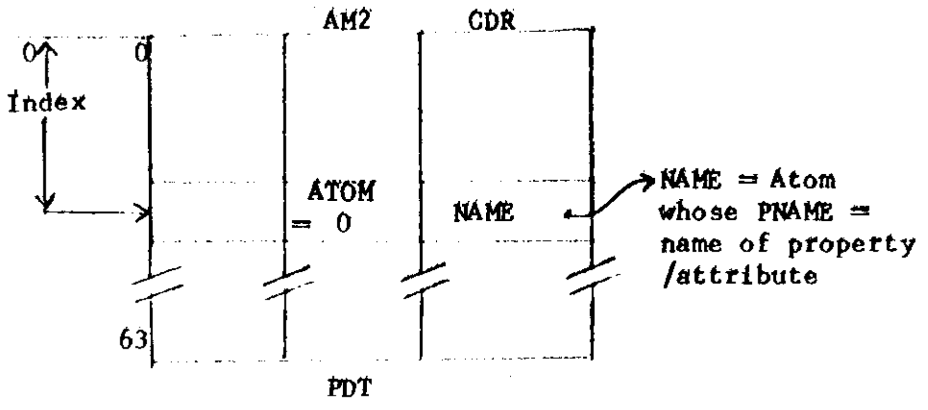


Figure 4: Conventional PList pair element

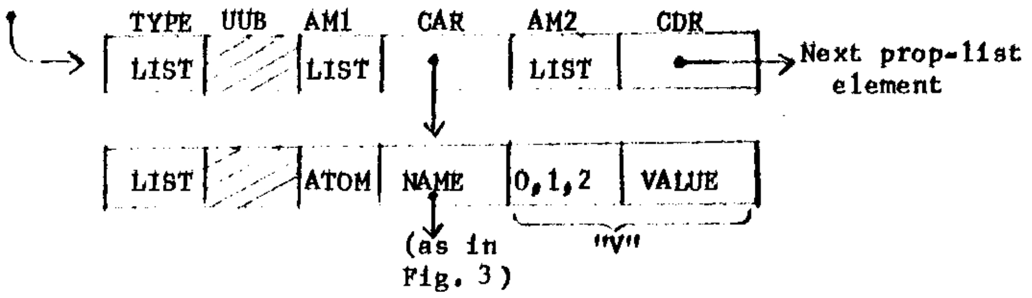
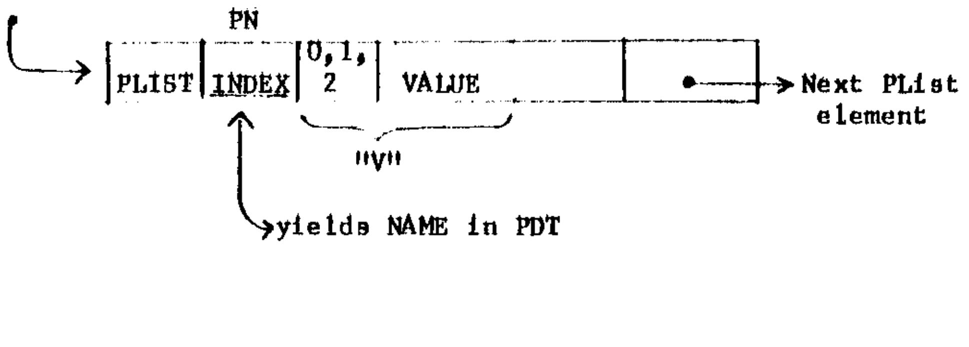


Figure 5: Our single-word PList element



Note: Value "V" may be list, atom, or immediate atom, in either scheme.

ANNOTATED BIBLIOGRAPHY OF COMPACTING GARBAGE COLLECTION

1. Cheney, C. J., "A Non-Recursive List-Compacting Algorithm", CACM 13, 11 (Nov. 1970) 677-678.

   "THE" CGC algorithm of choice, where recursion is not wanted. Much simpler than (4). Intended for CLS, but easily extended to SLS [Appendix 6, Sec. 2.4], with hints from (3).

2. Hansen, W. J., "Compact List Representation: Definition, Garbage Collection, and System Implementation", CACM 12, 9 (Sept. 1969) 499-507.

   "THE" seminal paper on CLS. All the right ideas, but refused to permit series links, thus ruling out "RPLACD" ("RN" in L*). Also his CGC requires recursion and has inelegant "fixup table" kludge for re-entrant lists. Predicts values of CGC even for SLS (page fault reduction, I/O to M.sec, etc.). Much emphasis on bit-packing 360 implementation.

3. Fenichel, Robert R. and Jerome C. Yochelson, "A LISP Garbage Collector for Virtual-Memory Computer Systems", CACM 12, 11 (Nov. 1969) 611-612.

   "THE" CGC to use on standard LISP where recursion is o.k. Points out inefficiency of conventional GC in huge [virtual] memories, as well as page-fault reduction of compacting GC. Also suggests calling CGC before space exhaustion, if "thrashing" gets bad due to scattered nature of active data. Uses CAR field to indicate already-copied.

4. Schorr, H., and W. M. Waite, "An Efficient Machine-Independent Process for Garbage Collection in Various List Structures", CACM 10, 8 (Aug. 1967) 501-506. Also in Knuth, Vol. 1, 417-419.

   Of interest for being first GC requiring neither recursion nor linear sweep of M. Non-compacting, no good for compact lists, and requires one dedicated bit per word.

5. Ross, Douglas T., "The AED Free-Storage Package", CACM 10, 8 (August 1967) 481-492.

   A case study in solving a problem (storage allocation for and garbage collection of various-sized blocks (plexes)) in the most complex way possible in the B.C. (Before Compacting) era. Shows how hard plexes are to implement without a storage-compacting garbage collector, since free storage fills up with odd-sized "holes".

6. Bell, C. G., P. Freeman, et al, "C.ai: A Computing Environment for AI Research", Department of Computer Science, Carnegie-Mellon University, (May 1971).

7. J. McCarthy et al, "The LISP 1.5 Programmer's Manual",MIT Press, Cambridge, Mass. (1962).

8. Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill (1970).

9. Knolton, K. C., "A Programmer's Description of L6", CACM 9, 8 (Aug. 1966).

APPENDIX 1

MICROCODE ADDRESSING

Microcode is stored in main memory and loaded into a particular processor (i.e., the microprogram cache associated with it) by C.amos*. Thus, the microinterpreter is general purpose in the sense that data paths, microwords and microcommands depend only on the language we are implementing (L*, LISP, SNOBOL, etc.).

By having the microcode residing in Mp, it is possible to modify it at will and fast (of course, some access privileges will have to be associated with given code). It may be necessary to implement a microassembler as the first step (later it can be bootstrapped to a higher level language) to facilitate the coding of microprograms.

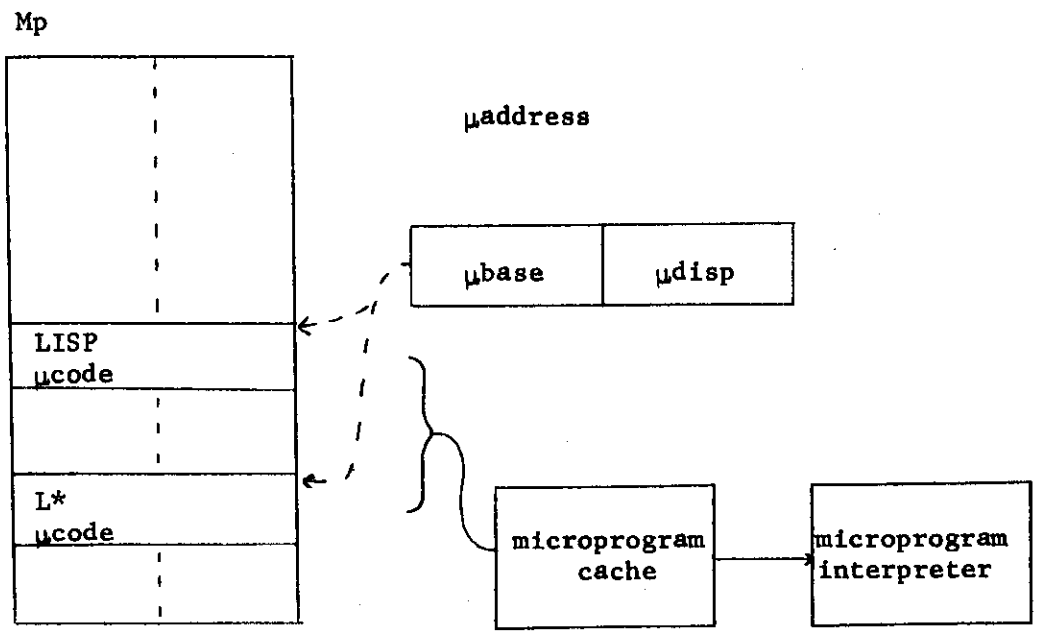


Figure 1.1

*See C.ai (Bell et al).

Microbase is provided by AMOS and this is the way we "assign" a particular language to a processor. Microdisp (displacement) is the "local" address inside the microcode corresponding to a given language. A small displacement is economical but restricts the size of the microprogram (possible new and complex languages that require large pieces of microcode may be impossible to implement at this level); on the other hand, a large displacement is flexible but wasteful.

APPENDIX 2

BYTE-TRANSFERS

Two methods for selecting a byte are presented; both are based on matrix-like transfer networks, the difference being in the trade-off between the cost (number of gates) vs. the number of bits in the micro-word to control the transfer. A general shift matrix should also be explored. The methods are based on the fact that byte transfers do not affect the relative ordering of bits, i.e., lower order bits go to lower order bits. A simple way to do the operation is based on a matrix-like transfer network, where the only gates that are activated are the ones aligned along a selected diagonal. Besides the selection of the byte, a mask must be prepared to enable the proper gates in the output register. A method to prepare the mask is explained later.

METHOD 1

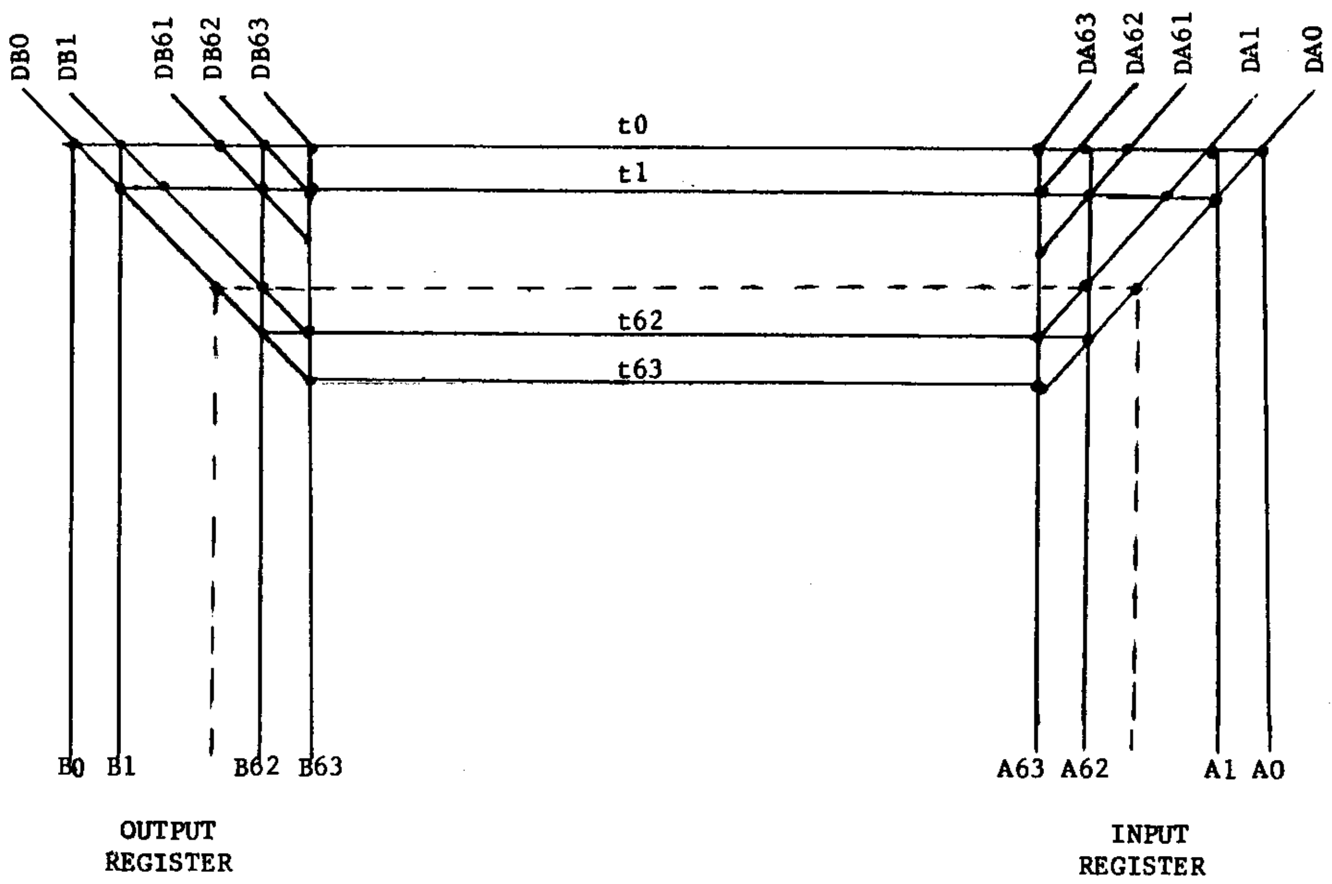


Figure 2.1

DAi is selected according to the initial bit (the ith) of the input byte. The byte is sent via lines t0 - tk where k+1 is the byte length. DBj is selected according to the initial bit (the jth) of the output byte.

<u>METHOD 2</u>

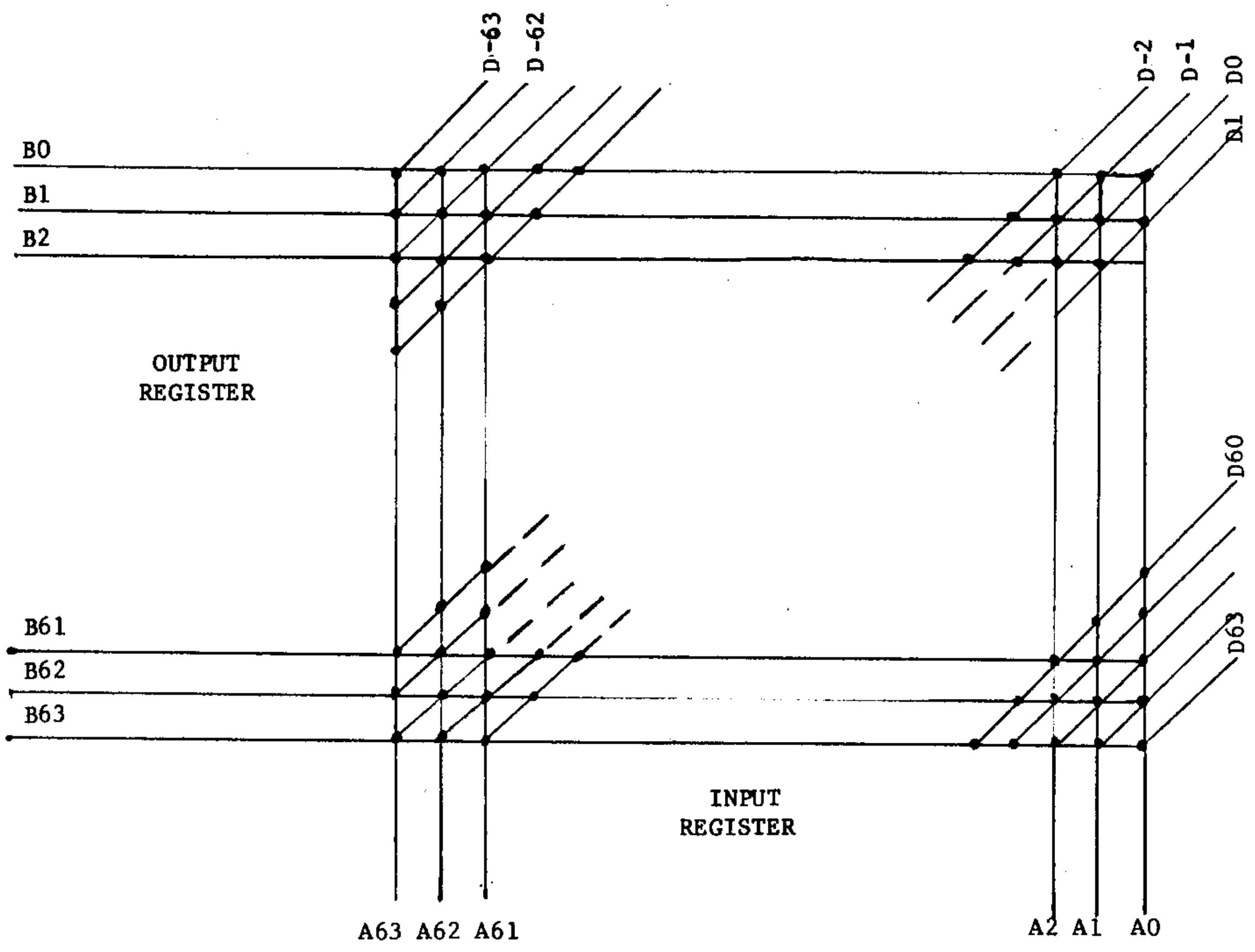


Figure 2.2

Di is selected according to the initial bit positions of both the input and output bytes, according to the following rule: Given Ak and Bj as initial bit positions, the transfer is controlled by Di where i = j-k (actually they do not need to be the initial bits, any two corresponding bit positions will do).

The selected byte appears on the proper lines without further selection as in Method 1.

COMPARISON

	Method 1	Method 2
Cost (number of dual input AND gates)	4160	4096
Diagonal Selection	DAi and DBj are given by initial bit position	Dk is given by subtraction of initial bit positions
Enabling mask for output register	Given by initial and final bit positions at output byte	Same as Method 1

Table 2.1

The fields needed to specify a byte transfer (besides the register's selection) are:

-- initial bit of input byte (6 bits)

-- initial bit of output byte (6 bits)

-- final bit of output byte (6 bits)

The subtraction operation in Method 2 can be avoided if we encode in the microword the diagonal number instead of the initial bit of the input byte, but this encoding requires 7 bits (there are 127 diagonals in Method 2) and this price can be too much to pay for the saving of 64 dual input AND gates in the matrix. Our feeling is then that Method 1 provides the fastest solution, with the smallest microword waste at a cost of 64 additional gates (which may very well be the cost of the arithmetic unit to perform the subtraction in Method 2).

DETAIL OF CROSSPOINT

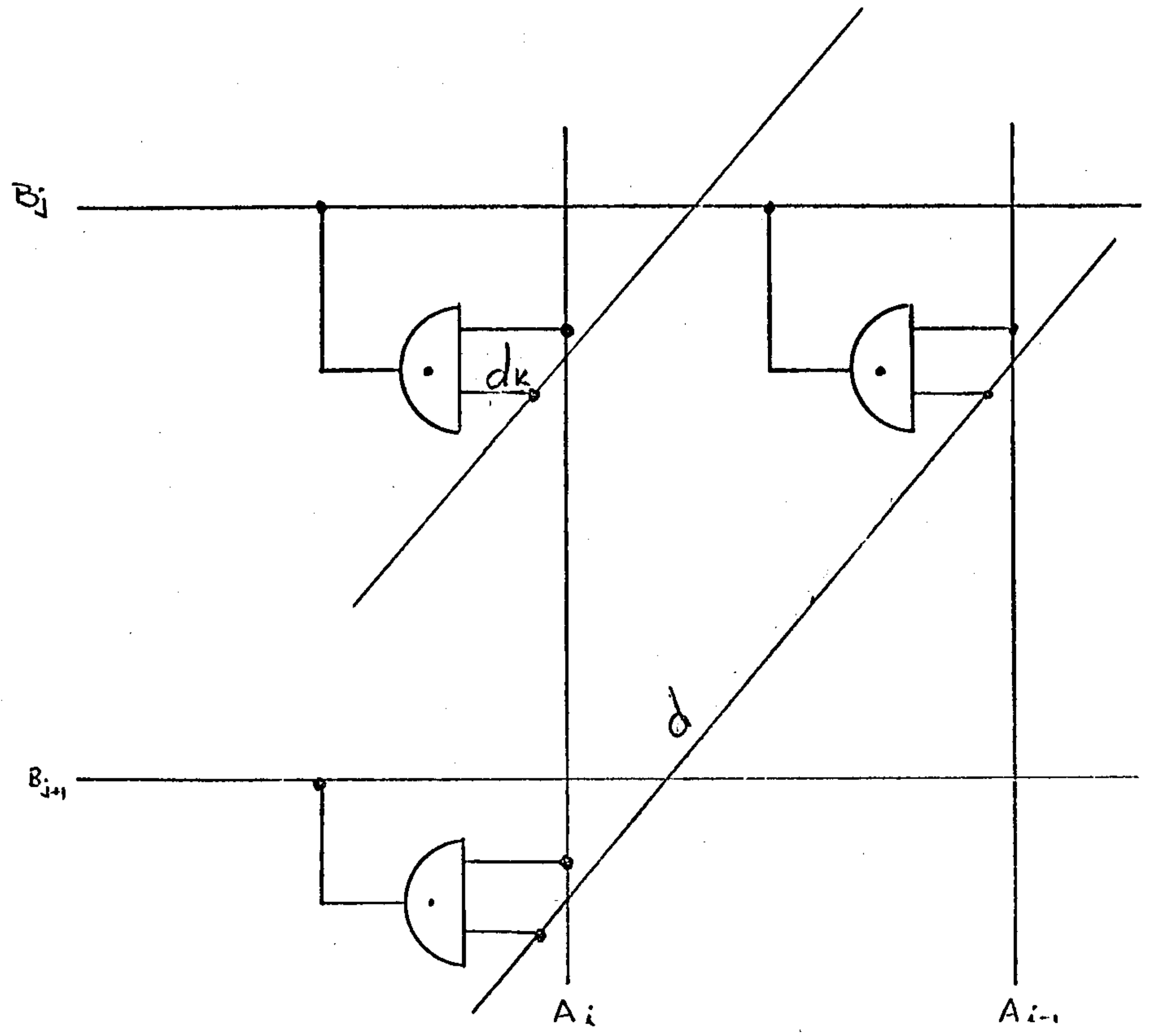


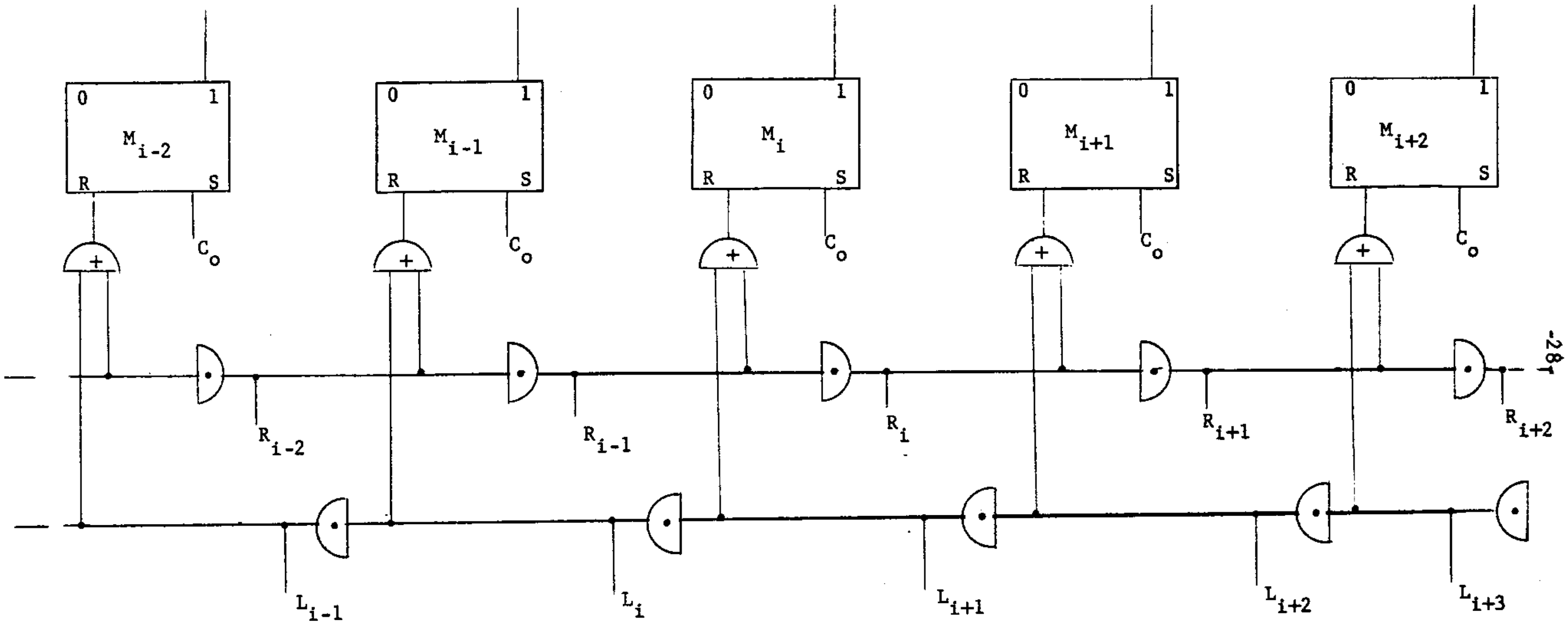
Figure 2.3

CREATING A BYTE TRANSFER MASK

Two signals coming out of two 1/64 decoders are used to mark the initial and final bits of the mask (they may be the same bit). The circuit shown in Figure 2.4 is a two-way simultaneous ripple propagation network. The mask is selected by two signals, L and R from the decoders. L[K] resets all bits to the left of bit K and R[J] resets all bits to the right of bit J. Clearly the mask can be a single bit.

If the propagation delay proves to be intolerable, some cary-look-ahead scheme can be provided, although it does not need to be as complex as a full adder carry look-ahead unit; a few OR gates will do.

MASK BITS



C_o is used to set the mask to 1's (first step)

R_j means that bit J is the leftmost bit of the byte

L_k means that bit K is the rightmost bit of the byte

(All 'AND' gates can be replaced by diodes.)

Figure 2.4

APPENDIX 3

```
M(LOCAL)
--------

MAR/MP-ADDRESS-REGISTER<0:23>
MBR/MP-BUFFER-REGISTER<0:63>
UPC/MICRO-PROGRAM-COUNTER<0:19>
        TYPE<0:3> := UPC<0:3>   %<ALWAYS 'UPC'>%
        UDISP/UDISPLACEMENT<0:15> := UPC<4:19>
UIR/MICRO-INSTRUCTION-REGISTER<0:63>
STACK<0:63>
R1<0:63>
R2<0:63>
R3<0:63>
T1<0:63>
T2<0:63>
T3<0:63>
PDT/PROPERTY-DESCRIPTOR-TABLE[64]<0:27>
        TYPE<0:3> := PDT<0:3>   %<ALWAYS 'PDT'>%
        PTR<0:23> := PDT<4:27>
ODT/OPERATOR-DESCRIPTOR-TABLE[64]<0:19>
        TYPE<0:3> := ODT<0:3>
        PTR<0:15> := ODT<4:19>
SAVCR/STRUCTURE-ACCESS-VECTOR-CONTROL-REGISTER<0:63>
        TYPE<0:3> := SAVCR<0:3> %<ALWAYS 'SAV'>%
        CNTR<0:5> := SAVCR<6:11>
        VECTOR<0:51> := SAVCR<12:63>
                LASTBIT := VECTOR<51>
IAR/INDIRECT-ADDRESS-REGISTER<0:7>
F[16]<0:63>
A[16]<0:63>
```

STACK IS THE TOP OF THE PUSH DOWN STORE. IT WORKS IN A FUNNY WAY: ANY TRANSFER TO 'STACK' IS AN ACTUAL PUSH DOWN AND ANY TRANSFER FROM 'STACK' IS A POP UP OPERATION.

PDT CONTAINS POINTERS TO THE ATOMS THAT DESCRIBE A SET OF PROPERTIES. SOME OF THEM WILL BE PREDEFINED BY THE SYSTEM (SYSTEM ATOMS) AND THE REST ARE USER DEFINED.

SAVCR (FIG. 3.1) IS A SPECIAL FUNCTION UNIT. IT CONSISTS OF A TWO WAY MOD(64) COUNTER AND A 52 BIT SHIFT REGISTER. THE VECTOR FIELD BEHAVES LIKE A STACK (ALTHOUGH THE UNIT ITSELF CAN BE LOADED/ UNLOADED IN PARALLEL), THE TOP BEING THE RIGHTMOST BIT, AND COUNTER KEEPING TRACK OF THE DEPTH. THIS UNIT IS USED TO HANDLE STRUCTURE ACCESS VECTORS AS DESCRIBED IN APPENDIX 4.

IAR IS USED AS AN INDEX REGISTER POINTING TO M(LOCAL). IS USED WHEN SCANNING THROUGH PDT AND ODT, AND IS IMPLEMENTED AS A MOD(256) COUNTER.

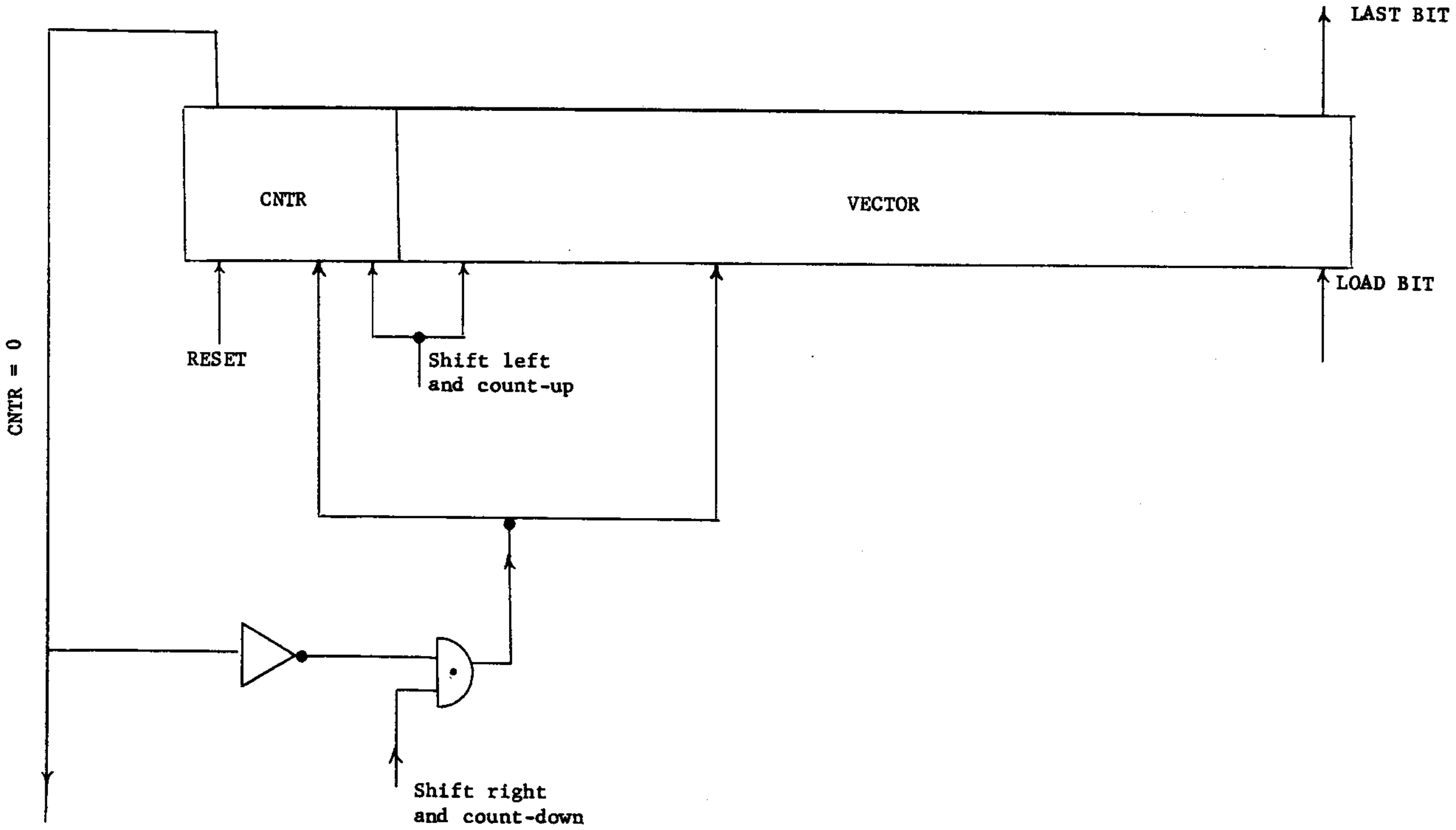


Figure 3.1
Structure Accesss Vector Control Register

APPENDIX 4

SOME DATA PRIMITIVES AND MICROPROGRAMMED ROUTINES:

DEFINITIONS OF SUBFIELDS IN THE WORD. SEE FIG. 4.1

TYPE<0:3>	:= CELL<0:3>
GC<0:1>	:= CELL<4:5>
UUB<0:5>	:= CELL<6:11>
AM1<0:1>	:= CELL<12:13>
CAR<0:23>	:= CELL<14:37>
AM2<0:1>	:= CELL<38:39>
CDR<0:23>	:= CELL<40:63>

UUB ARE THE UNIMPLEMENTED USER BITS.
THE ADDRESS MODES (AM1 AND AM2) ARE DEFINED AS FOLLOWS:

0	ATOMPOINTER (POINTS TO PROPERTY LIST)
1	LISTPOINTER (POINTS TO CELL)
2	LITERAL (INMEDIATE DATA)

SUBFIELDS IN A PROPERTY LIST.ELEMENT. SEE FIG. 4.2

TYPE<0:3>	:= CELL<0:3>
GC<0:1>	:= CELL<4:5>
PN<0:5>	:= CELL<6:11>
AM1<0:1>	:= CELL<12:13>
VL<0:23>	:= CELL<14:37>
AM2<0:1>	:= CELL<38:39>
NP<0:23>	:= CELL<40:63>

SUBFIELDS IN A TYPE POINTER

TYPE<0:3>	:= CELL<0:S>
GC<0:1>	:= CELL<4:5>
AM2<0:1>	:= CELL<38:39>
ADDR<0:23>	:= CELL<40:63>

SUBFIELDS IN A STRUCTURE ACCESS VECTOR. SEE FIG. 4.3.

THE STRUCTURE ACCESS VECTOR IS USED TO FOLLOW A PATH OF CAR'S AND CDR'S BEGINNING AT A GIVEN CELL OF THE PLEX.
THE COUNTER CONTAINS THE NUMBER OF SIGNIFICANT BITS IN THE VECTOR FIELD.
INDIVIDUAL BITS IN THE VECTOR FIELD ARE TESTED TO DETERMINE WHETHER TO TAKE THE CAR(0) OR THE CDR (1) OF THE CELL .
THE SUBFIELDS ARE:

TYPE<0:3>	:= CELL<0:3>
GC<0:1>	:= CELL<4:5>
COUNTER<0:5>	:= CELL<6:11>
VECTOR<0:51>	:= CELL<12:63>

THE LENGTH OF THE VECTOR ALLOWS US TO FOLLOW UP TO 52 DIFFERENT BRANCHES (CELLS).

A FEW PRIMITIVES:

```
CHAIN:  (A[2]<TYPE> # SAV) -> ERRORCHAIN
        (A[1]<TYPE> = PTR) -> CHAIN1
        (A[1]<TYPE> # CELL) -> ERRORCHAIN
        R1←NIL
        R2←A[1]
        -> CHAIN2
CHAIN1: MAR←A[1]<ADDR>
        MBR←M[MAR]
        R1←A[1]
        R2←MBR
CHAIN2: SAVCR←A[2]
LOOPCHAIN:      (COUNTER = 0) -> RETURN
        (R2<TYPE> # CELL) -> ERRORCHAIN
        (VECTOR<51> = 1) -> CDRCHAIN
        (R2<AM1> = INMEDIATE) -> ERRORCHAIN
        R1<TYPE>←PTR
        R1<AM2>←R2<AM1>
        R1<ADDR>←R2<CAR>
        MAR←R2<CAR>
        -> CDRCHAIN1
CDRCHAIN:       (R2<AM2> = INMEDIATE) -> ERRORCHAIN
        R1<TYPE>←PTR
        R1<AM2>←R2<AM2>
        R1<ADDR>←R2<ADDR>
        MAR←R2<CDR>
CDRCHAIN1:      MBR←M[MAR]
        COUNTER←COUNTER-1
        VECTOR←VECTOR/2
        -> LOOPCHAIN
ERRORCHAIN:     ERRORFLAG←1
        -> RETURN
```

THE ARGUMENTS FOR CHAIN ARE A PLEX AND A STRUCTURE ACCESS VECTOR IN A[1] AND A[2] RESP.

A[1] CAN BE EITHER A POINTER (TYPE PTR) OR A CELL. IF A CELL, WE SAVE AN EXTRA MEMORY CYCLE.

THE RESULT IN R1 IS A POINTER TO THE DESIRED ELEMENT WHICH IS ALSO STORED IN R2 (THIS IS A SIDE EFFECT). NOTE THAT WE WILL GET THE CORRECT ANSWER EVEN WHEN CHAINCOUNTER IS ZERO.

COUNTER AND VECTOR ARE FIELDS IN SAVCR (APPENDIX 3)

```
CONS:     (A[1]<TYPE> # PTR) -> ERRORCONS
          (A[2]<TYPE> # PTR) -> ERRORCONS
          STACK←UPC+1
          -> GETCELL
          R2<TYPE>←CELL
          R2<UUB>←0
          R2<AM1>←A[1]<AM2>
          R2<AM2>←A[2]<AM2>
          R2<CAR>←A[1]<CDR>
          R2<CDR>←A[2]<CDR>
          MAR←R1<CDR>
          MBR←R2
          M[MAR]←MBR
          -> RETURN
ERRORCONS:          ERRORFLAG←1
          -> RETURN
```

THE ARGUMENTS TO CONS ARE 2 POINTERS, IN A[1] AND A[2] RESP. THE RESULT IN R1 IS A POINTER TO THE NEWLY CREATED CELL, WHICH IS ALSO STORED IN R2 (THIS IS A SIDE EFFECT)

```
EQ:       R1←NIL
          (A[1]<TYPE> # A[2]<TYPE>) -> RETURN
          (A[1]=A[2]) -> R1←TRUE
          -> RETURN
```

```
RETURN:   T1←STACK
          (T1<TYPE> # UPC) -> RETURN
          UPC←T1
```

RETURN WILL GET US OUT OF TROUBLE BY POPPING UP THE STACK UNTIL IT FINDS A MICROPROGRAM COUNTER. I.E. A VALID RETURNING ADDRESS, AND BY CLEANING UP THE STACK.

```
GETI:     IFLAG←1
GET:      (A[1]<TYPE> = PTR) -> GETO
          (A[1]<TYPE> # PL) -> ERRORGET
          R1←NIL
          R2←A[1]
          -> GET2
GETO:     MAR←A[1]<ADDR>
          R1←A[1]
GET1:     MBR←M[MAR]
          R2←MBR
          (R2<TYPE> # PL) -> ERRORGET
GET2:     IAR←R2<PN>
          (IFLAG = 0) -> T1←PDT[IAR]
          (IFLAG = 1) -> T1←IAR
          (A[2]<CDR> = T1) -> RETURN
          (R2<AM2> = INMEDIATE) -> ENDGET1
          R1<TYPE>←PTR
          R1<AM2>←R2<AM2>
          R1<ADDR>←R2<NP>
          MAR←R2<NP>
          -> GET1
ERRORGET:         ERRORFLAG←1
          -> RETURN
ENDGET1:          R1←NIL
          -> RETURN
```

GET SEARCHES FOR A PROPERTY (A[2]) IN A PROPERTY LIST OF AN ATOM (A[1]).

THERE ARE TWO BASIC FLAVORS, DEPENDING ON WHETHER WE KNOW THE INDEX OF THE PROPERTY (I.E. ITS SHORT NAME OR ENTRY POINT IN PDT) OR NOT, IN WHICH CASE WE REQUIRE AN EXTRA TABLE LOOK-UP INTO PDT.

GET CHAINS DOWN THE PROPERTY LIST USING THE NP POINTER UNTIL IT FINDS THE END OF THE PROPERTY LIST (THE PROPERTY IS NOT THERE) OR A PROPERTY ELEMENT WITH THE SAME INDEX (IN THE IMMEDIATE VERSION) OR ONE WHOSE ASSOCIATED PDT ENTRY IS EQUAL TO THE SECOND ARGUMENT.

R1 WILL CONTAIN A POINTER TO THE PL ELEMENT AND R2 THE PL ELEMENT PROPER. IN CASE OF FAILURE TO FIND THE PROPERTY R1 WILL CONTAIN NIL AND R2 THE LAST PL ELEMENT.

```
EQUAL:   STACK←'CORK'
EQUAL1:  STACK←UPC+1
         -> EQ
         (R1 = TRUE) -> RETURN
         (A[1]<TYPE> # PTR) -> ENDEQUAL2
         (A[2]<TYPE> # PTR) -> ENDEQUAL2
         (A[1]<AM2> # LISTPOINTER) ->ENDEQUAL2
         (A[2]<AM2> # LISTPOINTER) -> ENDEQUAL2
         MAR←A[1]<ADDR>
         MBR←M[MAR]
         R1←MBR
         MAR←A[2]<CDR>
         MBR←M[MAR]
         R2←MBR
         T1<TYPE>←PTR
         T1<AM2>←R1<AM2>
         T1<ADDR>←R1<ADDR>
         STACK←T1
         T1<AM2>←R2<AM2>
         T1<ADDR>←R2<ADDR>
         STACK←T1
         A[1]<TYPE>←PTR
         A[1]<AM2>←R1<AM1>
         A[1]<ADDR>←R1<CAR>
         A[2]<TYPE>←PTR
         A[2]<AM2>←R2<AM1>
         A[2]<ADDR>←R2<CAR>
         STACK←UPC+1
         -> EQUAL1
         (R1 = NIL) -> ENDEQUAL3

         A[2]←STACK
         A[1]←STACK
         -> EQUAL1
ENDEQUAL2:        R1← NIL
         -> RETURN
ENDEQUAL3:        T1←STACK
         (T1<TYPE> = CORK) -> RETURN
         -> ENDEQUAL3
```

'CORK' IS OUR GUARANTEE THAT WE WILL ESCAPE EQUAL COMPLETELY, AS SOON AS ONE OF THE LEVELS OF RECURSION FAILS.

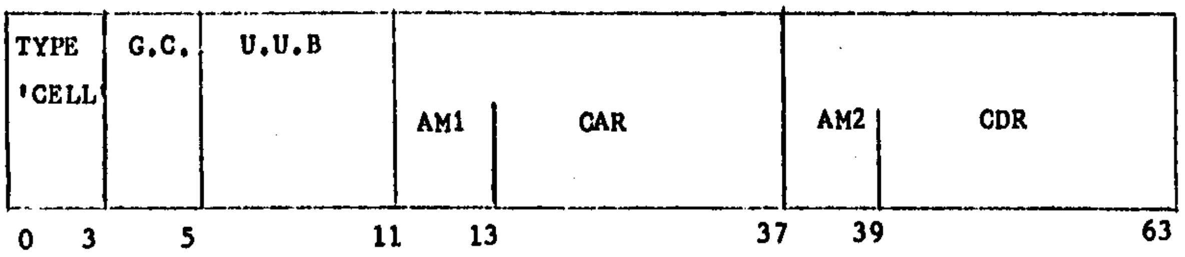


Figure 4.1: LIST CELL

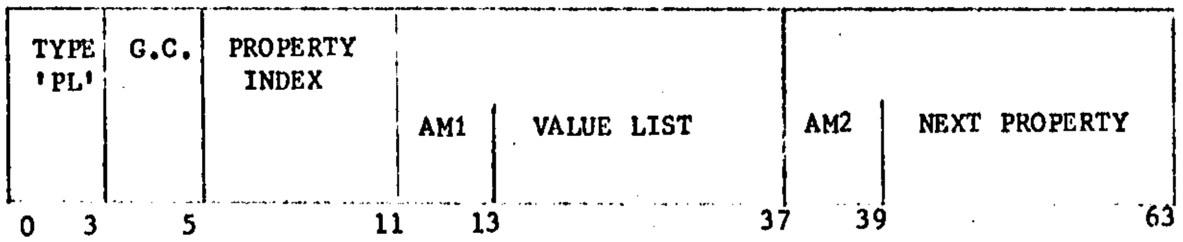


Figure 4.2: PROPERTY LIST ELEMENT

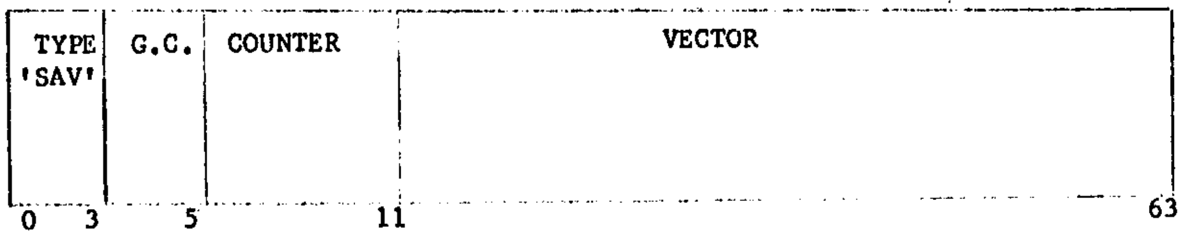


Figure 4.3: STRUCTURE ACCESS VECTOR

APPENDIX 5

DESCRIPTION OF MICROCOMMANDS

COMMON FIELDS IN MICROWORDS

```
MW/MICROWORD<0:63>
TYPE<0:3> := MW<0:3>       ALWAYS TYPE 'MICROCOMMAND'
OP<0:2> := MW<4:6>         SELECT  INTERPRETATION.
```

RBS/RELEASE-BUS-SIGNAL<0:4> := MW<59:63> USED TO RELEASE AND CLEAR BUSES

```
        RTB1 := RBS<0>
        RTB2 := RBS<1>
        RALU := RBS<2>
        RBS<3:4>          UNDEFINED
```

MICROWORD TYPE 1 , SPECIFY 1 ALU OP AND/OR A FULL WORD TRANSFER AND/OR AN INDIRECT FULL WORD TRANSFER (USING THE INDIRECT ADDRESS REGISTER). SEE FIG. 5.1.

```
ALUF/ALU-FIELD<0:21> := MW<7:28>
        ALUE/ALUF-ENABLE := ALUF<0>
        ALUB1/ALU-BUS-1<0:7> := ALUF<1:8>        OPERAND 1
        ALUB2/ALU-BUS-2<0:7> := ALUF<9:16>       OPERAND 2
        ALUFN/ALU-FUNCTION<0:4> := ALUF<17:21>
TF/TRANSFER-FIELD<0:17> := MW<29:46>
        TFE/TF-ENABLE := TF<0>
        TBS/TRANSFER-BUS-SELECTOR := TF<1>
        R1<0:7> := TF<2:9>
        R2<0:7> := TF<10:17>
ITF/INDIRECT-TRANSFER-FIELD<0:10> := MW<47:57>
        ITFE/ITF-ENABLE := ITF<0>
        ITFT/IT-FROM-TO := ITF<1>
        ITBS/IT-BUS-SELECT := ITF<2>
        R1<0:7> := ITF<3:10>
MW<58>            UNDEFINED
```

MICROWORD TYPE 2 SPECIFY 1 BYTE TRANSFER AND A POSSIBLE JUMP. SEE FIG. 5.2.

```
BTF/BYTE-TRANSFER-FIELD<0:34> := MW<7:41>
        TBS := BTF<0>
        R1<0:7> := BTF<1:8>
        IBP1/INITIAL-BIT-POSITION-1<0:5> := BTF<9:14>
        R2<0:7> := BTF<15:22>
        IBP2/INITIAL-BIT-POSITION-2<0:5> := BTF<23:28>
        FBP2/FINAL-BIT-POSITION-2<0:5> := BTF<29:34>
JF/JUMP-FIELD<0:16> := MW<42:58>
        JE/JUMP-ENABLE := JF<0>
        NEXT/NEXT-UINSTRUCTION<0:15> := JF<1:16>
```

MICROWORD TYPE 3 SPECIFY 2 PARALLEL FULL WORD TRANSFERS, THE INPUT WORDS ARE EITHER IN M(LOCAL) OR AS OUTPUT OF ALU. SEE FIG. 5.3.

```
SALU/STORE-ALU<0:9> := MW<7:16>
        SALUE/SALU-ENABLE := SALU<0>
        TBS := SALU<1>
        R1<0:7> := SALU<2:9>
B1F/BUS-1-FIELD<0:16> := MW<17:33>
        B1FE/B1F-ENABLE := B1F<0>
        B1R1<0:7> := B1F<1:8>
        B1R2<0:7> := B1F<9:16>
B2F/BUS-2-FIELD<0:16> := MW<34:50>
        B2FE/B2F-ENABLE := B2F<0>
        B2R1<0:7> := B2F<1:8>
        B2R2<0:7> := B2F<9:16>
MW<51:58>       UNDEFINED
```

MICROWORD TYPE 4 SPECIFY JUMP ON CONDITIONS. THE CONDITIONS FIELD IS STILL UNDEFINED, IT MAY BE EITHER A DIRECT BIT MASK OR A ENCODED FIELD. SEE FIG. 5.4.

```
TCF/TEST-CONDITIONS-FIELD<0:35> := MW<7:42>
JF/JUMP-FIELD<0:15> := MW<43:58>
```

MICROWORD TYPE 5 SPECIFY CONDITION BITS TO BE SET OR RESET IN THE EXTERNAL WORLD. SEE FIG. 5.5.

```
SCF/SET-CONDITIONS-FIELD<0:51> := MW<7:58>
```

MICROWORD TYPE 6 SPECIFY A INMEDIATE BYTE COMPARISON AND A JUMP ON EQUAL OR NOT EQUAL. SEE FIG. 5.6.

```
CBF/COMPARE-BYTE-FIELD<0:18> := MW<7:25>
        BS/BUS-SELECTOR := CBF<0>
        R1<0:7> := CBF<1:8>
        IBP1<0:5> := CBF<9:14>
        BL/BYTE-LENGTH<0:3> := CBF<15:18>
IBP/INMEDIATE-BIT-PATTERN<0:15> := MW<26:41>
JF/JUMP-FIELD<0:16> := MW<42:58>
        JC/JUMP-CONDITION := JF<0>
        JA/JUMP-ADDRESS<0:15> := JF<1:16>
```

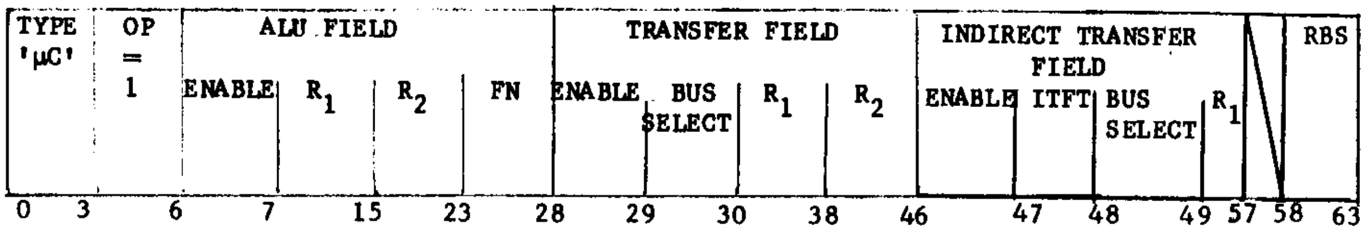


Figure 5.1: Microword type 1

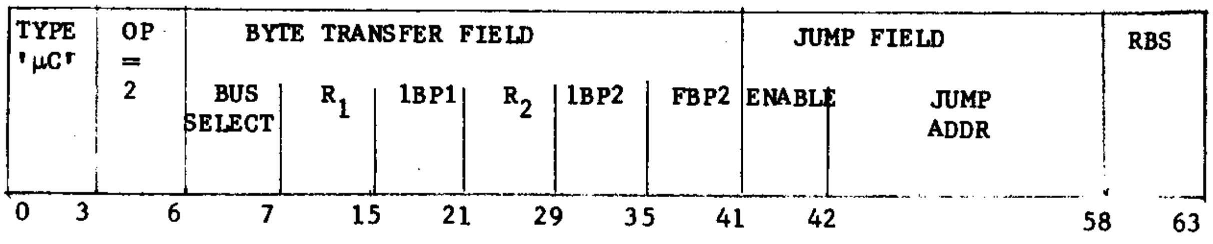


Figure 5.2: Microword type 2

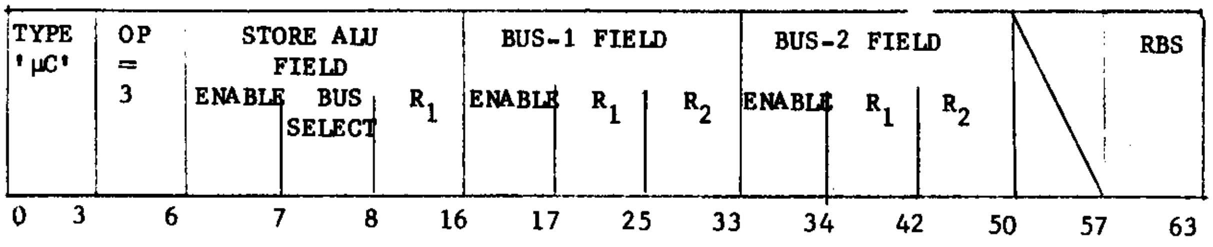


Figure 5.3: Microword type 3

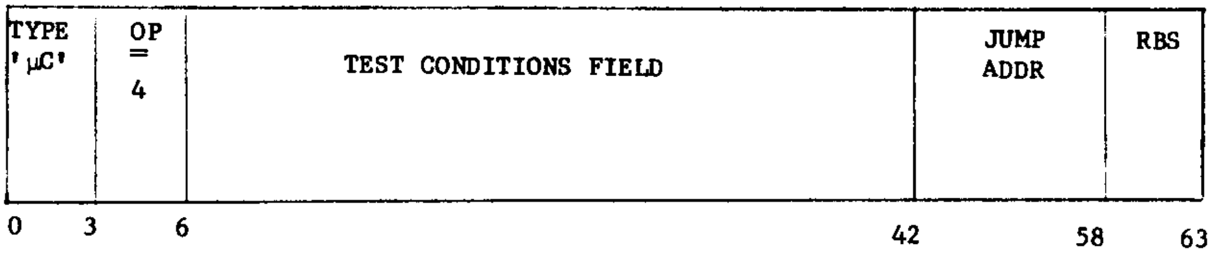


Figure 5.4: Microword type 4

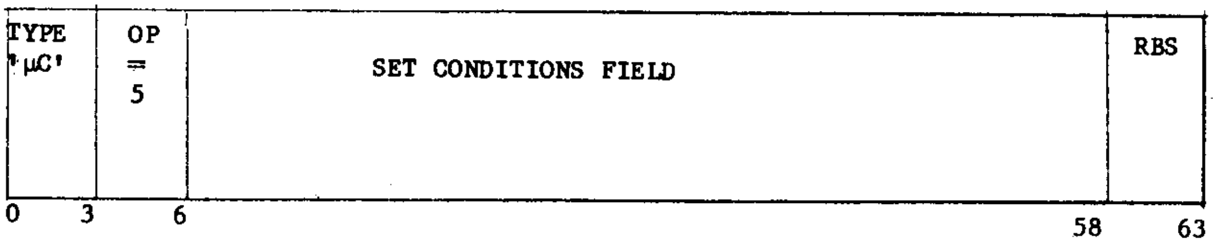


Figure 5.5: Microword type 5

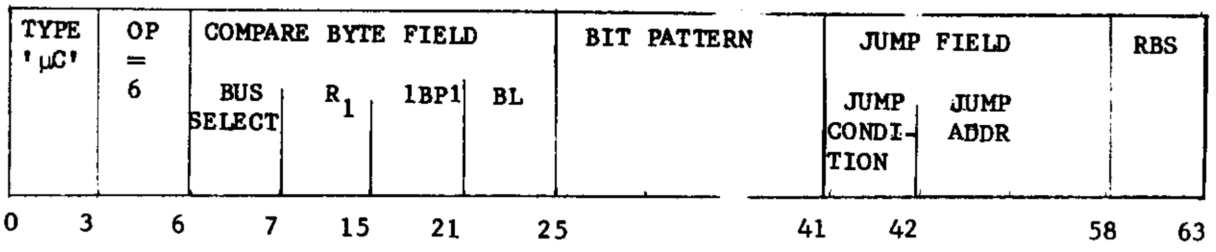


Figure 5.6: Microword type 6

APPENDIX 6

GARBAGE COLLECTION TECHNIQUES

OUTLINE

1. CONVENTIONAL GC

1.1 Description: (as in LISP 1.5)

Given pointers to main system lists (symbol tables, stack, OBLIST, etc.), enumerate and mark all cells that can be reached from these, by iteration on CDR's and recursion on CAR's, when latter is sublist. Then step linearly through memory, collecting all unmarked cells into a new Free Storage List / FSL; marked cells are unmarked (for next time GC is called) but otherwise untouched, thus returned to original (pre-GC) state. (Note: "/" introduces abbreviations.)

1.2 Advantages:

1) Simple and straightforward

2) Time-honored, well understood and debugged

3) Embodies basic GC principle: "If you can find it, save it; else re-use it".

1.3 Disadvantages:

1) Recursion requires use of stack, to potentially great depth.

2) Does nothing to consolidate/localize current data; if you've been working in 64K but only save 20K of "hot" lists, those 20,000 cells will still be scattered throughout the 64K space, i.e., your "working set" is not reduced, from either Paging or M.Cache standpoint.

3) Requires one dedicated marker bit in each word (P.LISP word allows two bits, though)

4) That linear sweep through all of allocated memory (after the find-and-mark phase) takes a while on the M-sizes we are talking about ($\sim$ 1 M words) [Fenichel 69] -- maybe several seconds.

5) Parallel GC is impossible.

6) Lack of consolidation makes conventional GC less attractive for compact-list/CLS systems, and ultimately useless where random-size plexes/RSP capability is desired.

2. COMPACTING GARBAGE COLLECTION/CGC

N.B. CGC is not confined to, nor does it imply, CLS [Hansen 69]. We use the term "compacted list" to mean a list of either type which has been consolidated by a CGC. See 2.4.1.

2.1 Description of Compacting GC Algorithms:

(References [Cheney 70] and [Fenichel 69] will be helpful in understanding what follows.) "GC" is an implementation-dependent (hence undefined at present [Fenichel 69] procedure which calls GC2 on each main system list (stack, object list, symbol tables, etc.).

"GC2" copies each list structure given by a pointer/PTR from GC into a "new" semi-space/SS, using a third routine "COPYLIST". Copylist copies only the top level of a list into the new area. Then GC2 advances a scanning pointer SCAN through the New copied lists, looking for yet-uncopied sublists in CAR's, to which COPYLIST is applied -- its result updates the CAR. Note whenever an Old cell is copied, its CDR is changed to point to the New copied cell. See Fig. 6.1. When SCAN catches up to NEXT, GC2 is finished. If GC has no more system lists, CGC is complete.

Note: NEXT is just the free-storage pointer, initialized to the beginning of the new SS.

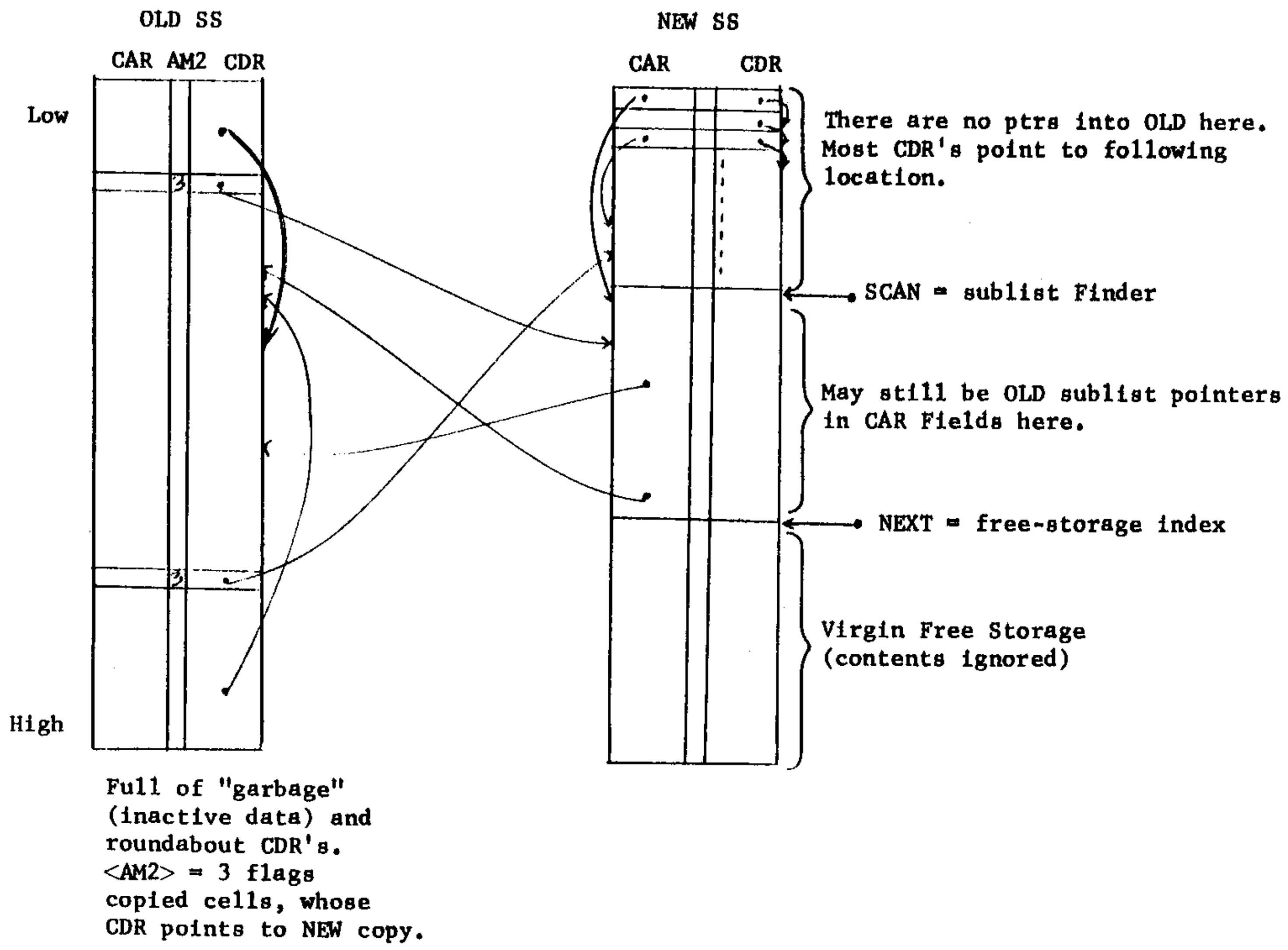

OLD SS
CAR AM2 CDR
Low
High
3
3
NEW SS
CAR CDR
There are no ptrs into OLD here. Most CDR's point to following location.
SCAN = sublist Finder
May still be OLD sublist pointers in CAR Fields here.
NEXT = free-storage index
Virgin Free Storage (contents ignored)
Full of "garbage" (inactive data) and roundabout CDR's. <AM2> = 3 flags copied cells, whose CDR points to NEW copy.

Figure 6.1
State of Semispaces with CGC Partially Completed

2.2 Advantages of CGC:

1) Reduction of page (or cache) faults; three reasons:

   a) Each individual list is localized in M following a CGC run, so accessing one word (say, the first) of it brings all or most of the entire list into real Mp (if paging), or the next few words into M.Cache.

   b) Reduction of total "working set" -- all active data are concentrated at low end of user's total allocated (virtual) M space.

   c) Free Storage List/FSL is linear (in rest of allocated M), which increases chances that new lists will be formed in contiguous blocks also.

2) Look-ahead schemes are profitable -- good chance that CDR (Mem [n]) = n + 1. (M.Cache is one crude form of look-ahead.)

3) Compacted lists can be read out onto disk, tape, etc. and read back in without disturbing common sublists or reentrant/circular/looped structures. [Hansen 69].

4) CGC can handle reentrant/circular/looped lists easily (though so does conventional GC).

5) CGC algorithm is simple [Cheney 70] -- especially for SLS. Much simpler than [Schorr 67].

6) CGC needs no recursion, stacks, or dedicated bits.

7) Works for (and more or less needed for) CLS.

8) Random-sized Plexes/RSP or Vectors are easy to implement because of the linear FSL, without the horrendous complexity of AED System [Ross 67]. Includes Strings and User-Defined Data Types a la SNOBOL 4.

9) PGC -- garbage collection in parallel with user-program execution -- becomes possible. No more n-second hangups while LISP collects!

2.3 Disadvantage of CGC:

Virtual memory size must be doubled [Fenichel 69] [Cheney 70] therefore address space must acquire one extra bit's worth of information to distinguish which "semi-space" an address is in during GC execution. But ...

1) Only two pages of "New" semi-space need be kept in Mp, and ~ 6 pages give best performance if special paging scheme is used (see 2.5). Faults are 1/page-size per page and predictable (hence special scheme).

2) Linear nature of read/write sequences in New SS goes well with M.Cache.

3) Size of M.virtual in the C.ai is such that doubling M.virtual needed may be o.k.

4) Or let OS.LISP (in collaboration with AMOS) keep a hunk of Mp around which severalLISP jobs can time-share for CGC (the physical location of this hunk changes: after a job CGC's into it, that job's Relocation Registers are changed to use it as working M; its old M is reallocated into the 'hunk' pool);

5) Or CGC onto Ms (disk, drum, etc.) directly and swap back into same Mp space. Fine, if you can stand the time required (you only write/read active words) and the use of recursion [Fenichel 69]; [Cheney 70] won't work without addressable memory.

2.4 Microcoded Implementation

Note: in flow charts, box numbers correspond to comments in code listings.

$$".X" \equiv \text{Mem}[X] = M[X\langle CDR\rangle]$$

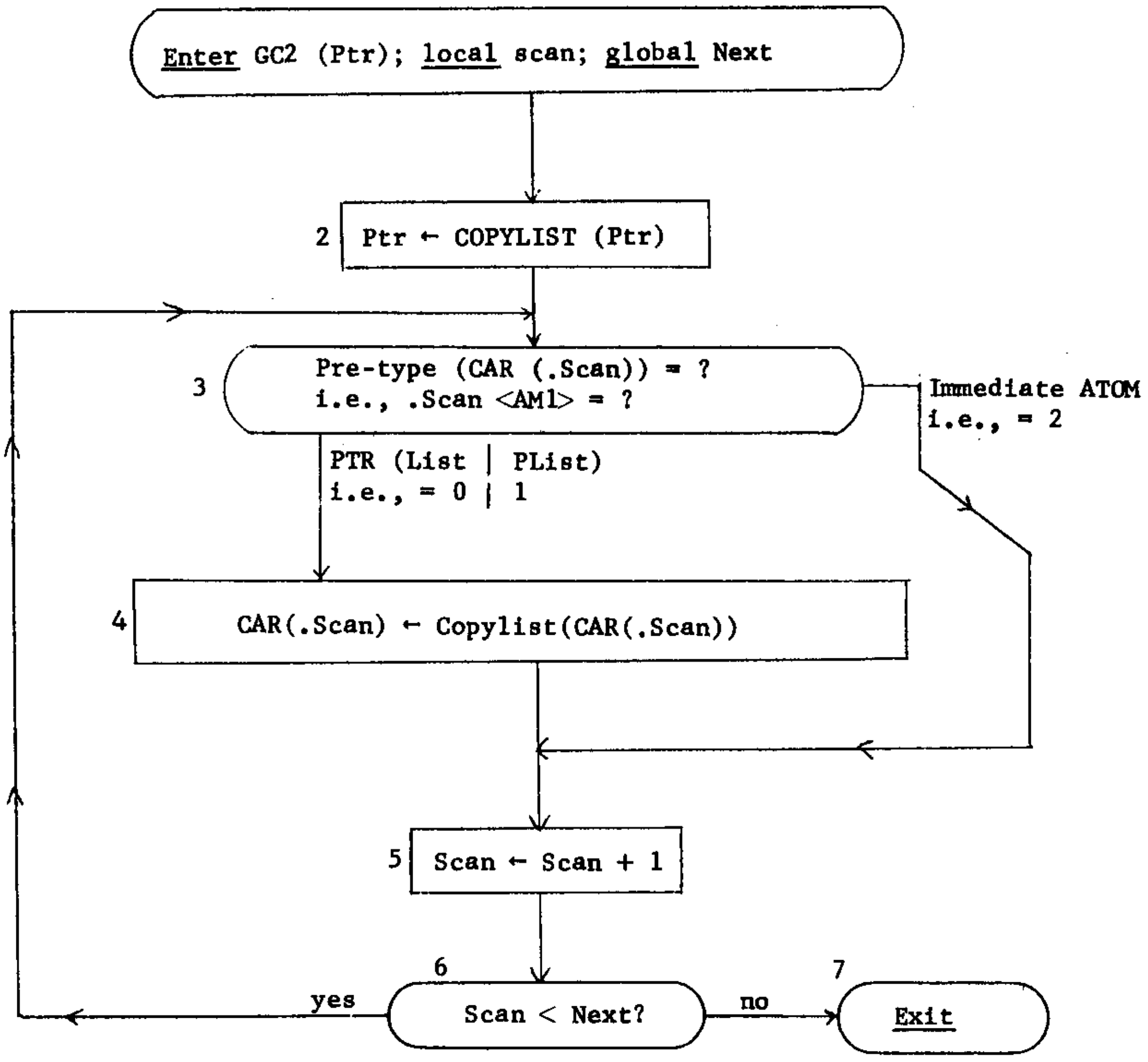
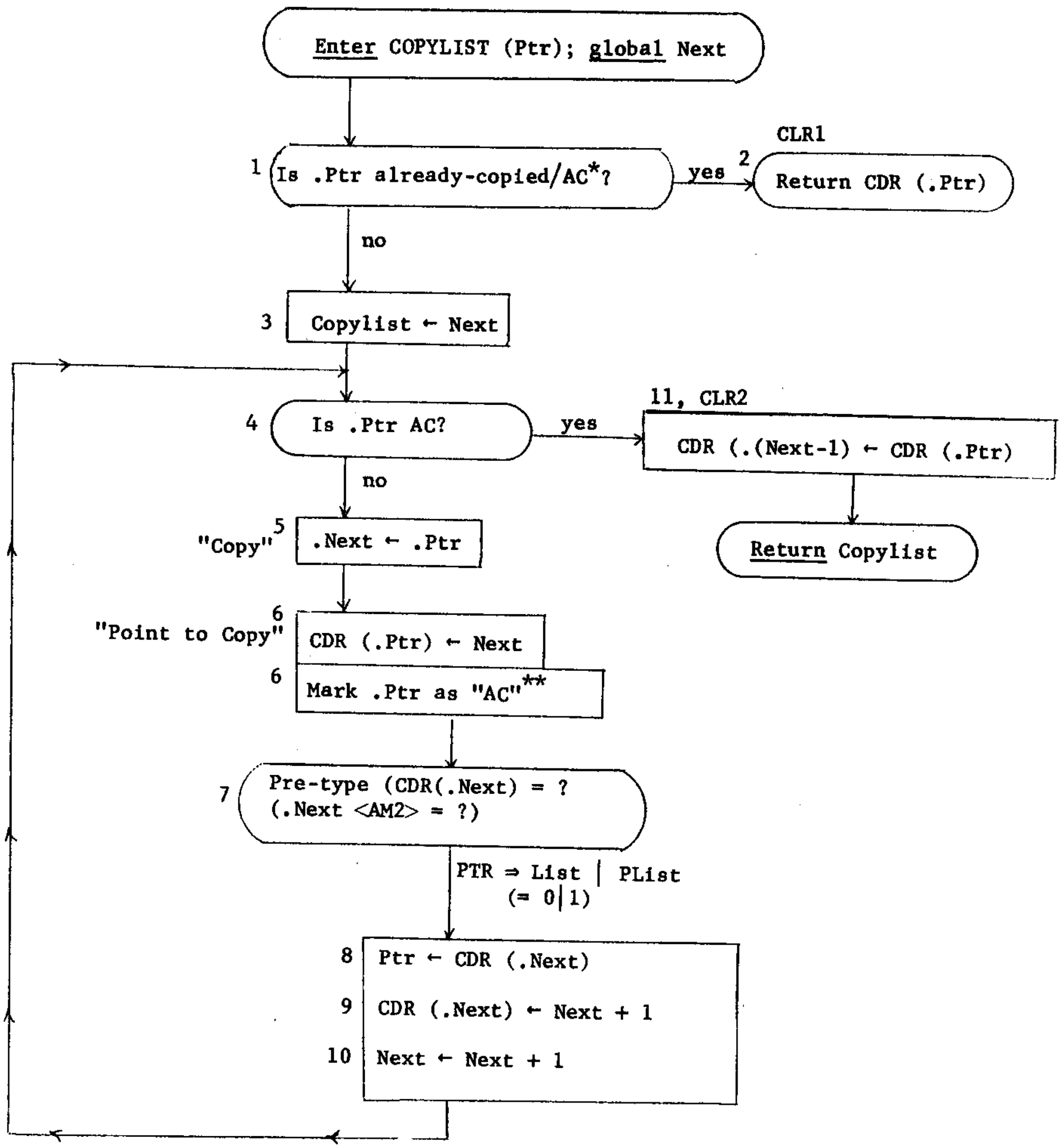


Figure 6.2
GC2



*Test <AM2> = 3, in this version

**.Ptr <AM2> ← 3, in this version

Note: We don't need the GC bits 4, 5; these can be used for TYPE or UUB fields.

Figure 6.3
COPYLIST

2.5 Paging Schemes for Compacting GC:

Special algorithms can take advantage of the predictable way GC2 and COPYLIST access New SS:

1) SCAN and NEXT both advance linearly through New, never point into Old SS.
2) All reads/writes in New are at locations pointed to by Scan and Next -- no probes at "random" locations.
3) NEXT is write-only, no fetches.
4) SCAN will read (and perhaps write) every word in a page once before running out of it -- likewise NEXT writes every word once before reaching boundary.
5) Thus n-word pages cause a fault about every 1/n accesses. You can't ask for better "folding".

Treat each of the two pointers' working sets as two ring buffers -- it requires ≥ two pages in Mp per pointer. When Scan or Next crosses a page boundary, the just-exited page is written out, and all Mp buffer sections other than that and the just-entered one are set up to fetch the next higher pages, once they have written out their previous contents.

> This look-ahead eliminates faults, if there are enough buffer sections to keep up with rate-of-advance of Next and Scan. Note since Next is write-only, it need never fetch pages into its buffer. The algorithm must allow for case where Scan and Next are on same page. The ring buffer is like a caterpillar tread rolling over Ms; see Figure 6.4.

Even if paging is not used, the linear-advance nature of New pointers will make good use of M.cache look-ahead capabilities --Scan can use all the look-ahead it can get.

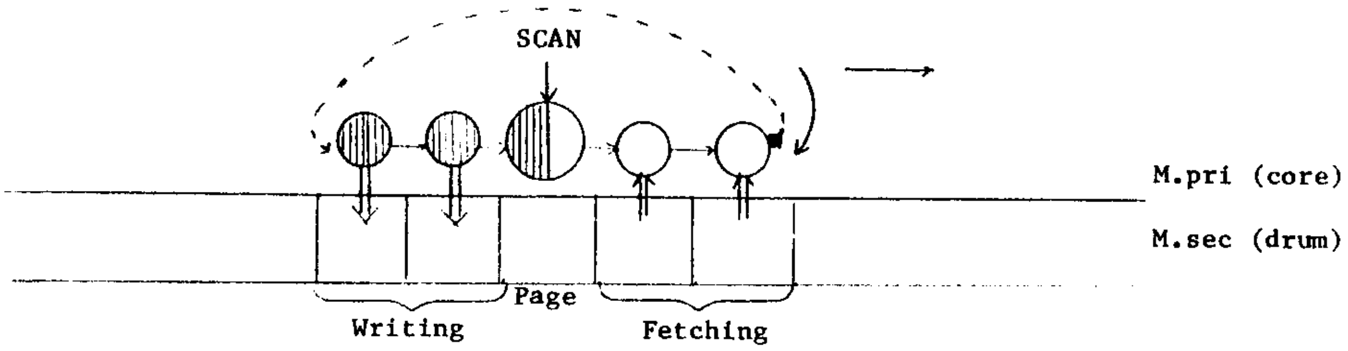


Figure 6.4
Ring Buffer "Caterpillar Tread" (See Sec. 2.5)

The best number of buffer sections is a function of how fast CGC pushes Scan and Next, page size, and time needed to swap a page in or out. Size of Next's buffer $\simeq$ 1/2 size of Scan's, since all sections can be writing-out, i.e., currently-active section is "leading tread" in the caterpillar picture (Figure 6.4) -- nobody is reading in.

3. PARALLEL [COMPACTING] GARBAGE COLLECTION/PGC

3.1 The Problem:

List-processing systems which rely on occasional garbage collections/GC to recover free storage (as opposed to SLIP-type LP systems using reference counts to keep track continuously of available space) must suspend operation entirely for a period ($\sim$ 1 second) while the GC runs.

This is o.k. in most applications, but increasing emphasis on real-time LP applications like robots and speech processing suggests a demand for a LISP system that would not go to sleep every minute or so.

The proposed solution is a Compacting GC that runs "in parallel" with user's LISP functions. Bits of GC are run in between bursts of user's computing, in usual time-sharing time-multiplexed fashion.

No change in LISP list structure is required (except slight address-space extension or addition of indicator bit, which is easily fit into P.LISP machine word), although PGC will also work for Compact List Structure/CLS.

PGC LISP runs around a 2-section ring buffer, filling current section ("semi-space") with computation while "simultaneously" copying still-valid previous results from preceding section.

3.2 Features (good and bad) of Parallel GC/PGC:

A = Advantage; D = Disadvantage; R = Requirement on implementation

(A) LISP system never has to stop dead for several seconds to GC (unless Bailing Out (3.3)).

(A) Applicable to conventional linked-lists/SLS, also to Compact List/CLS.

(D) Need double the (virtual) memory, for 2 semi-spaces, as with any CGC.

(R) PGC must compact storage (i.e., use CGC).

(D) Both semi-spaces/SS's are "active" most of time (though one usually more than other), as "working set" is distributed over both SS's. As a consequence, "thrashing" of pages may be serious enough that elaborate schemes to prevent this may be worthwhile. We are currently sketching some of these, based on recent discussions with Alan Kay of Stanford; these will be reported in a later paper.

(D) So special ring-buffer paging tricks (or M.cache advantages) of non-parallel GC (sec. 2.5) are unavailable.

(A) So $\exists$ no need to write special paging schemes!

(R) Address space must be well-ordered.

(R) Must be able to tell which SS an addr belongs to, without indicator bits, etc.

(R) STACK, ALIST, etc. may need special treatment (?).

(R,D) LISP primitives need slight mod's, which add slightly to running time.

(R) OS.LISP (or delegates) must synchronize PGC with user program -- lots of work here, though not that time-consuming.

(R) "Bail-out" procedure needed to recover from premature SS free-storage exhaustion. (3.3)

3.3 Description of Parallel GC Proposed: (See [Fenichel 69] for notions of "flip" and "semi-space".)

We use a slightly modified version of the CGC in Section 2. LISP jobs run under a supervisor/Super (some or all of which may be part of LISP interpreter) which can be in one of two modes -- User and GC. Mode at time t determines whether user's LISP code or GC is running at t.

When current semi-space is exhausted (i.e., 'next' points to top address of semi-space) there are two options, depending on whether PGC has been completed. If it has, the supervisor simply 'flips' semi-spaces and restarts. If not, then the storage has run out before GC has finished copying the old semi-space, and a special recovery procedure 'bail-out' is invoked (see 3.3.6).

A conventional interrupt system is unsafe. Some LISP and GC routines cannot be suspended in the middle of their operation. When pointers are being inspected or modified, instead of interrupts, these routines will check the system timer and 'voluntarily' return control to the supervisor, perhaps getting a little more than their time slice.

Let us call the time quanta for LISP and GC, QU and QG, respectively; in general QU $\neq$ QG. The supervisor alternates modes, switching at first opportunity after current mode's time slice is used up.

The ratio RG = QG/QU is fairly critical. If too small, the semi-space can be exhausted before GC has copied the old semi-space, requiring a bail-out action; if too large, it approaches the conventional GC, by stopping the user program for large periods of time after each 'flipping' of semi-spaces.

The supervisor can dynamically adjust RG during execution as follows:

Let S = semi-space size (number of cells)

Let UI = number of active cells (not garbage) at start of (i+1)th semi-space flip

Let GI = S-UI = number of recoverable (garbage) cells

The optimal relation RG is given by:

$$RG = QG/QU = K * GI / UI$$

where K = (average time required by GC to copy an active cell from old semi-space) / (average time required by LISP to obtain a new cell).

"BAIL-OUT" RECOVERY PROCEDURE FOR EXHAUSTED NEW SS (PROPOSED):

Figure 6.5. The Situation:

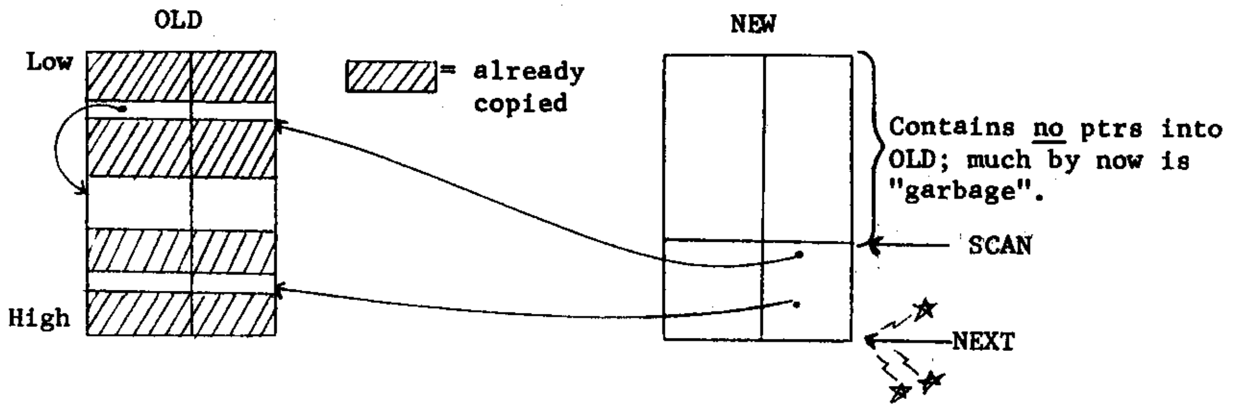


Figure 6.6. The Bail-Out: Append More M (Begged from AMOS) to NEW; Finish GC Exclusively

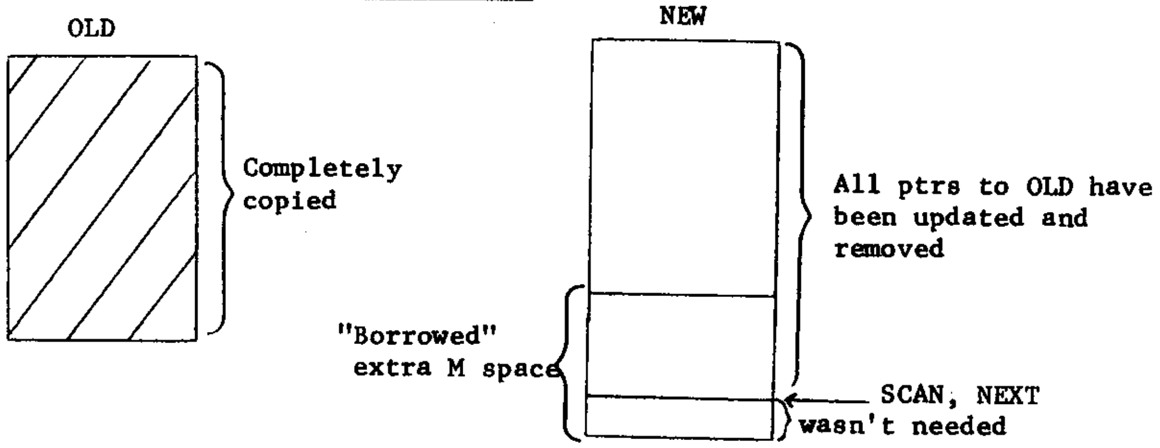
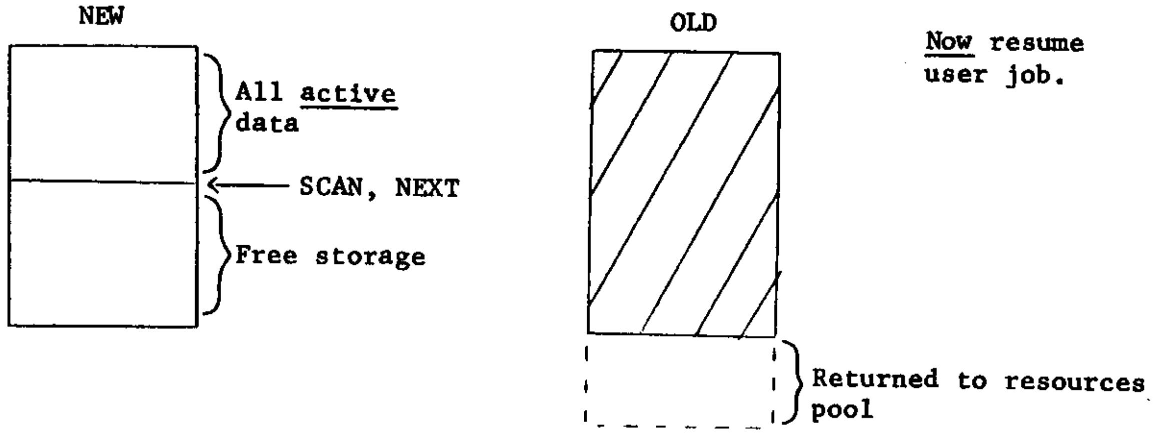


Figure 6.7. Now Flip SS's, Do Complete Exclusive GC; Return (De-allocate) Borrowed Space to LMOS.



Now resume user job.

IMPLEMENTATION

Some of the requirements of the PGC translate into small changes in the compacting garbage collector and introduce some extra overhead in some LISP primitives and on the LISP operating system:

LISP primitives must now recognize when the cell accessed has been copied into the new semi-space and if so spend one extra memory cycle to get it.

LISP 'pointer moving' primitives (RPLACA, RPLACD, NOONC, P-List functions, etc.) must do extra work to prevent errors; for instance, inserting in a cell already in the new semi-space, a pointer to a list still in the old one; if this is the only pointer to that list, it may never be copied into the new semi-space. (See Figure 6.8 for a solution.)

LISP system supervisor must now:

- control alternation of execution of LISP programs and PGC and dynamically compute the 'optimal' ratio RG

- initiate recovery procedures, with possible help from AMOS

- swapping policies should be integrated with the system, for instance, a job in I/O wait should not be swapped out. Instead, we can run 'solid' PGC (it does not hurt to finish earlier!

Subroutine GC2 must be modified to allow for cells created by non-pointer moving primitives like CONS with:

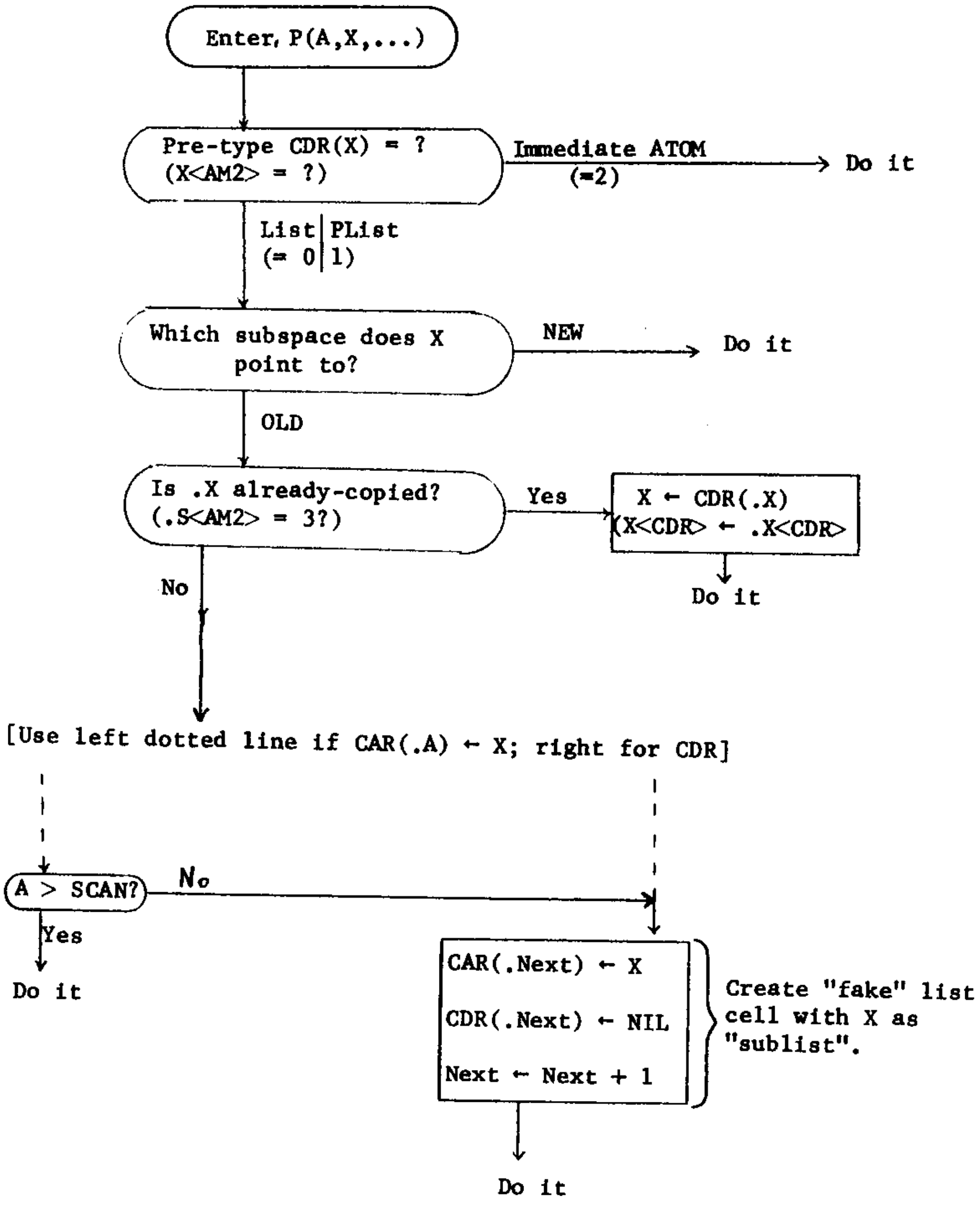
- CDR fields pointing to old semi-space

- CAR fields of cells reached by SCAN pointing to new semi-space

These require that GC2 test both CAR and CDR field at cell under SCAN.

Figure 6.8 "Patch" to be added at entry to any pointer-moving primitive P(A,X,...); <u>pointer</u> A,X; where intent is: CAR(.A) ← X <u>or</u> CDR(.A) ← X.

"Do it" = original (non-PGC) entry label of P(A,X...)



4. OVERVIEW OF GC AND LIST STRUCTURE OPTIONS

There are three independent (almost) Boolean features of a LISP system:

- Compacting Garbage Collection/CGC
- Compact-List Structure/CLS
- Parallel Garbage Collection/PGC

Figure 6.9 "Karnaugh Map" Summary of Options Possible in Combination

	Compacting GC/CGC		Conventional (McCarthy) GC	
Compact Lists/CLS	Yes - more or less required	Yes	No - PGC in CLS requires CGC	Not worth it
Linked (¬ compact) Lists/SLS	Yes - see Sec. 2	Yes - see Sec. 3	Not in our scheme;* PGC requires CGC	Sure - LISP 1.5, 1.6
		Parallel GC/PGC		

*Other schemes might use, e.g., page-wise McCarthy GC. See sec. 3.2.

APPENDIX 7

The following is an abstract of the report "C.ai: A Computing Environment for AI Research", obtainable from the Carnegie-Mellon University Computer Science Department.

A computer for artificial intelligence research is examined. The design is based on a large, straightforward primary memory facility (about 8 million 74 bit words). Access to the memory is via at least 16 ports which are hardware protected; there is dynamic assignment of the memory to the ports. The maximum port bandwidth is 8,600 million bits/sec. Processors for languages (e.g., LISP) and specialized terminals (e.g., video input/output) can be reliably connected to the system during its operation. The approach is evolutionary in that high performance processors, such as the Stanford AI Processor, can be connected to the memory structure, giving an overall power of at least 100 times a PDP-10 (and 200 to 300 times a PDP-10 for list processing languages) for 10 processors -- although no processors can be attached. Using this approach we might expect 40 $\sim$ 80 million PDP-10 operations/second.

At the same time, special language processors (P.ℓ) can be designed and attached. These processors give even larger power increases, but for restricted language use. Two processors, P.LISP and P.L* were examined for the LISP and L* languages and are reported on separately

A plan for building the machine in increments over the next three to five years is examined. Specific schedules are proposed.

Concurrent with the operation of the machine, there should be research into the design of hardware, software and theory of constructing large scale computing facilities with maximum modularity.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Computer Science Department Carnegie-Mellon University Pittsburgh, Pa. 15213	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

C.ai -- A LISP Processor for C.ai

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

M. Barbacci
H. Goldberg
M. Knudsen

6. REPORT DATE Aug. 9, 1971	7a. TOTAL NO. OF PAGES 61	7b. NO. OF REFS 9
8a. CONTRACT OR GRANT NO. F44620-70-C-0107	9a. ORIGINATOR'S REPORT NUMBER(S)	
b. PROJECT NO. A0827-5		
c. 61101D	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.		

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES TECH, OTHER	12. SPONSORING MILITARY ACTIVITY Air Force Office of Scientific Research 1400 Wilson Boulevard (SRMA) Arlington, Virginia 22209

13. ABSTRACT

A special microprogram controlled process designed for efficient interpretation of the LISP language is described. The processor has a fairly large, fast scratch-pad memory and uses two cache memories: for the LISP program and data being interpreted; and for the LISP interpreter. Several special purpose registers, small function units, and general byte manipulation capabilities are present.

The approach taken has been to avoid unorthodox implementation schemes and employs little in the way of unusually new (and untried) hardware. Such a conservative approach should enable an implementation in a reasonable length of time.

One of the places where efficiency in list processing (and in most programming applications) can be enhanced is in the ratio of instruction fetches to data fetches. To that end two unusual features were required: writable (up-datable) microcode and recursive control of microcode. With them, it is possible to implement the language interpreter as close as possible to the real hardware machine. Such a machine could also be a "shell" language processor. However, this was not a goal of the design, but a by-product.

The microprogrammed processes include a storage-compacting garbage-collector, which can be made to operate incrementally in parallel with user-program execution. This option avoids interruptions in LISP execution for garbage collection.

DD FORM 1 NOV 65 1473

Security Classification

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

Security Classification