THE LISP MACHINE

by

Richard Greenblatt

The Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
545 Technology Square
Cambridge, Massachusetts 01239

LISP Machine Demonstration

As many around the lab have become aware, I am working on a project designed to demonstrate the feasibility of a LISP machine concept. Below I briefly outline some aspects of the project.

History

The idea of a "LISP machine" has been kicked around more or less constantly since $t= -\infty$. A whole lore of techniques, etc. have been constantly building for many years. Recently, there are signs that this body to knowledge may be "going legit", perhaps concident with the availability (or at least technical possiblility) of suitable variable microprogrammed machines on which such a system might be implemented.

Peter Deutch of Xerox PARC wrote a paper for the 3IJCAI which described his ideas, not too similiar to the present ones beyond the basic idea of a single user minicomputer running a specialized microcode for LISP. As far as I know this is the only published reference along these lines. However the following draws freely on dozens of ideas from many systems including especially MACLISP, MUDDLE, and CONNIVER. I will also acknowledge a touch of special influence from LISP 2 and Donald Eastlake's attempt to "write Conniver in machine language."

Goals and Major Charactistics of the Target System

1. Capability to run large programs -- equivalent to several million PDP-10 words
2. Single user stand alone operation -- (with possible communication to a time-shared file system for program sharing and backup). The computer runs the same speed at 3pm as 3am!!!!!
3. Current non-prohibitive cost (approx $70K per system) with prospects for further decrease (both absolutely and relative to other means of accomplishing these objectives) in the future.
4. High degree of upward compatability with current PDP-10 MACLISP programs.
5. Competely integrated system -- all in common target language. Complete standardization of calls etc.
6. No ordinary local file system- but instead something better (see description of storage area feature).
7. Great storage efficency of compilied programs -- a factor of nearly 3 better "bit efficency" than MACLISP now.
8. "MUDDLE - CONNIVER" arg declaration syntax (available in an upward MACLISP compatable form).
9. Contexts, control structure hackery, et al CONNIVER style, with low cost unless you use it. Fully "co-recursive."
10. "Hardware" data types for increased computational efficency and error checking.
11. Extremely flexible base for experimentation in further changes to practically any aspect of the language.
12. Additional flavor because basic SUBRs like READ and PRINT are in LISP,

facilitating certain sorts of communication with them.
13. Improved methods to reclaim storage without garbage collection.
14. High speed transfer of data to and from Secondary store (a time-shared file system for example) via "SHIP" facility. This method does not involve conventional READing and PRINTing and is in some respects similar to the FASLOAD system on MACLISP.
15. Display oriented console interaction available.
16. High reliability and redundancy. If one machine goes down, others are still available.
17. Provisions to maintain "continuity" over long periods of time. ie you don't frequently walk up and type LISP^K and reload all your stuff. Instead the system does a much better job of protecting itself from getting clobbered and allowing programs to be segmented so they don't "get in each other's way" as much as now.
18. Frame oriented storage, "invisible" pointers, CDR codes, invoke pointers, trap pointers, etc. See description of storage conventions for details.
19. Miscellaneous other wonderful features.

Implementation

The basic components of the system are:

1) *the CONS microprocessor* -- see the CONS machine writeup by Tom Knight for a preliminary description.
2) *a PDP-11/10* which serves only as a console computer and UNIBUS supporter. The 11 cpu is idle in normal system operation after the system is loaded. In later, more "integrated" versions of the machine, the 11 would be eliminated.
3) *core memory* -- 50-100K PDP-11 memory ($12-25K at current prices very roughly.)
4) *fast backing store* -- the key to the feasibility of the system at least for the short term. Several options are available currently including:
   - -3600 RPM disk with 3 sets of heads around each track. max access = 5 2/3 ms 1/4 million wds price $10K
   - -3600 RPM 1 set of heads 1/2 million wds max access 16 2/3 ms price $18K (strongly rumored new DEC disk).
   - -existing on "tv" PDP-11 - 1800 RPM 1/4 million wds.
   - -existing on LOGO 11/45, 1800 RPM 1/2 million wds
   - -for the future, various "silicon disk" schemes now in the research phase.
5) *Slow backing store* -- 2315 type moving head disks like on LOGO and mini-robot machines. The one on LOGO cost $6K-$7K for 3 million wds. (Double density is now available at slight increase in price.)
6) *Display* -- "TK" type "direct connected" approx cost $4K-$5K.

Although "silicon disks" are not available yet at reasonable prices, several suitable devices (16 Kilobit IC shift registers for example) are strongly rumored to be slated for fourth quarter 74 announcement. Since fixed head disks have always been a prime source of pain, at the present time we greatly hope we will be able to skip over the phase of running the system on a fixed head disk and go directly to a solid state disk.

Processor

The CONS micro-processor is fairly completely described in a memo by Tom Knight, but the fundamentals will be gone over briefly here. The machine has provision for 4K of 45 bit words of control store. It has two scratch-pad memories internal to the processor, one of 32 words, the other of 256. The width of the basic data paths is 32 bits. The machine incorporates a map capable of dealing with a 23 bit virtual address space of 32 bit words. It also has a 1024 word pdl buffer, which greatly speeds up references to the top part of the stack. It is currently anticipated that it will require four 180 pattern Augat boards and in the neighborhood of 600 ICs.

Storage Managment

The fundamental unit of storage allocation is 32 bits, called a Q. A Q is capable of holding a single pointer to any virtual storage location, and in addition contains a garbage collector bit, a user control bit, a 2 bit cdr code and a 5 bit data type field. It takes two Q's to make a general LISP node, but via the cdr code bits an N element list can be stored in N consecutive Qs.

The mechanism of the storage area is highly integrated into the system. Essentially, CONS is a function of 3 arguments, the third being the area in which to do the CONS. A storage area is a region of contiguous pages each 64 Qs long. When allocated it is assigned a range of virtual addresses (which directly correspond to disk addresses). Each area has its own free storage list, which works in a way specified for that area (for example, an ascending free pointer or a list of free pages or a list or free nodes). These areas may not necessarily be data type related -- i.e. they do not correspond to FIXNUM space or FLONUM space etc. This sort of data type information is stored in data type fields of the Q's comprising a node.

There are many reasons one might choose to segregate a certain type of storage into its own region, separate from the large working storage area. For example, an area can be declared read-only. An area can be garbage collected independantly of other areas (or some other areas). An area can be set up for temporary storage then reclaimed en masse without a garbage collection if desired. This is a "dangerous" operation, but a debugging mode will be available to perform a gc mark and tell you if there are any live pointers to the area and where they are.

Areas can be used to improve utilization of core storage. For example, by storing PNAMEs and OBLIST pointers in a separate area, they can be kept out of the way during computation runs. (It is a fairly simple matter for the system to do this itself in the case of PNAMEs, but the area mechanism provides a convenient means for the user to do this himself for any type of storage he might choose to distinguish.)

It is possible to operate on the contents of a area as a whole, by copying it for backup for example. The external pointers of a relatively constant area can be factored out into a contiguous exit vector. This then could greatly speed up garbage collections since the area would not be marked, just the exit vector.

Since each area has its own routine for the sweep phase of garbage collection, certain storage conventions become possible that otherwise would not be even with the "hardware data types".

This discussion does not by any means exhaust the possible uses of areas.

Storage Conventions

Format of a Q:

```
        ---------------------------------------
        |   |   |   |      |               |
        | 1 | 1 | 2 |  5   |      23       |
        |   |   |   |      |               |
        ---------------------------------------
          |   |    |     |            |
  gc bit--|   |    |     |            |
    user bit--|    |     |            |
       cdr code---|     |            |
           data type-----|            |
                pointer---------------|
```

gc bit (1 bit)- obvious - used in conventional way during GC.

user control bit (1 bit) - unused by the system in ordinary list structure and is available for any use by the user. One suggested use: if set, it indicates this node has a hash-link in a particular hash table.

CDR code (2 bits) tells the CDR function what to do:
- 0 - normal node, CDR of this node is contained in the Q following this one.
- 1 - NIL CDR of this node is NIL.
- 2 - CDR NEXT - CDR of this node is the following Q.
- 3 - 2nd half of Q - for error checking. Says this Q is intended to be the 2nd half of a full node, so taking CDR of it directly is illegal. (As can been seen, these conventions allow the storage of an n element list in n consecutive Qs.)

pointer - (23 bits) pointer to element described by *data type*. In the case of FIX, a 23 bit immediate numeric quantity.

data type - (5 bits)

TRAP - any attempt to reference this pointer will trap.

LIST

ARRAY HEADER - Followed by indexing an type information for array, followed by array data. Strings and Symbol PNAMEs are stored as byte arrays.

SYMBOL (the term SYMBOL is used instead of "non-numeric atom" or the common but imprecise "atom") - points to a four Q block. The first of these is an ARRAY POINTER to the PNAME array. There follows the VALUE cell, the FUNCTION cell and the PROPERTY cell. Values of SYMBOLS are stored in VALUE cells as in PDP-10 MACLISP. However the VALUE cell is located directly following the SYMBOL header instead of being on the property list.

FIX number - 23 bits. Does not require any extra storage to exist in list structure (similar to INUMs on PDP-10 LISP).

Floating number - not implemented initially

Big number - also not implemented initially

Big Floating point number - also not implemented initially

FRAME - pointer to a whole 64 Q page (such as a PDL-FRAME, FUNCTION-ENTRY-FRAME, etc).

INVOKE - Has peculiar property that if somebody attempts to take a fundamental operation of it (such as CAR, CDR, RPLACA, or RPLACD) the invoke pointer gets called as a function (ACTOR style, sort of). Two arguments are supplied, the original invoke pointer and the attempted operation. The value returned by the function is returned as the value of the fundamental operation.

UCODE-ENTRY - entry pointer to microcompiled function.

LOCATIVE - a pointer to a single Q. Both CAR and CDR result in the same addressed Q.

INVZ-GC

INVZ-EFF-ADR

INVZ-CAR-CDR

INVZ-COMPONENT - *invisibe pointers*

Invisible Pointers

The concept of an invisible pointer is similiar to the concept of indirect addressing in a conventional machine, with the difference that the specification that *indirection* is to take place is made in the data rather than in the instruction. The possibility of implementing invisible pointers from the system design point of view depends on the fact that this is an *integrated* system. For example, in a conventional machine, it would clearly be unacceptable to have a range of number such that when they are added with the ADD instruction, what gets added is not these numbers themselves, but the contents of the memory location they point to. Here, due to the systematic interpretation of the data-type field which is defined to occur, we can have exactly that effect.

Since invisible pointers have never been available before to the system designer or to the data-base-structure designer, it is undoubtedly true that many uses for them will be found beyond what is forseen now. Already, however, it is apparent that they offer a new and very interesting dimension in these areas.

Some Proposed Uses of Invisible Pointers.

The efficiency and cleanliness of a copying type relocating garbage collector can be improved by the use of invisible pointers. Simply, as each piece of active data is moved into the new area, an invisible pointer to it is left behind in its *old home* pointing to the new one. This eliminates the necessity of doing a separate update pass. The system data base is in a consistant runnable state at all times. If a form of invisible pointer is used which carries the additional message "snap me out if possible", then the mark phase of the next succeeding garbage collection can serve to finish snapping any links not touched since the previous garbage collection. Indeed the system has a data type, INVZ-GC, which is used precisely in this manner.

Invisible pointers can be used to provide an *escape* mechanism to allow certain storage conventions to be used, which otherwise would lead to painful restrictions. For example, consider a scheme like the CDR-CODE scheme proposed, which allows more efficent storage of linearized lists. What happens if someone RPLACDs the list so it is no longer linear? Insuffcent space has been allocated to store a full CDR pointer. If it were not for invisible pointers, this operation would have to be forbidden. With invisible pointers, one merely does a new CONS of a full node (consisting of two Qs) and places an invisible pointer to it in the one available Q of the original list. The list has then been properly RPLACDed, and there are not even any problems about pre-existing pointers into the midde of the list. INVZ-EFF-ADR (invisible during effective address computation) type of invisible pointer turns out to have the right "micro-properties" for this application.

Invisible pointers can be used to solve certain design problems relating to the internal operation of the system in a convenient manner. For example, consider the problem of referencing SPECIAL versus LOCAL variables. The LOCAL variable values are stored on the PDL, and are referencable by supplying a delta from some index register (in this case the argument pointer register). The SPECIAL variable values, on the other hand, are stored in the VALUE cell of the particular variable involved. The best one can do toward making them referencable with a small delta is to have a table of pointers to the VALUE cells, and arrange to supply an index into this table. However doing this makes SPECIAL variables "one level of addressing" off compared to local variables. The problem is neatly solved by having the table consist of invisible pointers. Then the same instructions can be used in either case, and the extra level of indirection will be automatically supplied if necessary. Again, INVZ-EFF-ADR is the sort of invisible pointer that works here.

Invisible pointers can be used to provide high level "language design" features in an efficient manner. For example consider the *variable linking* mechanism as occurred in MICRO-PLANNER. Basically if a pattern match matched two unassigned variables, the values of the variables would become *linked* so that if one was assigned, the other would be also. This could be implemented efficently by storing an INVZ-EFF-ADR invisible pointer in the value cell of one variable to the other. This feature of MICRO-PLANNER was very powerful, since it allowed certain theorems to be written in an "equation" sort of way, applying equally in either direction.

We also desired to make available to the LISP programmer a form of invisible pointer. The ones described so far are not really available to the LISP programmer in that he cannot for example, set variables to them, since they are "invisible" to him. We define two more types, INVZ-CAR-CDR and INVZ-COMPONENT. INVZ-CAR-CDR does not act specially when referenced by data transmission instructions. Thus if a variable is set to an INVZ-CAR-CDR pointer and another variable is set equal to the first, the second variable will get set to the invisible pointer, not the thing being "invisibled to". However, if an INVZ-CAR-CDR pointer is fed to a fundamental component operation (CAR, CDR, RPLACA, RPLACD) the "invisibleness" will occur, followed by the operation.

INVZ-COMPONENT differs from INVZ-CAR-CDR in that only the component (CAR or CDR) in which it appears is invisibled, not both. Thus of the INVZ-COMPONENT is in the CAR position, CDR is unaffected while if it had been an INVZ-CAR-CDR CDR would be obtained by indirecting, then taking CDR of the resulting pointer.

This discussion does not exhaust by any means the obviously useful uses that have been found for invisible pointers already. In fact, they serve an additional conceptual purpose of providing a means to "open one's mind" in thinking about what is really going on in the system. For example, what happens if an invisible pointer is passed as an argument?

Subroutine calling

All subroutine calls in the system are directly compatible (no "UUO" handler for argument conversion).

As the args to a function are built up, they are placed directly in a frame in the correct position for their eventual reference by the target function (i.e. no argument loading into the AC's just before the call).

When a function is really called (at *destination last*), if its FUNCTION cell does not contain a frame pointer (to the FUNCTION-ENTRY-FRAME or FEF), a UCODE-ENTRY or an ARRAY-POINTER, then a trap to the error system (and possible interpretation) occurs.

The FEF contains information as to the number of desired args, desired data types or *don't care*, *local*ness or *special*ness, *optional*ness, *rest*ness etc. If any exception occurs during the entry process, a corresponding trap happens after the entry operation is complete. This greatly simplifies tracing, since the trace function does not need to worry about being transparent to the supplied args. If any args or AUXes are SPECIAL, appropriate binding is done. (This consists merely of swapping the current contents of the value cell with the allocated location that would have been used had the variable been local.)

For initialization of OPTIONAL args not supplied and AUXes, the FEF contains a pointer and a field decoding the following options:

Initialize var to pointer

Initialize var to contents of pointer

The FEF also has pointers to QUOTEd list structure, numbers, pointers to value and function cells and other Q quantities needed by the compiled code. The FEF contains the initial PC and the "NAMES" of all varibles (special and local).

```
                                PDL Frame

                     FPL                                   LPL
              -----------------------          ↑                       ↑
(64 Qs)       |    | return address |          |-----------------------|
              |-----------------------|        |   CALL state           |
              |    control link       |        |   EXIT state           |
              |    access link        |        |   ENTRY state          |
              |    local pdl pointer  |        |   FEF pointer          |← Argument ptr
              |    bind block pointer |        |-----------------------|
              |    (others)           |        |      args              |
Argument ptr →|    FEF pointer        |        |       ↓                |
              |-----------------------|        |-----------------------|
              |      args             |        |    local block         |
              |       ↓               |        |       ↓                |
              |-----------------------|        |-----------------------|
              |    local block        |        |    local pdl           |←- Stack poin
              |       ↓               |        ↓                        ↓
              |-----------------------|
              |    local pdl          |
Stack pointer→|       ↓               |

              |                       |
              -----------------------
```

As mentioned previously, the same frame is used to store the EVALuated args until they are all ready, transmit them to the called function and store them during the execution of the called function. It is allocated from a PDL FRAME space, and is linked to other frames only by pointers. There is no storagewise linearity to the PDL.

A PDL frame consists of first a fixed allocated section, followed by arg slots, internal variable slots, and internal pdl.

The fixed allocated section contains:

A pointer to the FEF
control link up
access link up
internal PDL pntr - a local pushdown list within the frame used for storing evaluated args and pointer to frames containing evaluated args.
Various fields giving the lengths of the other parts.

Potentially, a PDL frame may overflow into more than one block, but this may not be implemented initially.

Linear Pushdown List Mode versus Frame Pushdown List Mode

It is a very powerful feature to be able to save the entire context of the entire computation (variables and control structure), possibly to resume it at a later time. This feature is fundamental to the operation of Conniver, for example. Implementing this feature one finds that the conventional pushdown list generalizes into a tree-structured pushdown list (also refered to as a spaghetti stack). The maintainence of this stucture in a conventional way presents severe storage allocation problems, since one can return to any node of the tree and expect to grow new "linear" storage from there.

In Conniver, the solution used is to maintain the necessary storage in LIST space, which works but at a heavy cost in CONSes and resulting garbage collections. This overhead is fairly severe even in Conniver which is an interpreter, but becomes completely unacceptable when one is considering a compiled system with basic execution speeds several hundred times that of the existing Conniver.

The solution originally proposed is the so called Frame Pushdown List (FPL) scheme. The idea is that one maintain a pool blocks or pages, each of a fixed size. It would be expected that a single block could hold the calling an running environment for any function activation in the system, although overflow mechanism to use more than one page for a single call could be provided. Thus when it is desired for one function to call another, first a block is obtained from the free storage list of such blocks. The desired arguments then loaded in, along with access and control pointers as necessary. Finally, control is transfered to the called function, which also uses the block for any temporary storage it may require. Thus the logical push down list is completely *delinearized.* Each block would contain a bit saying "an environment pointer has been created pointing to me". When a function return occurs, the block can simply be returnned to the free storage list if this bit is not set. If it is set, then the block remains as active storage, and the "environment pointer points here" bit of its father blocks along the ACCESS and CONTROL links must be set. The block would then be reclaimed only by conventional garbage collection. Quite a bit of storage is wasted due to the large *grain size* of the blocks (many CALL-EXECUTE transactions require much less space the the maximum). However this sort of overhead is a function of the depth of function nesting and relatively constant otherwise. Since it is believed that the depth of function nesting grows at some "log sort of rate" with increasing size of the entire system, at some point the wasted storage should be an acceptably small fraction of the total.

This solution works fine for its intended purpose, but we came to realize that there was a tremendous amount of overhead involved in setting up a completely new local environment on each function call as opposed to just incrementing the push down pointer in a conventional system. We figured that the basic call instruction would be over 25 microseconds in memory references alone. This is not too bad if you are really making use of the context switching feature (it compares to a figure of several milliseconds in the current Conniver, for example), however this is too high a price to pay when running old-style LISP which is extremely dependant on subroutine calling and therefore very sensitive to any slowdown.

The solution was made possible by the micro-coded nature of the machine. There is a machine state word bit which indicates at any time whether the current mode of execution is FPL or a more conventional Linear Pushdown List (LPL) mode. All necessary changes are made to the computation of indexes, Call-Return sequences, etc such that the same identical source code can execute equally well in either mode. In fact, a dynamic transition is possible. Thus a function operating in LPL might call another function which does a SAVE-CONTEXT operation. This would result in the LPL stack being recopied into FPL form, with the appropriate implicit data being supplied at that time. When the called function returns, the calling function finds itself operating in FPL mode, but cares not since appropriate changes have occurred in the interpretation of the order code to allow the same instructions to work.

Context Switching and Environments

As has been explained, the pdl consists entirely of blocks linked by pointers so there are no great problems in allowing it to assume a tree or <u>spaghetti</u> shape. <u>Variable context</u> handling is somewhat more difficult however.

The following scheme works and is simple. No one knows how bad its overheads are, but we shall find out!

Briefly - variables have value cells or <u>shallow bindings</u> exactly as in MACLISP. The current value of a SYMBOL is always found immediately in the value cell.

Each *binding block* (pdl frame) is considered to have a one bit direction associated with it (up or down). At any time, one node at the end of one branch of the PDL tree is active. The links starting at the top of the tree to that node point *down* and all others *point up*. When a request is received to change context, the pointers from the target node are followed up block by block until a block with direction *down* is reached. This is the *intersection* of the <u>target context</u> with the <u>current context</u>. Then starting at the current active node, we work back up to the intersection "switching the arrows." This means swapping the current contents of the special variables bound in the block with the saved value in the block. When the intersection point is reached, we work back down to the target context, switching the arrows to the *down* state. This completes the context change. It should be noted that the PDL is not linked in the downward direction, thus in order to transverse it downward it is necessary either to have a temporary storage buffer and save the pointers on the way up or temporarily turn the pointers around on the way up. Probably the latter will be done initially. Normally PDL frames are collected on return, however, if a environment pointer has been created to the frame, a bit is set which indicates that the frame cannot be returned and must eventually be reclaimed by garbage collection. This bit also is propagated on both the control and access links.

Source Languages

As mentioned previously, it is desirable for a number of reasons to maintain compatability with MACLISP.

Therefore one normal mode of the compiler compiles MACLISP directly. A slightly different mode allows the use of "MUDDLE-CONNIVER" arg and variable declaration. The resulting output is in all ways run time compatable with that produced in the first mode. It is also possible to define a revised program structure (with read and print conventions) making use of the data types, a la MUDDLE. However, I have no plans to do this at the present time.

Most of the other additional features are accessed thru special functions and thus are not reflected in the language syntax.

Target Machine Details

Macro Instruction Set

The target machine macro-instruction is a 16 bit word divided as follows:

```
        ---------------------------------------
        |      |     |     |                  |
        |  4   |  3  |  3  |        6         |
        |      |     |     |               .  |
        ---------------------------------------
            |      |     |         |
main op---|      |     |         |
  destination----|     |         |
                       |         |
        register (index)--|      |
              displacement----------|
```

<u>Destination field</u>

The most interesting field is the destination field which (for data transfer instructions) specifies what to do with the data after it has been fetched from the effective address and operated on by the main op. The <u>destination field</u> has the following options.

ignore
to stack
to next
to last
to return
to next (quoted)
to last (quoted)
to next list

<u>To stack</u> -	pushes data on local stack and increments the pdl pntr
<u>To next</u> -	find a pointer to a frame (ala CALL) on the top of the stack. Increment the arg pointer of this frame and store the data where it points.
<u>to last</u> -	store next argument like *next*. Then locate pointer to the enter function left on the stack by CALL. Save exit state for current function and setup entry state for entering function. Transfer to starting address specified by FEF.
<u>To return</u> -	return data as value of this function
<u>Ignore</u> -	discard data but set *test indicators* (this is done in any case).

__To last (quote), next (quote)__ - same as last and next but store a one in the user control bit telling the target function it is getting a quoted argument (if it cares).

__To next list__ - The LIST n instruction (of operate class) allocates a space of n Qs and pushes its destination code and two pointers to the block. *Next list* stores the data in CAR of what's pointed to by the top of the stack. It then replaces the top of the stack by its CDR. If that result is NIL, the list has been filled and the other pointer to the head of the block is popped off and stored in the saved destination.

Main OP

__Call__ The effective address contains a pointer to the FUNCTION cell of the function to be called. A CALL block (four Qs) is allocated on the stack, the FEF pointer for the function to be entered is saved in one of them. The destination code along with other information is saved in the CALL state Q.

__Call0__ Like a CALL followed immediately by a destination last. (Used to call functions of no arguments.)

__MOV,CxR,CxxR__ Data instructions as above

__Misc__ groups 1 and 2 -- decoded by destination field includes:

MOVEM data off top of stack to effective address (E)
POP data off stack into E
PCAR pop data, take car of it, and store in E.
PCDR likewise but CDR
ADD,SUB,MUL,DIV,REM, - one op on stack, other in E, result to stack.
SCDR replace E with its CDR
SCDDR replace E with its CDDR
+1 replace E with its +1
-1 replace E with its -1
<, >, EQ set indicators. One arg in E, other on stack which is popped off.

__Branch__ - 3 bit branch code and 9 bit delta. If delta equals 0, the real delta is obtained from the following word.
There are two indicators, the ATOM indicator and the NIL indicator. They are set by all data transfer and arithmethic instructions. (for arithmetic instructions they become the ZERO and POSITIVE indicators).

__operate group__ - 512. miscellaneous functions. Have destination field as above. They take arguments on the pdl only. Included are:

NCONS
CONS
LIST 0-63 allocate 0-63 Q's and push two pointers to it (see discussion of destination NEXT LIST.)
CxxxR
CxxxxR
GET
GETL
ASSQ
ASSOC
EQUAL

These last are included only for speed, since this makes them accessible without going thru the frame handwavage. Eventually, provision for including certain type user functions in this group could be provided, with a compiler that compiled to microcode.