1979

Author: T. Knight

The LISP Machine Macro-instruction Set.

This document describes the LISP Machine's interpreted order code, referred to below as "macrocode." The macrocode is designed to be highly bit-efficient, and well-suited to LISP. The compilation of LISP into macrocode is very straightforward, as will be shown in examples below.

Macrocode is created by the "macrocompiler," which is called "QCMP." It is never neccesary to write out macrocode manually, because it correspondes so closely with the LISP source code that it would be easy to write the same thing in LISP.

When QCMP is run on a function, it is said to "macrocompile" the function. It produces one "Function Entry Frame," referred to below as a FEF, for each function compiled. The resulting LISP pointer has datatype DTP-FEF-POINTER, and points to the FEF, which is a block of memory at least 8 words long.

The FEF has several sections. The first section is always 7 words long, and contains various information about the format of the FEF and information about how the function should be invoked. For complete details of the bit layouts of these words, see the FORMAT document.

The next section of the FEF contains pointers to the VALUE cells and FUNCTION cells of symbols. These pointers are of datatype DTP-EXTERNAL-VALUE-CELL-POINTE and are used as an "exit vector", that is, compiled code can refer to these pointers in order to access special variables.

The next section contains the Argument Description List (ADL). The ADL contains one entry for each argument which the function expects to be passed, and contains all relevant information about the argument: whether it is required, optional, or rest, how to initialize it if it is not provided, whether it is local or special, datatype checking information, and so on. Sometimes the ADL can be dispensed with if the "fast argument option" can be used instead; this helps save time and memory for small, simple functions. The details can be found in the FORMAT document.

The next section of the ADL contains various constants which the function might want to refer to: if the function includes (FOO '(A B)), then the list (A B) would be put in the constants area so that the microcode can refer to it.

The rest of the FEF is the actual macroinstructions themselves. Each macroinstruction is 16 bits long, and so two macroinstructions are stored in each word of the LISP machine. There are four conceptual "classes" of microinstructions, each of which is broken down into fields in a different way.

CLASS I:

```
        ------------------------------------------
        |  3  |     4     |  3  |      6        |
        ------------------------------------------
          DEST.    OPCODE   REGISTER    OFFSET
```

There are nine class I instructions, designated by 0 through 10 (octal) in the OPCODE field. Each instruction has a source, whose address is computed from the "REGISTER" and OFFSET fields, and a destination given by the DESTINATION field. The instructions are:

OPCODE	NAME	
0	CALL	Open a call block on the stack, to call the function specified by the address. Whatever the function returns will go to the destination. The actual transfer of control will not happen until the arguments have been stored. (See destinations NEXT and LAST.)
1	CALL0	CALL in the case of a function with no arguments. The transfer of control happens immediately.
2	MOVE	Move the contents of E to the destination.
3	CAR	Put the CAR of the contents of E in the destination.
4	CDR	Analogous.
5	CADR	Analogous.
6	CDDR	Analogous.
7	CDAR	Analogous.
8	CAAR	Analogous.

The effective address, E, is computed from the "register" and the offset. The instructions really use addressing relative to some convenient place specified by the "register" field. The register may be:

REG	FUNCTION	
0	FEF	This is the starting location of the currently-running FEF. This is how the macrocode addresses the pointers to value and function cells, and the constants area.
1	FEF+100	Same as 0, plus 100 octal.
2	FEF+200	Analogous.
3	FEF+300	Analogous.
4	CONSTANTS PAGE	This is a page of widely used constants, such as T, NIL, small numbers, etc. There is only one constant page in t machine. They are kept all on one page so that they can be shared among all FEFs, so that they will not have to be repeated in each FEF's constant area.
5	LOCAL BLOCK	This is the address of the local block on the PDL, and is used for addressing local variables.
6	ARG POINTER	This is the argument pointer into the PDL, and is used for addressing arguments of the function.
7	PDL	The offset must be 77. The top of the stack is popped off and used as the operand. The other possible values of offset are not currently used.

(See the FORMAT file for how the PDL frame for each function is divided up into header, argument block, local block, and intermediate result stack.)

Note: The first 4 addressing modes are all provided to allow an effective 8-bit offset into the FEF.
Note: The same register-offset scheme is used in the class II instructions.

An additional complication in computing the "effective address" comes from invisible pointers. Once the register and offset have been used to compute an initial effective address E, the word at that location is examined (even if this is an instruction which uses E as a destination.) If the data type of that word is "Effective Address Invisible", the pointer field of that word is used as the effective address E. This is used, for example, to access value cells of special variables. The FEF "register" is used, and the location of the FEF addressed contains an effective address invisible pointer which points to the desired value cell. This scheme saves bits in the instruction, without requiring the use of extra instructions to make special value cells addressable.

The destination field is somewhat more complicated. First of all, before the result is moved to the destination, two "indicators" are set. The indicators are each stored as a bit, and correspond to processor status flags such as N and Z on the PDP-11. They are called the ATOM indicator, which is set if the result of the operation is an atom, and the NIL indicator, which is set if the resuly is NIL. The class III instructions (BRANCH) may look at the indicators.

Note: In actuality, there are not actually any physical indicators. Instead, the last result computed is saved in an internal register, and examined by the BRANCH instructions. The functional effect is the same.

The destinations are:

DEST	FUNCTION	
0	IGNORE	This is the simplest; the result is simply discarded. It is still useful, because it sets the flags.
1	TO STACK	This pushes the destination on the stack, which is useful for passing arguments to Class IV instructions, et
2	TO NEXT	This is actually the same thing as TO STACK, but it is used for storing the next argument to the open function when it is computed.
3	TO LAST	This is used for storing the last argument of the open function. It also pushes the result on the stack, and then it "activates" the open call block. That is, control will be passed to the function specified by the last CALL instruction, and the value returned by the function will be sent to the destination specified by the destination field of the call instruction.
4	TO RETURN	Return the result of the instruction as the value of this function. (i.e. return from subroutine.)
5	TO NEXT, QUOTED	This is the same as TO NEXT, except that for error checking, the USER-CONTROL bit of the word being pushed is set, telling the called function that it is getting a quoted argument (if it cares). (This is not implemented.)
6	TO LAST, QUOTED	Analogous.
7	TO NEXT LIST	This one is fairly tricky. It is used in conjunction with the LIST (Class IV) instruction to efficiently perform the lisp "LIST" function. It is documented under the LIST instruction.

Note: 5 and 6 (the QUOTED) destinations have not been implemented as of 11/03/76.

Note: The same DESTINATION field is used by the class IV instructions.

CLASS II:

```
        |       7        |  3  |      6      |
        -------------------------------------
               OPCODE        REGISTER    OFFSET
```

The class II instructions have no destination field; the result of the operation (if any) is either pushed on the stack (like a destination TO STACK or TO NEXT in a class one instruction) or is stored at the effective address. The "register" and offset are used in exactly the same way as in the class I instructions, except that the E calculated is sometimes used as a destination instead of a source.

The instructions are broken up into three subgroups by the first three bits of the opcode [in the microcode they are referred to as Non-destination instruction groups 1, 2 and 3], and then into the separate instuctions by the next four bits as follows: (a "-" in the left hand column means that this instruction pops the stack; a "+" means that it pushes something onto the stack.)

	GRP.	OPCODE	FUNCTION	
	11	0		Not used.
	11	1	+	Adds C(E) to the top of the stack and replaces the result on the stack.
	11	2	-	Subtracts C(E) from the top of the stack.
	11	3	*	Analogous.
	11	4	/	Analogous.
	11	5	AND	Analogous.
	11	6	XOR	Analogous.
	11	7	OR	Analogous.
-	12	0	=	\ These compare C(E) to the top of the stack, and if the
-	12	1	>	|-condition being tested is true, the NIL indicator is cl
-	12	2	<	| otherwise it is set. The stack is popped.
-	12	3	EQ	/
	12	4	SCDR	Get the CDR of C(E), and store it in E.
	12	5	SCDDR	Analogous.
	12	6	1+	Analogous.
	12	7	1-	Analogous.
	13	0	BIND	The cell at E is bound to itself. The linear binding pdl is used.
	13	1	BINDNIL	The cell at E is bound to NIL.
-	13	2	BINDPOP	The cell at E is bound to a value popped off the stack.
	13	3	SETNIL	Store NIL in E.
	13	4	SETZERO	Store fixnum 0 in E.
+	13	5	PUSH-E	Push a locative pointer to E on the stack.
	13	6	MOVEM	Move the data on top of the stack to E.
-	13	7	POP	Pop the top of the stack into E.

CLASS III

```
        -------------------------------------
        |  3  |    4    |         9         |
        -------------------------------------
         BRANCH    (14)           OFFSET
          CODE
```

The class III instruction is for branching. There is a 3 bit Branch Code field which determines whether the branch happens, and sometimes what to do if it fails. It is decoded as follows:

BRANCH

CODE	FUNCTION	
0	ALWAYS	Always branch.
1	NILIND	Branch if the NIL indicator is set, else drop through.
2	NOT NILIND	Branch if the NIL indicator is not set, else drop through
3	NILIND ELSE POP	\ These two are the same as NILIND and NOT NILIND, | except that if the condition fails,
4	NOT NILIND ELSE POP/	the stack is popped.
5	ATOMIND	Branch if the ATOM indicator is set, else drop through.
6	NOT ATOMIND	Analogous.

If the decision is made to perform the branch the offset, considered as a signed (two's complement) offset is added to the PC (i. e. a relative branch, such as is used by the PDP-11). If the offset is 777, however, it is interpreted as meaning that this is a long-distance branch, and the real offset is obtained from the next (16-bit) halfword. The PC added to is always the incremented PC; i.e. the address of the instruction +1 in the short case, and the address of the instruction +2 in the long case.

CLASS IV

```
---------------------------------------------------
| 3 |    4     |              9                   |
---------------------------------------------------
 DEST.   (15)               OPCODE
```

The class IV (miscellaneous) instructions take their arguments on the stack, and have a destination field which works the same way as the Class I instructions. They all have 15 (octal) in the opcode bits, and the actual opcode is in the last 9 bits. Thus there can be up to 512. of them. Most of them may be called directly from interpretive LISP, and so some of them duplicate functions available in classes I and II. [Note on implementation: since there are far too many class IV instructions to dispatch on using the dispatch memory of the CONS machine, the starting locations of the routines are kept in main memory. The location of the base of the dispatch table is kept in A-V-MISC-BASE at all times.]

Since most of these functions are callable from interpretive level (the normal LISP user sees them), they form part of the nuclear system, and so are documented in the Lisp Machine Nuclear System document (LMNUC >).

The first 200 (octal) Class IV operations are not in the dispatch table in main memory, but are specially checked for. These are the "LIST" instructions, which work in cooperation with the NEXT-LIST destination.
Operations 0-77 are called LIST 0 through LIST 77. The list <N> instruction allocates N Q's in the default consing area [A-CNSADF] which is initialized to a CDR-NEXT, CDR-NIL style list of NILs. Then the instruction pushes three words on the stack:
1) A pointer to the newly allocated block,
2) The destination foeld of the LIST instruction,
3) A pointer to the newly allocated block (another copy).
Note that the destination, as in the CALL instruction, is not used instantly; it is saved and used later.
After the LIST instruction has been performed, further instructions can store to destination NEXT-LIST; once the macro-code computes the next arg of what was the LIST function in the source code

it stores it to NEXT LIST. What destination NEXT LIST does is: the word on the top of the stack is taken to be a pointer to the next allocated cell. The pointer is popped, the result of the instruction is stored where it points, and then the CDR of the pointer is pushed back on to the stack. If the CDR was NIL however, then we must be finished with the LIST operation, so the NIL is popped off the stack and discarded and a pointer to the newly allocated area (another copy of which was thoughtfully stored on the stack) is sent to the destination of the LIST <n> instruction (which was also stored on the stack), and the two remaining words which the LIST <n> pushed are popped off.

CLASS V

Opcodes 16 and 17 (octal) are not used and reserved for future expansion.

EXAMPLES:

Here is a (very) typical Lisp function, for computing factorials.

```
(defun fact (x)
    (cond ((zerop x) 1)
          (t (* x (fact (1- x))))))
```

This is the macrocode produced by the compiler, typed out by the DISASSEMBLE function on the Lisp Machine.

```
22 MOVE D-PDL ARG|0      ;X
23 MISC D-IGNORE ZEROP
24 BR-NIL 26
25 MOVE D-RETURN '1
26 MOVE D-PDL ARG|0      ;X
27 CALL D-PDL FEF|10     ;@FUNCTION-CELL FACT
30 MOVE D-PDL ARG|0      ;X
31 MISC D-LAST 1-
32 * PDL-POP
33 MOVE D-RETURN PDL-POP
```

The first thing (line 22) is to push argument 0 (x) onto the stack, and (line 23) check if it is equal to zero. Line 23 uses the ZEROP miscellaneous function, which sets the "indicators" to NIL if the quantity was not ZERO. Line 24 is a branch instruction which tests the "indicators"; if NIL is set, it will branch to 26. If NIL was not set (the number was zero), it falls through to line 25, which returns the value 1.

If the number was not zero (the second clause of the COND in the source), then control passes to line 26, which pushes X on the PDL (first argument to the multiply on line 32). Next line 27 opens a call to FACT. Line 30 subtracts 1 from X (with the 1- miscellaneous function), and moves the result to "destination LAST". This result is thus the first and only argument to the recursive invokation of FACT, the result of which is left on the PDL because of the destination field of the CALL instruction on line 27.

Now (FACT (1- x)) and X are on the PDL, and they are multiplied by the multiply instruction on line 32. It leaves its result on the PDL, to be found by line 33, which returns the result.