28. How to Read Assembly Language

Sometimes it is useful to study the machine language code produced by the Lisp Machine's compiler, usually in order to analyze an error, or sometimes to check for a suspected compiler problem. This chapter explains how the Lisp Machine's instruction set works and how to understand what code written in that instruction set is doing. Fortunately, the translation between Lisp and this instruction set is very simple; after you get the hang of it, you can move back and forth between the two representations without much trouble. The following text does not assume any special knowledge about the Lisp Machine, although it sometimes assumes some general computer science background knowledge.

28.1 Introduction

Nobody looks at machine language code by trying to interpret octal numbers by hand. Instead, there is a program called the Disassembler which converts the numeric representation of the instruction set into a more readable textual representation. It is called the Disassembler because it does the opposite of what an Assembler would do; however, there isn't actually any assembler that accepts this input format, since there is never any need to manually write assembly language for the Lisp Machine.

The simplest way to invoke the Disassembler is with the Lisp function **disassemble**. Here is a simple example. Suppose we type:

```
(defun foo (x)
  (assq 'key (get x 'propname)))

(compile 'foo)

(disassemble 'foo)
```

This defines the function **foo**, compiles it, and invokes the Disassembler to print out the textual representation of the result of the compilation. Here is what it looks like:

```
22 MOVE D-PDL FEF|6           ;'KEY
23 MOVE D-PDL ARG|0           ;X
24 MOVE D-PDL FEF|7           ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

The Disassembler is also used by the Error Handler and the Inspector. When you see stuff like the above while using one of these programs, it is disassembled code, in the same format as the **disassemble** function uses. Inspecting a compiled code object shows the disassembled code.

Now, what does this mean? Before we get started, there is just a little bit of jargon to learn.

The acronym PDL stands for Push Down List, and means the same thing as Stack: a last-in first-out memory. The terms PDL and stack will be used interchangeably. The Lisp Machine's architecture is rather typical of "stack machines"; there is a stack that most instructions deal with, and it is used to hold values being computed, arguments, and local variables, as well as flow-of-control information. An important use of the stack is to pass arguments to instructions, though not all instructions take their arguments from the stack.

The acronym "FEF" stands for Function Entry Frame. A FEF is a compiled code object produced by the compiler. After the **defun** form above was evaluated, the function cell of the symbol **foo** contained a lambda expression. Then we compiled the function **foo**, and the contents of the function cell were replaced by a "FEF" object. The printed representation of the "FEF" object for **foo** looks like this:

```
#<DTP-FEF-POINTER 11464337 FOO>
```

The FEF has three parts (this is a simplified explanation): a header with various fixed-format fields; a part holding constants and invisible pointers, and the main body, holding the machine language instructions. The first part of the FEF, the header, is not very interesting and is not documented here (you can look at it with **describe** but it won't be easy to understand). The second part of the FEF holds various constants referred to by the function; for example, our function **foo** references two constants (the symbols **key** and **propname**), and so (pointers to) those symbols are stored in the FEF. This part of the FEF also holds invisible pointers to the value cells of all symbols that the function uses as variables, and invisible pointers to the function cells of all symbols that the function calls as functions. The third part of the FEF holds the machine language code itself.

Now we can read the disassembled code. The first instruction looked like this:

```
22 MOVE D-PDL FEF|6              ;'KEY
```

This instruction has several parts. The **22** is the address of this instruction. The Disassembler prints out the address of each instruction before it prints out the instruction, so that you can interpret branching instructions when you see them (we haven't seen one of these yet, but we will later). The **MOVE** is an opcode: this is a **MOVE** instruction, which moves a datum from one place to another. The **D-PDL** is a destination specification. The **D** stands for "Destination", and so **D-PDL** means "Destination-PDL": the destination of the datum being moved is the PDL. This means that the result will be pushed onto the PDL, rather than just moved to the top; this instruction is pushing a datum onto the stack. The next field of the instruction is **FEF|6**. This is an *address*, and it specifies where the datum is coming from. The vertical bar serves to separate the two parts of the address. The part before the vertical bar can be thought of as a *base register*, and the part after the bar can be thought of as being an offset from that register. **FEF** as a base register means the address of the FEF that we are disassembling, and so this address means the location six words into the FEF. So what this instruction does is to take the datum located six words into the FEF, and push it onto the PDL. The instruction is followed by a "comment" field, which looks like ;'KEY. This is not a comment that any person wrote; the disassembler produces these to explain what is going on. The semicolon just serves to start the comment, the way semicolons in Lisp code do. In this case, the body of the comment, 'KEY, is telling us that the address field (**FEF|6**) is addressing a constant (that is what the single-quote in 'KEY means), and that the printed representation of that constant is **KEY**. With the help of this

"comment" we finally get the real story about what this instruction is doing: it is pushing (a pointer to) the symbol key onto the stack.

The next instruction looks like this:

```
23 MOVE D-PDL ARG|0              ;X
```

This is a lot like the previous instruction; the only difference is that a different "base register" is being used in the address. The ARG "base register" is used for addressing your arguments: ARG|0 means that the datum being addressed is the zeroth argument. Again, the "comment" field explains what that means: the value of X (which was the zeroth argument) is being pushed onto the stack.

The third instruction is just like the first one; it pushes the symbol propname onto the stack.

The fourth instruction is something new:

```
25 (MISC) GET D-PDL
```

The first thing we see here is (MISC). This means that this is one of the so-called *miscellaneous* instructions. There are quite a few of these instructions. With some exceptions, each miscellaneous instruction corresponds to a Lisp function and has the same name as that Lisp function. If a Lisp function has a corresponding miscellaneous instruction, then that function is hand-coded in Lisp Machine microcode.

Miscellaneous instructions only have a destination field; they don't have any address field. The inputs to the instruction come from the stack: the top *n* elements on the stack are used as inputs to the instruction and popped off the stack, where *n* is the number of arguments taken by the function. The result of the function is stored wherever the destination field says. In our case, the function being executed is get, a Lisp function of two arguments. The top two values will be popped off the stack and used as the arguments to get (the value pushed first is the first argument, the value pushed second is the second argument, and so on). The result of the call to get will be sent to the destination D-PDL; that is, it will be pushed onto the stack. (In case you were wondering about how we handle optional arguments and multiple-value returns, the answer is very simple: functions that use either of those features cannot be miscellaneous instructions! If you are curious as to what functions are hand-microcoded and thus available as miscellaneous instructions, you can look at the defmic forms in the file SYS: SYS; DEFMIC LISP.)

The fifth and last instruction is similar to the fourth:

```
26 (MISC) ASSQ D-RETURN
```

What is new here is the new value of the destination field. This one is called D-RETURN, and it can be used anywhere destination fields in general can be used (like in MOVE instructions). Sending something to "Destination-Return" means that this value should be the returned value of the function, and that we should return from this function. This is a bit unusual in instruction sets; rather than having a "return" instruction, we have a destination that, when stored into, returns from the function. What this instruction does, then, is to invoke the Lisp function assq on the top two elements of the stack and return the result of assq as the result of this function.

Now, let's look at the program as a whole and see what it did:

```
22 MOVE D-PDL FEF|6          ;'KEY
23 MOVE D-PDL ARG|0          ;X
24 MOVE D-PDL FEF|7          ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

First it pushes the symbol key. Then it pushes the value of x. Then it pushes the symbol propname. Then it invokes get, which pops the value of x and the symbol propname off the stack and uses them as arguments, thus doing the equivalent of evaluating the form (get x 'propname). The result is left on the stack; the stack now contains the result of the get on top, and the symbol key underneath that. Next, it invokes assq on these two values, thus doing the equivalent of evaluating (assq 'key (get x 'propname)). Finally, it returns the value produced by assq. Now, the original Lisp program we compiled was:

```
(defun foo (x)
  (assq 'key (get x 'propname)))
```

We can see that the code produced by the compiler is correct: it will do the same thing as the function we defined will do.

In summary, we have seen two kinds of instructions so far: the MOVE instruction, which takes a destination and an address, and two of the large set of miscellaneous instructions, which take only a destination, and implicitly get their inputs from the stack. We have seen two destinations (D-PDL and D-RETURN), and two forms of address (FEF addressing and ARG addressing).

28.2 A More Advanced Example

Here is a more complex Lisp function, demonstrating local variables, function calling, conditional branching, and some other new instructions.

```
(defun bar (y)
  (let ((z (car y)))
    (cond ((atom z)
           (setq z (cdr y))
           (foo y))
          (t
           nil))))
```

The disassembled code looks like this:

```
20 CAR D-PDL ARG|0              ;Y
21 POP LOCAL|0                  ;Z
22 MOVE D-IGNORE LOCAL|0        ;Z
23 BR-NOT-ATOM 30
24 CDR D-PDL ARG|0              ;Y
25 POP LOCAL|0                  ;Z
26 CALL D-RETURN FEF|6          ;#'FOO
27 MOVE D-LAST ARG|0            ;Y
30 MOVE D-RETURN 'NIL
```

The first instruction here is a CAR instruction. It has the same format as MOVE: there is a destination and an address. The CAR instruction reads the datum addressed by the address, takes the car of it, and stores the result into the destination. In our example, the first instruction addresses the zeroth argument, and so it computes (car y); then it pushes the result onto the stack.

The next instruction is something new: the POP instruction. It has an address field, but it uses it as a destination rather than as a source. The POP instruction pops the top value off the stack, and stores that value into the address specified by the address field. In our example, the value on the top of the stack is popped off and stored into address LOCAL|0. This is a new form of address; it means the zeroth local variable. The ordering of the local variables is chosen by the compiler, and so it is not fully predictable, although it tends to be by order of appearance in the code; fortunately you never have to look at these numbers, because the "comment" field explains what is going on. In this case, the variable being addressed is z. So this instruction pops the top value on the stack into the variable z. The first two instructions work together to take the car of y and store it into z, which is indeed the first thing the function bar ought to do. (If you have two local variables with the same name, then the "comment" field won't tell you which of the two you're talking about; you'll have to figure that out yourself. You can tell two local variables with the same name apart by looking at the number in the address.)

The next instruction is a familiar MOVE instruction, but it uses a new destination: D-IGNORE. This means that the datum being addressed isn't moved anywhere. If so, then why bother doing this instruction? The reason is that there is conceptually a set of *indicator* bits, as in the PDP-11. Every instruction that moves or produces a datum sets the "indicator" bits from that datum so that following instructions can test them. So the reason that the MOVE instruction is being done is so that someone can test the "indicators" set up by the value that was moved, namely the value of z.

All instructions except the branch instructions set the "indicator" bits from the result produced and/or stored by that instruction. (In fact, the POP in instruction 21 set the "indicators" properly, and so the MOVE at instruction 22 is superfluous and the compiler would eliminate it.)

The next instruction is a conditional branch; it changes the flow of control, based on the values in the "indicator" bits. The instruction is BR-NOT-ATOM 30, which means "Branch, if the quantity was not an atom, to location 30; otherwise proceed with execution". If z was an atom, the Lisp Machine branches to location 30, and execution proceeds there. (As you can see by skipping ahead, location 30 just contains a MOVE instruction, which will cause the function to return nil.)

If z is not an atom, the program keeps going, and the CDR instruction is next. This is just like the CAR instruction except that it takes the cdr; this instruction pushes the value of (cdr y) onto the stack. The next one pops that value off into the variable z.

There are just two more instructions left. These two instructions will be our first example of how function calling is compiled. It is the only really tricky thing in the instruction set. Here is how it works in our example:

```
26 CALL D-RETURN FEF|6          ;#'FOO
27 MOVE D-LAST ARG|0            ;Y
```

The form being compiled here is (foo y). This means we are applying the function which is in the function cell of the symbol foo, and passing it one argument, the value of y. The way function calling works is in the following three steps. First of all, there is a CALL instruction that specifies the function object being applied to arguments. This creates a new stack frame on the stack, and stores the function object there. Secondly, all the arguments being passed except the last one are pushed onto the stack. Thirdly and lastly, the last argument is sent to a special destination, called D-LAST, meaning "this is the last argument". Storing to this destination is what actually calls the function, *not* the CALL instruction itself.

There are two things you might wonder about this. First of all, when the function returns, what happens to the returned value? Well, this is what we use the destination field of the CALL instruction for. The destination of the CALL is not stored into at the time the CALL instruction is executed; instead, it is saved on the stack along with the function operation (in the stack frame created by the CALL instruction). Then, when the function actually returns, its result is stored into that destination.

The other question is what happens when there isn't any last argument; that is, when there is a call with no arguments at all? This is handled by a special instruction called CALL0. The address of CALL0 addresses the function object to be called; the call takes place immediately and the result is stored into the destination specified by the destination field of the CALL0 instruction.

So, let's look at the two-instruction sequence above. The first instruction is a CALL; the function object it specifies is at FEF|6, which the comment tells us is the contents of the function cell of foo (the FEF contains an invisible pointer to that function cell). The destination field of the CALL is D-RETURN, but we aren't going to store into it yet; we will save it away in the stack frame and use it later. So the function doesn't return at this point, even though it says D-RETURN in the instruction; this is the tricky part.

Next we have to push all the arguments except the last one. Well, there's only one argument, so nothing needs to be done here. Finally, we move the last argument (that is, the only argument: the value of y) to D-LAST, using the MOVE instruction. Moving to D-LAST is what actually invokes the function, so at this point the function foo is invoked. When it returns, its result is sent to the destination stored in the stack frame: D-RETURN. Therefore, the value returned by the call to foo will be returned as the value of the function bar. Sure enough, this is what the original Lisp code says to do.

When the compiler pushes arguments to a function call, it sometimes does it by sending the values to a destination called D-NEXT (meaning the "next" argument). This is exactly the same as D-PDL when producing a compiled function. The distinction is important when the compiler output is passed to the microcompiler to generate microcode.

Here is another example to illustrate function calling. This Lisp function calls one function on the results of another function.

```
(defun a (x y)
  (b (c x y) y))
```

The disassembled code looks like this:

```
22 CALL D-RETURN FEF|6          ;#'B
23 CALL D-PDL FEF|7             ;#'C
24 MOVE D-PDL ARG|0             ;X
25 MOVE D-LAST ARG|1            ;Y
26 MOVE D-LAST ARG|1            ;Y
```

The first instruction starts off the call to the function b. The destination field is saved for later: when this function returns, we will return its result as a's result. Next, the call to c is started. Its destination field, too, is saved for later; when c returns, its result should be pushed onto the stack, so that it will be the next argument to b. Next, the first and second arguments to c are passed; the second one is sent to D-LAST and so the function c is called. Its result, as we said, will be pushed onto the stack, and thus become the first argument to b. Finally, the second argument to b is passed, by storing in D-LAST; b gets called, and its result is sent to D-RETURN and is returned from a.

28.3 The Rest of the Instructions

Now that we've gotten some of the feel for what is going on, I will start enumerating the instructions in the instruction set. The instructions fall into four classes. Class I instructions have both a destination and an address. Class II instructions have an address, but no destination. Class III instructions are the branch instructions, which contain a branch address rather than a general base-and-offset address. Class IV instructions have a destination, but no address; these are the miscellaneous instructions.

We have already seen just about all the Class I instructions. There are nine of them in all: MOVE, CALL, CALL0, CAR, CDR, CAAR, CADR, CDAR, and CDDR. MOVE just moves a datum from an address to a destination; the CxR and CxxR instructions are the same but perform the function on the value before sending it to the destination; CALL starts off a call to a function with some arguments; CALL0 performs a call to a function with no arguments.

We've seen most of the possible forms of address. So far we have seen the FEF, ARG, and LOCAL base registers. There are two other kinds of addresses. One uses a "constant" base register, which addresses a set of standard constants: NIL, T, 0, 1, and 2. The disassembler doesn't even bother to print out CONSTANT|*n*, since the number *n* would not be even slightly interesting; it just prints out 'NIL or '1 or whatever. The other kind of address is a special one

printed as PDL-POP, which means that to read the value at this address, an object should be popped off the top of the stack.

There is a higher number of Class II instructions. The only one we've seen so far is POP, which pops a value off the stack and stores it into the specified address. There is another instruction called MOVEM (from the PDP-10 opcode name, meaning MOVE to Memory), which stores the top element of the stack into the specified address, but doesn't pop it off the stack.

Then there are seven Class II instructions to implement heavily-used two-argument functions: +, -, *, /, LOGAND, LOGXOR, and LOGIOR. These instructions take their first argument from the top of the stack (popping them off) and their second argument from the specified address, and they push their result on the stack. Thus the stack level does not change due to these instructions.

Here is a small function that shows some of these new things:

```
(defun foo (x y)
  (setq x (logxor y (- x 2))))
```

The disassembled code looks like this:

```
16 MOVE D-PDL ARG|1            ;Y
17 MOVE D-PDL ARG|0            ;X
20 - '2
21 LOGXOR PDL-POP
22 MOVEM ARG|0                 ;X
23 MOVE D-RETURN PDL-POP
```

Instructions 20 and 21 use two of the new Class II instructions: the - and LOGXOR instructions. Instructions 21 and 23 use the PDL-POP address type, and instruction 20 uses the "constant" base register to get to a fixnum 2. Finally, instruction 22 uses the MOVEM instruction; the compiler wants to use the top value of the stack to store it into the value of x, but it doesn't want to pop it off the stack because it has another use for it: to return it from the function.

Another four Class II instructions implement some commonly used predicates: =, >, <, and EQ. The two arguments come from the top of the stack and the specified address; the stack is popped, the predicate is applied to the two objects, and the result is left in the "indicators" so that a branch instruction can test it, and branch based on the result of the comparison. These instructions remove the top item on the stack and don't put anything back, unlike the previous set, which put their results back on the stack.

Next, there are four Class II instructions to read, modify, and write a quantity in ways that are common in Lisp code. These instructions are called SETE-CDR, SETE-CDDR, SETE-1+, and SETE-1-. The SETE- means to set the addressed value to the result of applying the specified one-argument function to the present value. For example, SETE-CDR means to read the value addressed, apply cdr to it, and store the result back in the specified address. This is used when compiling (setq x (cdr x)), which commonly occurs in loops; the other functions are used frequently in loops, too.

There are two instructions used to bind special variables. The first is BIND-NIL, which binds the cell addressed by the address field to nil; the second is BIND-POP, which binds the cell to an object popped off the stack rather than nil. The latter instruction pops a value off the stack; the former does not use the stack at all.

There are two instructions to store common values into addressed cells. SET-NIL stores nil into the cell specified by the address field; SET-ZERO stores 0. Neither instruction uses the stack at all.

Finally, the PUSH-E instruction creates a locative pointer to the cell addressed by the specified address, and pushes it onto the stack. This is used in compiling (value-cell-location 'z) where z is an argument or a local variable, rather than a symbol (special variable).

Those are all of the Class II instructions. Here is a contrived example that uses some of the ones we haven't seen, just to show you what they look like:

```
(declare (special *foo* *bar*))

(defun weird (x y)
  (cond ((= x y)
         (let ((*foo* nil) (*bar* 5))
           (setq x (cdr x)))
         nil)
        (t
         (setq x nil)
         (caar (value-cell-location 'y)))))
```

The disassembled code looks like this:

```
24 MOVE D-PDL ARG|0            ;X
25 = ARG|1                     ;Y
26 BR-NIL 35
27 BIND-NIL FEF|6              ;*FOO*
30 MOVE D-PDL FEF|8            ;'5
31 BIND-POP FEF|7              ;*BAR*
32 SETE-CDR ARG|0              ;X
33 (MISC) UNBIND 2 bindings
34 MOVE D-RETURN 'NIL
35 SET-NIL ARG|0               ;X
36 PUSH-E ARG|1                ;Y
37 CAAR D-RETURN PDL-POP
```

Instruction 25 is an = instruction; it numerically compares the top of the stack, x, with the addressed quantity, y. The x is popped off the stack, and the indicators are set to the result of the equality test. Instruction 26 checks the indicators, branching to 35 if the result of the call to = was NIL; that is, the machine will branch to 35 if the two values were not equal. Instruction 27 binds *foo* to nil; instructions 30 and 31 bind *bar* to 5. Instruction 32 demonstrates the use of SETE-CDR to compile (setq x (cdr x)), and instruction 35 demonstrates the use of SET-NIL to compile (setq x nil). Instruction 36 demonstrates the use of PUSH-E to compile (value-

cell-location 'y).

The next class of instructions, Class III, are the branching instructions. These have neither addresses nor destinations of the usual sort. Instead, they have branch-addresses; they say where to branch, if the branch is going to happen. There are several instructions, differing in the conditions under which they branch and whether they pop the stack. Branch-addresses are stored internally as self-relative addresses, to make Lisp Machine code relocatable, but the disassembler does the addition for you and prints out FEF-relative addresses so that you can easily see where the branch is going to.

The branch instructions we have seen so far decide whether to branch on the basis of the "nil indicator", that is, whether the last value dealt with was nil or non-nil. BR-NIL branches if it was nil, and BR-NOT-NIL branches if it was not nil. There are two more instructions that test the result of the atom predicate on the last value dealt with. BR-ATOM branches if the value was an atom (that is, if it was anything besides a cons), and BR-NOT-ATOM branches if the value was not an atom (that is, if it was a cons). The BR instruction is an unconditional branch (it always branches).

None of the above branching instructions deal with the stack. There are two more instructions called BR-NIL-POP and BR-NOT-NIL-POP, which are the same as BR-NIL and BR-NOT-NIL except that if the branch is not done, the top value on the stack is popped off the stack. These are used for compiling and and or special forms.

Finally, there are the Class IV instructions, most of which are miscellaneous hand-microcoded Lisp functions. The file SYS: SYS; DEFMIC LISP has a list of all the miscellaneous instructions. Most correspond to Lisp functions, including the subprimitives, although some of these functions are very low level internals that may not be documented anywhere (don't be disappointed if you don't understand all of them). Please do not look at this file in hopes of finding obscure functions that you think you can use to speed up your programs; in fact, the compiler automatically uses these things when it can, and directly calling weird internal functions will only serve to make your code hard to read, without making it any faster. In fact, we don't guarantee that calling undocumented functions will continue to work in the future.

The DEFMIC file can be useful for determining if a given function is in microcode, although the only definitive way to tell is to compile some code that uses it and look at the results, since sometimes the compiler converts a documented function with one name into an undocumented one with another name.

28.4 Function Entry

When a function is first entered in the Lisp Machine, interesting things can happen because of the features that are invoked by use of the various "&" keywords. The microcode performs various services when a function is entered, even before the first instruction of the function is executed. These services are called for by various fields of the header portion of the FEF, including a list called the *Argument Descriptor List*, or *ADL*. We won't go into the detailed format of any of this, as it is complex and the details are not too interesting. Disassembling a function that makes use of the ADL prints a summary of what the ADL says to do, before the beginning of the code.

The function-entry services include the initialization of unsupplied optional arguments and of &AUX variables. The ADL has a little instruction set of its own, and if the form that computes the initial value is something simple, such as a constant or a variable, then the ADL can handle things itself. However, if things get too complicated, instructions are needed, and the compiler generates some instructions at the front of the function to initialize the unsupplied variables. In this case, the ADL specifies several different starting addresses for the function, depending on which optional arguments have been supplied and which have been omitted. If all the optional arguments are supplied, then the ADL starts the function off after all the instructions that would have initialized the optional arguments; since the arguments were supplied, their values should not be set, and so all these instructions are skipped over. Here's an example:

```
(declare (special *y*))

(defun foo (&optional (x (car *y*)) (z (* x 3)))
  (cons x z)
```

The disassembled code looks like this:

```
Arg 0 (X) is optional, local,
  initialized by the code up to pc 34.
Arg 1 (Z) is optional, local,
  initialized by the code up to pc 37.

32 CAR D-PDL FEF|6              ;*Y*
33 POP ARG|0                    ;X
34 MOVE D-PDL ARG|0             ;X
35 * FEF|11                     ;'3
36 POP ARG|1                    ;Z
37 MOVE D-PDL ARG|0             ;X
40 MOVE D-PDL ARG|1             ;Z
41 (MISC) CONS D-RETURN
```

If no arguments are supplied, the function will be started at instruction 32; if only one argument is supplied, it will be started at instruction 34; if both arguments are supplied, it will be started at instruction 37.

The thing to keep in mind here is that when there is initialization of variables, you may see it as code at the beginning of the function, or you may not, depending upon whether it is too complex for the ADL to handle. This is true of **&aux** variables as well as unsupplied **&optional** arguments.

When there is a **&rest** argument, it is passed to the function as the zeroth local variable, rather than as any of the arguments. This is not really so confusing as it might seem, since a **&rest** argument is not an argument passed by the caller; rather it is a list of some of the arguments, created by the function-entry microcode services. In any case the "comment" tells you what is going on. In fact, one hardly ever looks much at the address fields in disassembled code, since the "comment" tells you the right thing anyway. Here is a silly example of the use of a **&rest** argument:

```
(defun prod (&rest values)
  (apply #'* values))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6           ;#'*
21 MOVE D-PDL LOCAL|0         ;VALUES
22 (MISC) APPLY D-RETURN
```

As can be seen, values is referred to as LOCAL|0.

Another thing the microcode does at function entry is to bind the values of any arguments or &aux variables that are special. Thus, you won't see BIND instructions doing this, but it is still being done.

28.5 Special Class IV Instructions

We said earlier that most of the Class IV instructions are miscellaneous hand-microcoded Lisp functions. However, a few of them are not Lisp functions at all. There are two instructions that are printed as UNBIND 3 bindings or POP 7 values; the number can be anything up to 16 (these numbers are printed in decimal). These instructions just do what they say, unbinding the last *n* values that were bound or popping the top *n* values off the stack.

The array referencing functions—aref, aset, and aloc—take a variable number of arguments, but they are handled differently depending on how many there are. For one-, two-, and three-dimensional arrays, these functions are turned into internal functions with names ar-1, as-1, and ap-1 (with the number of dimensions substituted for 1). Again, there is no point in using these functions yourself; it would only make your code harder to understand but not any faster at all. When there are more than three dimensions, the functions aref, aset and aloc are called in the ordinary manner.

When you call a function and expect to get more than one value back, a slightly different kind of function calling is used. Here is an example that uses multiple-value to get two values back from a function call:

```
(defun foo (x)
  (let (y z)
    (multiple-value (y z)
      (bar 3))
    (+ x y z)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6            ;#'BAR
23 MOVE D-PDL '2
24 (MISC) %CALL-MULT-VALUE D-IGNORE
25 MOVE D-LAST FEF|7           ;'3
26 POP LOCAL|1                 ;Z
27 POP LOCAL|0                 ;Y
30 MOVE D-PDL ARG|0            ;X
31 + LOCAL|0                   ;Y
32 + LOCAL|1                   ;Z
33 MOVE D-RETURN PDL-POP
```

A %CALL-MULT-VALUE instruction is used instead of a CALL instruction. The destination field of %CALL-MULT-VALUE is unused and will always be D-IGNORE. %CALL-MULT-VALUE takes two "arguments", which it finds on the stack; it pops both of them. The first one is the function object to be applied; the second is the number of return values that are expected. The rest of the call proceeds as usual, but when the call returns, the returned values are left on the stack. The number of objects left on the stack is always the same as the second "argument" to %CALL-MULT-VALUE. In our example, the two values returned are left on the stack, and they are immediately popped off into z and y. There is also a %CALL0-MULT-VALUE instruction, for the same reason CALL0 exists.

The multiple-value-bind form works similarly; here is an example:

```
(defun foo (x)
  (multiple-value-bind (y *foo* z)
      (bar 3)
    (+ x y z)))
```

The disassembled code looks like this:

```
24 MOVE D-PDL FEF|8            ;#'BAR
25 MOVE D-PDL FEF|7            ;'3
26 (MISC) %CALL-MULT-VALUE D-IGNORE
27 MOVE D-LAST FEF|7           ;'3
30 POP LOCAL|1                 ;Z
31 BIND-POP FEF|6              ;*FOO*
32 POP LOCAL|0                 ;Y
33 MOVE D-PDL ARG|0            ;X
34 + LOCAL|0                   ;Y
35 + LOCAL|1                   ;Z
36 MOVE D-RETURN PDL-POP
```

The %CALL-MULT-VALUE instruction is still used, leaving the results on the stack; these results are used to bind the variables.

Calls done with multiple-value-list work with the %CALL-MULT-VALUE-LIST instruction. It takes one "argument" on the stack: the function object to apply. When the function returns, the list of values is left on the top of the stack. Here is an example:

```
(defun foo (x y)
  (multiple-value-list (foo 3 y x)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6              ;#'FOO
23 (MISC) %CALL-MULT-VALUE-LIST D-IGNORE
24 MOVE D-PDL FEF|7              ;'3
25 MOVE D-PDL ARG|1              ;Y
26 MOVE D-LAST ARG|0             ;X
27 MOVE D-RETURN PDL-POP
```

Returning of more than one value from a function is handled by special miscellaneous instructions. %RETURN-2 and %RETURN-3 are used to return two or three values; these instructions take two and three arguments, respectively, on the stack and return from the current function just as storing to D-RETURN would. If there are more than three return values, they are all pushed, then the number that there were is pushed, and then the %RETURN-N instruction is executed. None of these instructions use their destination field. Note: the **return-list** function is just an ordinary miscellaneous instruction; it takes the list of values to return as an argument on the stack and returns those values from the current function.

The function **lexpr-funcall** is compiled using a special instruction called %SPREAD to iterate over the elements of its last argument, which should be a list. %SPREAD takes one argument (on the stack), which is a list of values to be passed as arguments (pushed on the stack). If the destination of %SPREAD is D-PDL (or D-NEXT), then the values are just pushed; if it is D-LAST, then after the values are pushed, the function is invoked. **lexpr-funcall** will always compile using a %SPREAD whose destination is D-LAST. Here is an example:

```
(defun foo (a b &rest c)
  (lexpr-funcall #'format t a c)
  b)
```

The disassembled code looks like this:

```
20 CALL D-IGNORE FEF|6           ;#'FORMAT
21 MOVE D-PDL 'T
22 MOVE D-PDL ARG|0              ;A
23 MOVE D-PDL LOCAL|0            ;C
24 (MISC) %SPREAD D-LAST
25 MOVE D-RETURN ARG|1           ;B
```

Note that in instruction 23, the address **LOCAL|0** is used to access the **&rest** argument.

The ***catch** special form is also handled specially by the compiler. Here is a simple example of ***catch**:

```
(defun a ()
  (*catch 'foo (bar)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6            ;'26
23 (MISC) %CATCH-OPEN D-PDL
24 MOVE D-PDL FEF|7            ;'FOO
25 CALL0 D-LAST FEF|8          ;#'BAR
26 MOVE D-RETURN PDL-POP
```

The %CATCH-OPEN instruction is like the CALL instruction; it starts a call to the *catch function. It takes one "argument" on the stack, which is the location in the program that should be branched to if this *catch is *thrown to. In addition to saving that program location, the instruction saves the state of the stack and of special-variable binding so that they can be restored in the event of a *throw. So instructions 22 and 23 start a *catch block, and the rest of the function passes its two arguments. The *catch function itself simply returns its second argument; but if a *throw happens during the evaluation of the (bar) form, then the stack will be unwound and execution will resume at instruction 26. The destination field of %CATCH-OPEN is like that of CALL; it is saved on the stack, and controls what will be done with the result of the call to the *catch. Note that even though *catch is really a Lisp special form, it is compiled more or less as if it were a function of two arguments.

To allow compilation of (multiple-value (...) (*catch ...)), there is a special instruction called %CATCH-OPEN-MULT-VALUE, which is a cross between %CATCH-OPEN and %CALL-MULT-VALUE. multiple-value-list with *catch works by asking for all four values and passing them to the function list.

28.6 Estimating Run Time

You may sometimes want to estimate the speed at which a function will execute by examination of the compiled code. This section gives some rough guidelines to the relative cost of various instructions; the actual speed may vary from these estimates by as much as a factor of two. Some of these speeds vary with time; they speed up as work is done to improve system efficiency and slow down sometimes when sweeping changes are made (for instance, when garbage collection was introduced it slowed down some operations even when garbage collection is not turned on). However these changes are usually much less than a factor of two.

It is also important to realize that in many programs the execution time is determined by paging rather than by CPU run time. The cost of paging is unfortunately harder to estimate than run time, because it depends on dynamic program behavior and locality of data structure.

On a conventional computer such as the PDP-10, rough estimates of the run time of compiled code are fairly easy to make. It is a reasonable approximation to assume that all machine instructions take about the same amount of time to execute. When the compiler generates a call to a runtime support routine, the user can estimate the speed of that routine since it is implemented in the same instructions as the user's compiled program. Actual speeds can vary widely because of data dependencies; for example, when using the plus function the operation will be much slower if an argument is a bignum than if the arguments are all fixnums. However, in Maclisp most performance-critical functions use declarations to remove such data dependencies, because generic, data-dependent operations are so much slower than type-specific operations.

Things are different in the Lisp Machine. The instruction set we have just seen is a high-level instruction set. Rather than specifying each individual machine operation, the compiler calls for higher-level Lisp operations such as cdr or memq. This means that some instructions take many times longer to execute than others. Furthermore, in the Lisp machine we do not use data-type declarations. Instead the machine is designed so that all instructions can be *generic*; that is, they determine the types of their operands at run time. This means that there are data dependencies that can have major effects on the speed of execution of an instruction. For instance, the + instruction is quite fast if both operands turn out to be fixnums, but much slower if they are bignums.

The Lisp machine also has a large amount of microcode, both to implement certain Lisp functions and to assist with common operations such as function calling. It is not as easy for a user to read microcode and estimate its speed as it is with compiled code, although the Lisp machine has a much more readable microcode than most computers.

In this section we give some estimates of the speed of various operations. There are also facilities for measuring the actual achieved speed of a program. These will not be documented here as they are currently being changed.

We will express all times in terms of the time to execute the simplest instruction, MOVE D-PDL ARG|0. This time is about two microseconds and will be called a "unit".

MOVE takes the same amount of time if the destination is D-IGNORE or D-NEXT, or if the address is a LOCAL or PDL-POP rather than an ARG. A MOVE that references a constant, via either the FEF base register or the CONSTANT base register, takes about two units. A MOVE that references a special variable by means of the FEF base register and an invisible pointer takes closer to three units.

Use of a complex destination (D-LAST, D-RETURN, or D-NEXT-LIST) takes extra time because of the extra work it has to do: calling a function, returning from a function, or doing the bookkeeping associated with forming a list. These costs will be discussed a bit later.

The other Class I instructions take longer than MOVE. Each memory reference required by car/cdr operations costs about one unit. Note that cdr requires one memory cycle if the list is compactly cdr-coded and two cycles if it is not. The CALL instruction takes three units. The CALL0 instruction takes more, of course, since it actually calls the function.

The Class II (no destination) instructions vary. The MOVEM and POP operations take about one unit. (Of course they take more if FEF or CONSTANT addressing is used.) The arithmetic and logical operations and the predicates take two units when applied to fixnums, except for multiplication and division which take five. The SETE-1+ and SETE-1- instructions take two units, the same time as a push followed by a pop; i.e. (setq x (1+ x)) takes the same amount of time as (setq x y). The SET-NIL and SET-ZERO instructions take one unit. The special-variable binding instructions take several units.

A branch takes between one and two units.

The cost of calling a function with no arguments and no local variables that doesn't do anything but return nil is about 15 units (7 cdrs or additions). This is the cost of a CALL FEF|*n* instruction, a MOVE to D-LAST, the simplest form of function-entry services, and a MOVE to D-RETURN. If the function takes arguments the cost of calling the function includes the cost of the instructions in the caller that compute the arguments. If the function has local variables initialized to nil or optional arguments defaulted to nil there is a negligible additional cost. The cost of having a &rest argument is less than one additional unit. But if the function binds special variables there is an additional cost of 8 units per variable (this includes both binding the variables on entry and unbinding them on return).

If the function needs an ADL, typically because of complex optional-argument initializations, the cost goes up substantially. It's hard to characterize just how much it goes up by, since this depends on what you do. Also calling for multiple values is more expensive than simple calling.

We consider the cost of calling functions to be somewhat higher than it should be, and would like to improve it. But this might require incompatible system architecture changes and probably will not happen, at least not soon.

Flonum and bignum arithmetic are naturally slower than fixnum arithmetic. For instance, flonum addition takes 8 units more than fixnum addition, and addition of 60-bit bignums takes 15 units more. Note that these times include some garbage-collection overhead for the intermediate results which have to be created in memory. Fixnums and small flonums do not take up any memory and avoid this overhead. Thus small-flonum addition takes only about 2 units more than fixnum addition. This garbage-collection overhead is of the "extra-pdl-area" sort rather than the full Baker garbage collector sort; if you don't understand this don't worry about it for now.

Floating-point subtraction, multiplication, and division take just about the same time as floating-point addition. Floating-point execution times can be as many as 3 units longer depending on the arguments.

The run time of a Class IV (or miscellaneous) instruction depends on the instruction and its arguments. The simplest instructions, predicates such as atom and numberp, take 2 units. This is basically the overhead for doing a Class IV instruction. The cost of a more complex instruction can be estimated by looking at what it has to do. You can get a reasonable estimate by charging one unit per memory reference, car operation, or cdr-coded cdr operation. A non-cdr-coded cdr operation takes two units. For instance, (memq 'nil '(a b c)) takes 13 units, of which 4 are pushing the arguments on the stack, 2 are Class IV instruction overhead, 6 are accounted for by cars and cdrs, and 1 is "left over".

The cost of array accessing depends on the type of array and the number of dimensions. aref of a 1-dimensional non-indirect art-q array takes 6 units and aset takes 5 units, not counting pushing the arguments onto the stack. (These are the costs of the AR-1 and AS-1 instructions.) A 2-dimensional array takes 6 units more. aref of a numeric array takes the same amount of time as aref of an art-q array. aset takes 1 unit longer. aref of an art-float array takes 5 units longer than aref of an art-q array. aset takes 3 units longer.

The functions copy-array-contents and copy-array-portion optimize their array accessing to remove overhead from the inner loop. copy-array-contents of an art-q array has a startup overhead of 8 units, not including pushing the arguments, then costs just over 2 units per array

element.

The cons function takes 7 units if garbage collection is turned off. (list a b c d) takes 24 units, which includes 4 units for getting the local variables a, b, c, and d.