Revised Internal Design of Spice Lisp

Skef Wholey
Scott E. Fahlman
Joseph Ginder

DRAFT

Spice Document S026 [Revised]
Keywords and index categories: Lisp
Location of machine-readable file: CMUC::<Wholey.Australia>Revguts.Mss

Table of Contents

Acknowledgments

The following people have been contributors to this and earlier versions of the design of the Spice Lisp instruction set: Guy L. Steele Jr., Gail E. Kaiser, Walter van Roggen, Neal Feinberg, Jim Large, and Rob MacLachlan. The original instruction set was loosely based on the MIT Lisp Machine instruction set.

The FASL file format was designed by Guy L. Steele Jr. and Walter van Roggen, and the appendix on this subject is their document with very few modifications.

1. Introduction

1.1. Scope and Purpose

NOTE: This document describes a new implementation of Spice Lisp as it is to be implemented on the PERQ, running the Spice operating system, Accent. This new design is undergoing rapid change, and for the present is not guaranteed to accurately describe any past, present, or future implementation of Spice Lisp. All questions and comments on this material should be directed to Skef Wholey (Wholey@CMU-CS-C).

This document specifies the instruction set and virtual memory architecture of the PERQ Spice Lisp system. This is a working document, and it will change frequently as the system is developed and maintained. If some detail of the system does not agree with what is specified here, it is to be considered a bug.

Spice Lisp will be implemented on other microcodable machines, and implementations of Common Lisp based on Spice Lisp exist for conventional machines with fixed instructions sets. These other implementations are very different internally and are described in other documents.

1.2. Notational Conventions

Spice Lisp objects are 32 bits long. The low-order bit of each word is numbered 0; the high-order bit is numbered 31. If a word is broken into smaller units, these are packed into the word from right to left. For example, if we break a word into bytes, byte 0 would occupy bits 0-7, byte 1 would occupy 8-15, byte 2 would occupy 16-23, and byte 3 would occupy 24-31. In these conventions we follow the conventions of the VAX; the PDP-10 family follows the opposite convention, packing and numbering left to right.

All Spice Lisp documentation uses decimal as the default radix; other radices will be indicated by a subscript (as in 77_8) or by a clear statement of what radix is in use.

2. Data Types and Object Formats

2.1. Lisp Objects

Lisp objects are 32 bits long. They come in 32 basic types, divided into three classes: immediate data types, pointer types, and forwarding pointer types. The storage formats are as follows:

Immediate Data Types:

```
 31           27 26                                                    0
-------------------------------------------------------------------------
| Type Code (5) |              Immediate Data (27)                      |
-------------------------------------------------------------------------
```

Pointer and Forwarding Types:

```
 31           27 26           25 24                     1            0
-------------------------------------------------------------------------
| Type Code (5) | Space Code (2) |    Pointer (24)        | Unused (1) |
-------------------------------------------------------------------------
```

2.2. Table of Type Codes

Code	Type	Class	Explanation
0	Misc	Immediate	Trap object, stacks, system tables
1	Bit-Vector	Pointer	Vector of bits
2	Integer-Vector	Pointer	Vector of integers
3	String	Pointer	Character string
4	Bignum	Pointer	Bignum
5	Long-Float	Pointer	Long float
6	Complex	Pointer	Complex number
7	Ratio	Pointer	Ratio
8	General-Vector	Pointer	Vector of Lisp objects
9	Function	Pointer	Compiled function header
10	Array	Pointer	Array header
11	Symbol	Pointer	Symbol
12	List	Pointer	Cons cell
13-15	Unused		
16	+ Fixnum	Immediate	Fixnum >= 0
17	- Fixnum	Immediate	Fixnum < 0
18	+ Short-Float	Immediate	Short float >= 0
19	- Short-Float	Immediate	Short float < 0
20	Character	Immediate	Character object
21	Values-Marker	Immediate	Multiple values marker
22	Call-Header	Immediate	Control stack call frame header
23	Catch-Header	Immediate	Control stack catch frame header
24	Catch-All	Immediate	Catch-All object
25	GC-Forward	Forward	Object in newspace of same type
26-31	Unused		

2.3. Table of Space Codes

Code	Space	Explanation
0	Dynamic-0	Storage normally garbage collected, space 0.
1	Dynamic-1	Storage normally garbage collected, space 1.
2	Static	Permanent objects, never moved or reclaimed.
3	Read-Only	Objects never moved, reclaimed, or altered.

2.4. Immediate Data Type Descriptions

Misc — Reserved for assorted internal values. Bits 25-26 specify a sub-type code.

0 Trap — Illegal object trap. If you fetch one of these, it's an error except under very specialized conditions. Note that a word of all zeros is of this type, so this is useful for trapping references to uninitialized memory. This value also is used in symbols to flag an undefined value or definition.

1 Control-Stack-Pointer — The low 25 bits are a pointer into the control stack. This is a word pointer that points to the proper virtual memory address. Pointers of this form are returned by certain system routines for use by debugging programs.

2 Binding-Stack-Pointer — The low 25 bits are a pointer into the binding stack. This is a word pointer that points to the proper virtual memory address. Pointers of this form are returned by certain system routines for use by debugging programs.

3 System-Table-Pointer — The low 25 bits are a pointer into an area of memory used for system tables. This is a word pointer into an area of the address space reserved for data sent and received in Accent messages.

Fixnum — A 28-bit two's complement integer. The sign bit is stored as part of the type code.

Short-Float — As with fixnums, the sign bit is stored as part of the type code. The format of short floating point number can be viewed as follows:

```
 31           28     27     26        19 18               0
------------------------------------------------------------
| Type code (4) | Sign (1) |  Expt (8)  |   Mantissa (19)  |
------------------------------------------------------------
```

The sign of the mantissa is moved to the left so that these flonums can be compared just like fixnums. The exponent is the binary two's complement exponent of the number, plus 128; then, if the mantissa is negative, the bits of the exponent field are inverted. The mantissa is a 21-bit two's complement number with the sign moved to bit 27 and the leading significant bit (which is always the complement of the sign bit and hence carries no information) stripped off. The short flonum representing 0.0 has 0's in bits 0 - 27. It is illegal for the sign bit to be 1 with all the other bits equal to 0. This encoding gives a range of about 10^{-38} to 10^{+38} and about 6 digits of accuracy. Note that long-flonums are available for those wanting more accuracy, but they are less efficient to use because they generate garbage that must be collected later.

Character

A character object holding a character code, control bits, and font in the following format:

```
 31           27 26        24 23      16 15       8 7        0
---------------------------------------------------------------
| Type code (5) | Unused (3) | Font (8) | Bits (8) | Code (8) |
---------------------------------------------------------------
```

Values-Marker

Used to mark the presence of multiple values on the stack. The low 16 bits indicate how many values are being returned. Note then, that only 65535 values can be returned from a multiple-values producing form. These are pushed in order, then the Values-Marker is pushed.

Call-Header

Marks the start of each call frame on the control stack. The low-order 27 bits are used as a place to stash information for certain special kinds of calls.

For a normal function call, as created by the CALL or CALL-0 instruction, the low 27 bits are always 0.

Bit 22, if 1, indicates an ``escape to macro'' call frame, created when a macro-instruction cannot be completed entirely within the microcode. In this case, bits 16-17 indicate where the result is supposed to go (see section 6.3).

Bit 21, if 1, indicates a call frame that will accept multiple values to be returned. Such frames are created by Call-Multiple, and cause Return to take certain special actions. See section 6.1.3 for details.

	Bits 22 and 21 are mutually exclusive. It is undefined what happens when both of these are on at once.
Catch-Header	Marks a catch frame on the control stack. If bit 21 is on, this indicates that the catching form will accept multiple values. See section 6.2 for details.
Catch-All	Object used as the catch tag for unwind-protects. Special things happen when a catch frame with this as its tag is encountered during a throw. See section 6.2 for details.

2.5. Pointer-Type Objects and Spaces

Each of the pointer-type lisp objects points into a different space in virtual memory. There are separate spaces for Bit-Vectors, Symbols, Lists, and so on. The 5-bit type-code provides the high-order virtual address bits for the object, followed by the 2-bit space code, followed by the 25-bit pointer address. This gives a 31-bit virtual address to a 32-bit word; since the PERQ is a word-addressed machine, the low-order bit will be 0, and under Accent, the high order bit will be 0 (because the operating system lives in the upper half of the address space). This leaves us with a 30-bit virtual address. In effect we have carved a 30-bit space into a fixed set of 24-bit subspaces, not all of which are used.

The space code divides each of the type spaces into four sub-spaces, as shown in the table above. At any given time, one of the dynamic spaces is considered newspace, while the other is oldspace. The garbage collector continuously moves accessible objects from oldspace into newspace. When oldspace contains no more accessible objects it is considered empty. A ``flip'' can then be done, turning the old newspace into the new oldspace. All type-spaces are flipped at once. Allocation of new dynamic objects always occurs in newspace.

Optionally, the user (or system functions) may allocate objects in static or read-only space. Such objects are never reclaimed once they are allocated -- they occupy the space in which they were initially allocated for the lifetime of the Lisp process. The advantage of static allocation is that the GC never has to move these objects, thereby saving a significant amount of work, especially if the objects are large. Objects in read-only space are static, in that they are never moved or reclaimed; in addition, they cannot be altered once they are set up. Pointers in read-only space may only point to read-only or static space, never to dynamic space. This saves even more work, since read-only space does not need to be scavenged, and pages of read-only material do not need to be written back onto the disk during paging.

Objects in a particular type-space will contain either pointers to garbage-collectable objects or words of raw non-garbage-collectable bits, but not both. Similarly, a space will contain either fixed-length objects or variable-length objects, but not both. A variable-length object always contains a 24-bit length field right-justified in the first word, with the fixnum type-code in the high-order four bits. The remaining four bits can be used for sub-type information. The length field gives the size of the object in 32-bit words, including the header word. The garbage collector needs this information when the object is moved, and it is also useful for bounds checking.

The format of objects in each space are as follows:

Symbol	Each symbol is represented as a fixed-length block of boxed Lisp cells. The number of cells per symbol is 5, in the following order: 0 Value cell for shallow binding. 1 Definition cell: a function or list. 2 Property list: a list of attribute-value pairs. 3 Print name: a string. 4 Package: the obarray holding this symbol.
List	A fixed-length block of two boxed Lisp cells, the CAR and the CDR.
General-Vector	Vector of lisp objects, any length. The first word is a fixnum giving the number of words allocated for the vector (up to 24 bits). The highest legal index is this number minus 2. The second word is vector entry 0, and additional entries are allocated contiguously in virtual memory. General vectors are sometimes called G-Vectors. (See section 2.8 for further details.)
Integer-Vector	Vector of integers, any length. The 24 low bits of the first word give the allocated length in 32-bit words. The low-order 28 bits of the second word gives the length of the vector in entries, whatever the length of the individual entries may be. The high-order 4 bits of the second word contain access-type information that yields, among other things, the number of bits per entry. Entry 0 is right-justified in the third word of the vector. Bits per entry will normally be powers of 2, so they will fit neatly into 32-bit words, but if necessary some empty space may be left at the high-order end of each word. Integer vectors are sometimes called I-Vectors. (See section 2.8 for details.)
Bit-Vector	Vector of bits, any length. Bit-Vectors are represented in a form identical to I-Vectors, but live in a different space for efficiency reasons.
Bignum	Bignums are infinite-precision integers, represented in a format identical to I-Vectors. Each bignum is stored as a series of 8-bit bytes, with the low-order byte stored first. The representation is two's complement, but the sign of the number is redundantly encoded in the subtype field of the bignum: positive bignums are sub-type 0, negative bignums sub-type 1. The access-type code is always 8-Bit.
Long-Float	Long floats are stored as two consecutive words of bits, in the following format:

```
      31      30                20 19                          0
---------------------------------------------------------------
|  Sign (1)  |   Exponent (11)   |        Fraction (20)        |
---------------------------------------------------------------
|                         Fraction (32)                        |
---------------------------------------------------------------
```

The exponent is biased by 1023. Exponents of 0 and 2047 are reserved. The most significant bit of the fraction is stripped off since it is always the complement of the sign bit, and carries no information.

Ratio	Ratios are stored as two consecutive words of Lisp objects, which should both be integers.
Complex	Complex numbers are stored as two consecutive words of Lisp objects, which should both be numbers.
Array	This is actually a header which holds the accessing information and other information about the array. The actual array contents are held in a vector (either an I-Vector or G-Vector) pointed to by an entry in the header. The header is identical in format to a G-Vector. For details on what the array header contains, see section 2.8.3.
String	A vector of bytes. Identical in form to I-Vectors with the access type always 8-Bit. However, instead of accepting and returning fixnums, string accesses accept and return character objects. Only the 8-bit code field is actually stored, and the returned character object always has bit and font values of 0.
Function	A compiled Spice Lisp function consists of both lisp objects and raw bits for the code. The Lisp objects are stored in the Function space in a format identical to that used for general vectors, with a 24-bit length field in the first word. This object contains assorted parameters needed by the calling machinery, a pointer to an 8-bit I-Vector containing the compiled byte codes, a number of pointers to symbols used as special variables within the function, and a number of lisp objects used as constants by the function. For details of the code format and definitions of the byte codes, see section 5.1.

2.6. Forwarding Pointers

GC-Forward	When a data structure is transported into newspace, a GC-Forward pointer is left behind in the first word of the oldspace object. This points to the same type-space in which it is found. For example, a GC-Forward in G-Vector space points to a structure in the G-Vector newspace. GC-Forward pointers are only found in oldspace.

2.7. System and Stack Spaces

The virtual addresses below 08000000_{16} are not occupied by Lisp objects, since Lisp objects with type code 0 are immediate objects. Some of this space is used for other purposes by Lisp. The current allocations are as follows:

Address (base 16)	Use
00000000 - 01FFFFFF	Microcode tables
02000000 - 03FFFFFF	Control Stack
04000000 - 05FFFFFF	Binding Stack
06000000 - 07FFFFFF	System tables

Microcode tables for a given process are never accessed by Lisp-level code from that process (the SAVE function looks at the allocation table of another Lisp process). These tables contain allocation information for the various spaces and pointers to functions that are called when escapes to macrocode are done. There are currently two microcode tables:

Address (base 16)	Use
00010000 - 00010100	Allocation table
00020000 - 00020100	Escape function table

The format of the allocation table is described in chapter 4, and the format of the escape function table is described in section 6.3.

The control stack grows upward (toward higher addresses) in memory, and is a framed stack. It contains only general Lisp objects (with some random things encoded as fixnums or Misc codes). Every object pointed to by an entry on this stack is kept alive. The frame for a function call contains an area for the function's arguments, an area for local variables, a pointer to the caller's frame, and a pointer into the binding stack. The frame for a Catch form contains similar information. The precise stack format can be found in chapter 3.

The special binding stack also grows upward. This stack is used to hold previous values of special variables that have been bound. It grows and shrinks with the depth of the binding environment, as reflected in the control stack. This stack contains symbol-value pairs, with only boxed Lisp objects present.

System table space is used to interface Lisp to the operating system. This is the only part of the address space that contains invalid memory, so all Accent messages received will appear in this space. Since files are sent and received in messages, all files will be mapped into this space. Data in system table space may be accessed and altered by the instructions described in section 5.2.11.

There are significant performance advantages to be gained by aligning all objects on the PERQ's ``quad-word'' (64-bit) boundaries. This happens automatically for conses, long-floats, complex numbers, and ratios, which are all two Lisp-words long. For all other pointer-type objects, the allocator makes sure that the object starts on a quad-word boundary, wasting a word with a Misc-Trap code if necessary. Thus, every pointer (except possibly for stack and system area pointers) will have its two low-order bits 0. User-level code should never have to notice this distinction.

2.8. Vectors and Arrays

Common Lisp arrays can be represented in a few different ways in Spice Lisp -- different representations have different performance advantages. Simple general vectors, simple vectors of integers, and simple strings are basic Spice Lisp data types, and access to these structures is quicker than access to non-simple (or ``complex'') arrays. However, all multi-dimensional arrays in Spice Lisp are complex arrays, so references to these is always through a header structure.

2.8.1. General Vectors

G-Vectors contain Lisp objects. The format is as follows:

```
------------------------------------------------------------------
|  Fixnum code (4) | Subtype (4) |   Allocated length (24)       |
------------------------------------------------------------------
|  Vector entry 0   (Additional entries in subsequent words)     |
------------------------------------------------------------------
```

Note that the subtype field overlaps the type field -- this means that the subtype can change the sign bit of the fixnum. The first word of the vector is a header indicating its length; the remaining words hold the boxed entries of the vector, one entry per 32-bit word. The header word is of type fixnum. It contains a 3-bit subtype field, which is used to indicate several special types of general vectors. At present, the following subtype codes are defined:

0	Normal. Used for assorted things.
1	Named structure created by DEFSTRUCT, with type name in entry 0.
2	EQ Hash Table, last rehashed in dynamic-0 space.
3	EQ Hash Table, last rehashed in dynamic-1 space.
4	EQ Hash Table, must be rehashed.

Following the subtype is a 24-bit field indicating how many 32-bit words are allocated for this vector, including the header word. Legal indices into the vector range from zero to the number in the allocated length field minus 2, inclusive. The index is checked on every access to the vector. Entry 0 is stored in the second word of the vector, and subsequent entries follow contiguously in virtual memory.

Once a vector has been allocated, it is possible to reduce its length by using the Shrink-Vector instruction, but never to increase its length, even back to the original size, since the space freed by the reduction may have been reclaimed. This reduction simply stores a new smaller value in the length field of the header word.

It is not an error to create a vector of length 0, though it will always be an out-of-bounds error to access such an object. The maximum possible length for a general vector is $2^{24}-2$ entries, and that is only possible if no other general vectors are present in the space.

Objects of type Function and Array are identical in format to general vectors, though they have their own spaces. In both cases, only 0 is currently used in the sub-type field of the header word.

2.8.2. Integer Vectors

I-Vectors contain unboxed items of data, and their format is more complex. The data items come in a variety of lengths, but are of constant length within a given vector. Data going to and from an I-Vector are passed as Fixnums, right justified. Internally these integers are stored in packed form, filling 32-bit words without any type-codes or other overhead. The format is as follows:

```
----------------------------------------------------------------
| Fixnum code (4) | Subtype (4) |  Allocated length (24)       |
----------------------------------------------------------------
| Access type (4) | Number of entries (28)                     |
----------------------------------------------------------------
|                                     Entry 0 right justified |
----------------------------------------------------------------
```

Note that the subtype field overlaps the type field -- this means that the subtype can change the sign bit of the fixnum. The first word of an I-Vector contains the Fixnum type-code in the top 4 bits, a 4-bit subtype code in the next four bits, and the total allocated length of the vector (in 32-bit words) in the low-order 24 bits. At present, the following subtype codes are defined:

0	Normal. Used for assorted things.
1	Code. This is the code-vector for a function object.

The second word of the vector is the one that is looked at every time the vector is accessed. The low-order 28 bits of this word contain the number of valid entries in the vector, regardless of how long each entry is. The lowest legal index into the vector is always 0; the highest legal index is one less than this number-of-entries field from the second word. These bounds are checked on every access. Once a vector is allocated, it can be reduced in size but not increased. The Shrink-Vector instruction changes both the allocated length field and the number-of-entries field of an integer vector.

The high-order 4 bits of the second word contain an access-type code which indicates how many bits are occupied by each item (and therefore how many items are packed into a 32-bit word). The encoding is as follows:

0	1-Bit	8	Unused
1	2-Bit	9	Unused
2	4-Bit	10	Unused
3	8-Bit	11	Unused
4	16-Bit	12	Unused
5	Unused	13	Unused
6	Unused	14	Unused
7	Unused	15	Unused

In I-Vectors, the data items are packed into the third and subsequent words of the vector. Item 0 is right justified in the third word, item 1 is to its left, and so on until the allocated number of items has been accommodated. All of the currently-defined access types happen to pack neatly into 32-bit words, but if this should not be the case, some unused bits would remain at the left side of each word. No attempt will be made to split items between words to use up these odd bits. When allocated, an I-Vector is initialized to all 0's.

As with G-Vectors, it is not an error to create an I-Vector of length 0, but it will always be an error to access such a vector. The maximum possible length of an I-Vector is $2^{28}-1$ entries or $2^{24}-3$ words, whichever is smaller.

Objects of type String are identical in format to I-Vectors, though they have their own space. Strings always have subtype 0 and access-type 3 (8-Bit). Strings differ from normal I-Vectors in that the accessing instructions accept and return objects of type Character rather than Fixnum.

Bignums are also identical in format and operation to I-Vectors, though they may also be operated on directly by microcoded routines. For details of the currently-defined sub-types and their access-codes, see section 2.5.

2.8.3. Arrays

An array header is identical in form to a G-Vector. Like any G-Vector, its first word contains a fixnum type-code, a 4-bit subtype code, and a 24-bit total length field (this is the length of the array header, not of the vector that holds the data). At present, the subtype code is always 0. The entries in the header-vector are interpreted as follows:

0 Data Vector — This is a pointer to the I-Vector, G-Vector, or string that contains the actual data of the array. In a multi-dimensional array, the supplied indices are converted into a single 1-D index which is used to access the data vector in the usual way.

1 Number of Elements — This is a fixnum indicating the number of elements for which there is space in the data vector.

2 Fill Pointer	This is a fixnum indicating how many elements of the data vector are actually considered to be in use. Normally this is initialized to the same value as the Number of Elements field, but in some array applications it will be given a smaller value. Any access beyond the fill pointer is illegal.
3 Displacement	This fixnum value is added to the final code-vector index after the index arithmetic is done but before the access occurs. Used for mapping a portion of one array into another. For most arrays, this is 0.
4 Range of First Index	This is the number of index values along the first dimension, or one greater than the largest legal value of this index (since the arrays are always zero-based). A fixnum in the range 0 to $2^{24}-1$. If any of the indices has a range of 0, the array is legal but will contain no data and accesses to it will always be out of range. In a 0-dimension array, this entry will not be present.
5 - N Ranges of Subsequent Dimensions	

The number of dimensions of an array can be determined by looking at the length of the array header. The rank will be this number minus 6. The maximum array rank is 65535 - 6, or 65529.

The ranges of all indices are checked on every access, during the conversion to a single data-vector index. In this conversion, each index is added to the accumulating total, then the total is multiplied by the range of the following dimension, the next index is added in, and so on. In other words, if the data vector is scanned linearly, the last array index is the one that varies most rapidly, then the index before it, and so on.

2.9. Symbols Known to the Microcode

A large number of symbols will be pre-defined when a Spice Lisp system is fired up. A few of these are so fundamental to the operation of the system that their addresses have to be assembled into the microcode. These symbols are listed here. All of these symbols are in static space, so they will not be moving around.

NIL	$5C000000_{16}$	The value of NIL is always NIL; it is an error to alter it. NIL is unique among symbols in that you can take its CAR and CDR, yielding NIL in either case.
T	$5C00000C_{16}$	The value of T is always T; it is an error to alter it.
%SP-Internal-Apply	$5C000018_{16}$	The function stored in the definition cell of this

symbol is called by the microcode whenever compiled code calls an interpreted function. See section 6.1.2 for details.

%SP-Internal-Error
$5C000024_{16}$ The function stored in the definition cell of this symbol is called whenever an error is detected during the execution of a byte instruction. See section 6.4 for details.

%SP-Software-Interrupt-Handler
$5C000030_{16}$ The function stored in the definition cell of this symbol is called whenever a software interrupt occurs. See section 6.6 for details.

%SP-Internal-Throw-Tag
$5C00003C_{16}$ This symbol is bound to the tag being thrown when a Catch-All frame is encountered on the stack. See section 6.2 for details.

3. Runtime Environment

3.1. Control Registers

To describe the instructions in chapter 5 and the complicated control conventions in chapter 6 requires that we talk about a number of ``machine registers.'' All of these registers will be treated as if they contain 32-bit Lisp objects.

Control-Stack-Pointer	The stack pointer for the control stack, an object of type Misc-Control-Stack-Pointer. Points to the first unused word in Control-Stack space; this upward growing stack uses a write-increment/decrement-read discipline.
TOS	The top entry of the control stack, which is kept in a register for efficiency. References to local variables are faster if they can assume that the local in question is on the stack in main memory and that it has not popped up into the TOS register. To ensure this, the compiler adds an extra local variable to each function, so that none of the locals that are actually used can ever pop into TOS.
Binding-Stack-Pointer	The stack pointer for the special variable binding stack, an object of type Misc-Binding-Stack-Pointer. The binding stack follows the same write-increment/decrement-read discipline as the control stack.
Active-Frame	An object of type Misc-Control-Stack-Pointer which points to the first word of the call frame for the currently executing function. The virtual address of the start of the arguments and locals area of the active frame is this pointer plus a constant (see section 3.3).
Open-Frame	An object of type Misc-Control-Stack-Pointer which points to the first word of the call frame currently being built (i.e. whose arguments are being evaluated).
Active-Catch	An object of type Misc-Control-Stack-Pointer which points to the first word of the most recent catch frame built.
Active-Function	The compiled function object for the function that is currently being executed. The virtual address of the start of the symbols and constants area of the current function is this pointer plus a constant (see section 3.2).
Active-Code	The I-Vector of instructions for the currently executing function.
PC	A pointer into I-Vector space that indicates the next quadword from which the instruction buffer will be filled. This and the hardware BPC determine the next instruction to be executed.

When a PC is pushed on the stack by a Call or Catch instruction, it is stored in the form of a 16-bit offset from the base of the Active-Code vector and the BPC:

```
 31               27 26        20 19     16 15              0
---------------------------------------------------------------
| Trap type code (5) | Unused (7) | BPC (4) |   Offset  (16)  |
---------------------------------------------------------------
```

3.2. Function Object Format

Each compiled function is represented in the machine as a Function Object. This is identical in form to a G-Vector of lisp objects, and is treated as such by the garbage collector, but it exists in a special function space. (There is no particular reason for this distinction. We may decide later to store these things in G-Vector space, if we become short on spaces or have some reason to believe that this would improve paging behavior.) Usually, the function objects and code vectors will be kept in read-only space, but nothing should depend on this; some applications may create, compile, and destroy functions often enough to make dynamic allocation of function objects worthwhile.

The function object contains a vector of header information needed by the function-calling mechanism: a pointer to the I-Vector that holds the actual code. Following this is the so-called ``symbols and constants'' area. The first few entries in this area are fixnums that give the offsets into the code vector for various numbers of supplied arguments. Following this begin the true symbols and constants used by the function. Any symbol used by the code as a special variable or the name of another function will appear here. Fixnum constants in the range of -256 to +255 can be generated within the byte code, and so do not need to be stored in the constants area as full-word constants.

After the one-word G-Vector header, the entries of the function object are as follows:

```
0  A fixnum with bit fields as follows:
   0  - 14: Number of symbols/constants in this fn object (0 to 32K-1).
            This number includes the optional-arg offsets.
   15 - 26: Not used.
   27:      0 => All args evaled.  1 => This is a FEXPR.
1  Pointer to the unboxed Code vector holding the macro-instructions.
2  A fixnum with bit fields as follows:
   0  -  7: The minimum legal number of args (0 to 255).
   8  - 15: The maximum number of args, not counting &rest (0 to 255).
   16 - 26: Number of local variables allocated on stack (0 to 2047).
   27:      0 => No &rest arg.     1 => One &rest arg.
3  Name of this function (a symbol).
4  Vector of argument names, in order, for debugging use.
5  The symbols and constants area starts here.
   This word is entry 0 of the symbol/constant area.
   The first few entries in this area are fixnums representing
     the code-vector entry points for various numbers of
     optional arguments.  See section 6.1.2.
```

3.3. Control-Stack Format

The Spice Lisp control stack is a framed stack. Call frames, which hold information for function calls, are intermixed with catch frames, which hold information used for non-local exits. In addition, the control stack is used as a scratchpad for random computations.

3.3.1. Call Frames

At any given time, the machine contains pointers to the current top of the control stack, the start of the current active frame (in which the current function is executing), and the start of the most recent open frame. In addition, there is a pointer to the current top of the special binding stack. An open frame is one which has been partially built, but which is still having arguments for it computed. When all the arguments have been computed and saved on the frame, the function is then started. This means that the call frame is completed, becomes the current active frame, and the function is executed. At this time, special variables may be bound and the old values are saved on the binding stack. Upon return, the active frame is popped away and the result is either sent as an argument to some previously opened frame or goes to some other destination. The binding stack is popped and old values are restored.

The active frame contains pointers to the previously-active frame, to the most recent open frame, and to the point to which the binding stack will be popped on exit, among other things. Following this is a vector of storage locations for the function's arguments and local variables. Space is allocated for the maximum number of arguments that the function can take, regardless of how many are actually supplied.

In an open frame, the structure is built up to the point where the arguments are stored. Thus, as arguments are computed, they can simply be pushed on the stack. When the function is finally started, the remainder of the frame is built. A call frame looks like this:

```
0   Header word.  Type Call-Frame-Header.
1   Function object or EXPR for this call.
2   Pointer to previous active frame.  Type Misc-Control-Stack-Ptr.
3   Pointer to previous open frame.  Type Misc-Control-Stack-Ptr.
4   Pointer to previous binding stack.  Type Misc-Binding-Stack-Ptr.
5   Saved PC of caller.  A fixnum.
6   Args-and-locals area starts here.  This is entry 0.
```

The first slot is pointed to by the Active-Frame register if this frame is currently active, and by the Open-Frame register if this frame is the currently opened frame.

3.3.2. Catch Frames

Catch frames contain much of the same information that call frames do, and have a very similar format. A catch frame holds the function object for the current function, a stack pointer to the current active and open frames, a pointer to the current top of the binding stack, and a pointer to the previous catch frame. When a Throw occurs, an operation equivalent to returning from this catch frame (as if it were a call frame) is performed, and the stacks are unwound to the proper place for continued execution in the current function. A catch frame looks like this:

0	Header word. Type Catch-Frame-Header.
1	Function object for this call.
2	Pointer to current active frame.
3	Pointer to current open frame.
4	Pointer to current binding stack.
5	Destination PC for a Throw.
6	Tag caught by this catch frame.
7	Pointer to previous catch frame.

The conventions used to manipulate call and catch frames are described in chapter 6.

3.4. Binding-Stack Format

Each entry of the binding-stack consists of two boxed (32-bit) words. Pushed first is a pointer to the symbol being bound. Pushed second is the symbol's old value (any boxed item) that is to be restored when the binding is popped.

4. Storage Management

New objects are allocated from the lowest unused addresses within the specified space. Each allocation call specifies how many words are wanted, and a Free-Storage pointer is incremented by that amount. There is one of these Free-Storage pointers for each space, and it points to the lowest free address in the space. There is also a Clean-Space pointer associated with each space that is used during garbage collection. These pointers are stored in a table in the microcode table area which is indexed by type and space code. The address of the Free-Storage pointer for a given space is

```
        (+ alloc-table-base (lsh type 4) (lsh space 2)).
```

The address of the Clean-Space pointer is

```
        (+ alloc-table-base (lsh type 4) (lsh space 2) 2).
```

PERQ Spice Lisp uses a stop-and-copy garbage collector to reclaim storage. The Collect-Garbage instruction performs a full GC. The algorithm used is a degenerate form of Baker's incremental garbage collection scheme. When the Collect-Garbage instruction is executed, the following happens:

1. The current newspace becomes oldspace, and the current oldspace becomes newspace.

2. The newspace Free-Storage and Clean-Space pointers are initialized to point to the beginning of their spaces.

3. The contents of the ``registers inside the barrier'' are transported. There are only three such registers: Active-Function, Active-Code, and TOS. However, the PC is stored internally as an absolute address, so it must be recomputed if the code vector in Active-Code is transported. This is easily done by subtracting Active-Code from PC before it is transported, and adding it back in afterwards. Because the Active-Code vector will be transported from a quadword boundary to a quadword boundary, the PERQ's internal BPC needn't be modified.

4. The control stack and binding stack are scavenged.

5. Each static pointer space is scavenged.

6. Each new dynamic space is scavenged. The scavenging of the dynamic spaces is done until an entire pass through all of them does not result in anything being transported. At this point, every live object is in newspace.

A Lisp-level GC function must return the oldspace pages to Accent.

4.1. The Transporter

The transporter moves objects from oldspace to newspace. It is given an address A, which contains the object to be transported, B. If B is an immediate object, a pointer into static space, a pointer into read-only space, or a pointer into newspace, the transporter does nothing.

If B is a pointer into oldspace, the object it points to must be moved. It may, however, already have been moved. Fetch the first word of B, and call it C. If C is a GC-forwarding pointer, we form a new pointer with the type code of B and the low 27 bits of C. Write this into A.

If C is not a GC-forwarding pointer, we must copy the object the B points to. Allocate a new object of the same size in newspace, and copy the contents. Replace C with a GC-forwarding pointer to the new structure, and write the address of the new structure back into A.

Hash tables maintained with an EQ relation need special treatment by the transporter. Whenever a G-Vector with subtype 2 or 3 is transported to newspace, its subtype code is changed to 4. The Lisp-level hash-table functions will see that the subtype code has changed, and re-hash the entries before any access is made.

4.2. The Scavenger

The scavenger looks through an area of pointers for pointers into oldspace, transporting the objects they point to into newspace. The stacks and static spaces need to be scavenged once, but the new dynamic spaces need to be scavenged repeatedly, since new objects will be allocated while garbage collection is in progress. To keep track of how much a dynamic space has been scavenged, a Clean-Space pointer is maintained. The Clean-Space pointer points to the next word to be scavenged. Each call to the scavenger scavenges the area between the Clean-Space pointer and the Free-Storage pointer. The Clean-Space pointer is then set to the Free-Storage pointer. When all Clean-Space pointers are equal to their Free-Storage pointers, GC is complete.

To maintain (and create) locality of list structures, list space is treated specially. Two separate Clean-Space pointers are maintained for list space: one for cars and one for cdrs. The scavenger works on the Clean-Cdr pointer unless it is at the Free-Storage pointer, in which case it works on the Clean-Car pointer. When Clean-Car, Clean-Cdr, and the Free-Storage pointer for list space coincide, list space has been completely scavenged.

4.3. Purification

Garbage is created when the files that make up a Spice Lisp system are loaded. Many functions are needed only for initialization and bootstrapping (e.g. the ``one-shot'' functions produced by the compiled for random forms between function definition), and these can be thrown away once a full system is built. Most of the functions in the system, however, will be used after initialization. Rather than bend over backwards to make the compiler dump some functions in read-only space and others in dynamic space (which involves dumping their constants in the proper spaces, also), we will dump everything into dynamic space, and use the following storage allocation feature to move what we need after initialization into read-only and static space.

The Set-Newspace-For-Type instruction lets us set the free pointer of the next newspace to dynamic or read-only space instead of the corresponding dynamic space. When the next GC happens, objects in newspace will be transported to this other space (static or read-only) instead of dynamic space. Care must be taken, of course, to ensure that objects in read-only space point only to static or read-only space. Probably the following should be used for ``purifying'' a system:

```
(set-newspace-for-type  1  2)   ; bit-vectors to static
(set-newspace-for-type  2  2)   ; likewise for i-vectors
(set-newspace-for-type  3  2)   ; and strings
(set-newspace-for-type  4  2)   ; and bignums
(set-newspace-for-type  5  2)   ; and long-floats
(set-newspace-for-type  6  3)   ; complexes can be read-only
(set-newspace-for-type  7  3)   ; as can ratios
(set-newspace-for-type  8  2)   ; g-vectors should be static
(set-newspace-for-type  9  3)   ; functions should be read-only
(set-newspace-for-type 10  3)   ; arrays can be, also
(set-newspace-for-type 11  2)   ; symbols should be static
(set-newspace-for-type 12  2)   ; as should conses.
```

5. Macro Instruction Set

The intent is that this instruction set should be a very direct mapping from the S-expression source it is derived from. There should therefore never be any temptation for users to write macrocode by hand; all of the system that is not in microcode should be written in Lisp. Since the compiler will run both in Spice Lisp and in Maclisp, we need not hand-code things even for bootstrapping.

5.1. Macro-Instruction Formats

There are three ways in which instructions fetch their arguments and store their results.

1. Most instructions pop all of their operands off of the stack and push a result back onto the stack, behaving like little Lisp functions. There are some instructions that will take their last operand from a place other than the stack (an immediate constant, a local variable, etc).

2. Some instructions modify a value in place. This value is sometimes the top of the stack, but could be a local variable, argument, or special variable. In the descriptions of the instructions below, these instructions operate on a pseudo-operand E, the effective address, which is specified as part of the opcode.

3. Finally, a few instructions pop the top of the stack but leave no result. The Pop, Branch, and Dispatch instructions do this.

All non-branching Spice Lisp instructions are made up of 1 or 2 opcode bytes, that contain an implicit effective address, and 0 to 2 bytes following the opcode that are used as part of the effective address. Branching instructions have 1 or 2 bytes of opcode followed by a 1 or 2 byte branch offset. The possible effective addresses (and their use of additional effective address bytes) are these:

Stack
: The operand is taken from the stack. Then the operation takes place, in some cases pushing a result onto the stack. No effective address bytes are fetched. The names of instructions that take all stack operands are not suffixed with an effective address specifier, as others are. These instructions are called ``basic'' instructions. In most cases, the compiler-writer need concern himself with only these forms of an instruction. The peephole optimizer will replace sequences of stack referencing instructions with instructions with differerent addressing modes if the resulting sequence is faster.

Positive Short Integer Constant
: A byte is fetched and is converted to a positive fixnum in the range 0 to 255. This is used as the operand. The ``-PSIC'' suffix on an instruction name is used for instructions with positive short integer constant operands. Some instructions imply a particular short integer without a second byte. These

are suffixed with ``-PSICn'' where n is the short integer. A short integer constant may never be used as a result effective address, of course.

Negative Short Integer Constant

A byte is fetched and is converted to a negative fixnum in the range -1 to -256. This is used as the operand. The ``-NSIC'' suffix on an instruction name is used for instructions with negative short integer constant operands.

Arguments & Locals

In most cases, one byte is fetched and used as an unsigned offset (0 - 255) into the arguments and local variables area of the currently active call frame (``-AL'' suffix). The contents of this cell are used as the operand. For a few instructions, two bytes are fetched to form a 16-bit offset (0 - 65535). In fetching this double offset, the low-order byte comes in first (``-LongAL'' suffix). Some instructions imply a particular offset without the need for another offset byte. These instructions are those that are suffixed with ``-ALn'' where n is an integer which denotes the implied offset. When used as a result effective address, the result is stored in the specified slot of the call frame.

Constants

In most cases, one byte is fetched and used as an unsigned offset (0 - 255) into the vector of symbols and constants in the code object of the current function. The constant in this cell is used directly (``-C'' suffix). For a few instructions, the next two bytes are fetched to form a 16-bit unsigned offset (0 - 65535) (``-LongC'' suffix). In fetching this double offset, the low-order byte comes in first. Sometimes an instruction implies an offset into the symbols and constants vector without the need of another byte for the offset. In these instances when the offset is implied, the instruction will have the suffix ``-Cn'' where n is an integer denoting the offset. Constants may not be used as a result effective address.

Symbols

In most cases, one byte is fetched and used as an unsigned offset (0 - 255) into the vector of symbols and constants in the code object of the current function. The constant in this cell is supposed to be a symbol pointer, and the operand is fetched from its value cell (``-S'' suffix). If the value is Misc-Trap, an unbound variable error is signalled. For some instructions, the next two bytes are fetched to form a 16-bit offset (``-LongS'' suffix). In fetching this double offset, the low-order byte comes in first. Sometimes an instruction implies an offset into the symbols and constants vector without the need of another byte for the offset. In these instances when the offset is implied, the instruction will have the suffix ``-Sn'' where n is an integer denoting the offset. When a symbol is used as a result effective address, its value cell

	is set to the result.
Ignore	Specified with a ``-Ignore'' suffix. This may be used only as a result effective address.

In the following listing, the effective address is called ``E'' and its contents are called ``CE''. The opcodes for these instructions are defined in a file read by the microassembler, compiler, error system, and disassembler. This file lives on CMU-CS-C as PRVA:<Slisp.Compiler.New-And-Improved>Instrdefs.Slisp and CMU-Badger as >Slisp>Instrdefs.Lisp. It is included in this document as an appendix.

5.2. Instructions

There are 11 classes of instructions: allocation, stack manipulation, list manipulation, symbol manipulation, array manipulation, type predicate, arithmetic and logical, branching and dispatching, function call and return, miscellaneous, and system hacking. A number of the instructions below combine the effect of two or more simpler instructions, and are part of the instruction set for efficiency reasons. It is envisioned that the compiler will generate code using the stack forms of most instructions, with lots of Push and Pop instructions to get operands and store results. An optimizing assembler will then collapse sequences of these simple instructions into the faster, more compact complex instructions. Each basic instruction is flagged with an asterisk (*).

5.2.1. Allocation

All non-immediate objects are allocated in the ``current allocation space,'' which is dynamic space, static space, or read-only space. The current allocation space is initially dynamic space, but can be changed by using the Set-Allocation-Space instruction below. The current allocation space can be determined by using the Get-Allocation-Space instruction. One usually wants to change the allocation space around some section of code; an unwind protect should be used to insure that the allocation space is restored to some safe value.

Get-Allocation-Space () pushes 0, 2, or 3 if the current allocation space is dynamic, static, or read-only respectively.

```
     Get-Allocation-Space*
```

Set-Allocation-Space (X) sets the current allocation space to dynamic, static, or read-only if X is 0, 2, or 3 respectively. Pushes X.

```
     Set-Allocation-Space*
```

Alloc-Bit-Vector (Length) pushes a new bit-vector Length bits long, which is allocated in the current allocation space. Length must be a positive fixnum.

```
     Alloc-Bit-Vector*
```

Alloc-I-Vector (Length A) pushes a new I-Vector Length bytes long, with the access code specified by A. Length and A must be positive fixnums.

Alloc-I-Vector*

Alloc-String (Length) pushes a new string Length characters long. Length must be a fixnum.

Alloc-String*

Alloc-Bignum (Length) pushes a new bignum Length 8-bit bytes long. Length must be a fixnum.

Alloc-Bignum*

Make-Complex (Realpart Imagpart) pushes a new complex number with the specified Realpart and Imagpart. Realpart and Imagpart should be the same type of non-complex number.

Make-Complex*

Make-Ratio (Numerator Denominator) pushes a new ratio with the specified Numerator and Denominator. Numerator and Denominator should be integers.

Make-Ratio*

Alloc-G-Vector (Length Initial-Element) pushes a new G-Vector with Length elements initialized to Initial-Element. Length should be a fixnum.

Alloc-G-Vector*

Vector (Elt$_0$ Elt$_1$ ... Elt$_{\text{Length - 1}}$ Length) pushes a new G-Vector containing the specified Length elements. Length should be a fixnum.

Vector*
Vector-PSIC

Alloc-Function (Length) pushes a new function with Length elements. Length should be a fixnum.

Alloc-Function*

Alloc-Array (Length) pushes a new array with Length elements. Length should be a fixnum.

Alloc-Array*

Alloc-Symbol (Print-Name) pushes a new symbol with the print-name as Print-Name. The value is initially Misc-Trap, the definition is Misc-Trap, the property list and the package are initially NIL. The symbol is not interned by this operation -- that is done in macrocode. Print-Name should be a simple-string.

Alloc-Symbol*

Cons (Car Cdr) pushes a new cons with the specified Car and Cdr.

Cons*

Set-LPush (Car E) pushes a new cons with the specified Car and CE as the cdr, and stores the cons back into E.

Set-LPush-AL
Set-LPush-S

List (Elt$_0$ Elt$_1$ ... Elt$_{CE - 1}$ Length) pushes a new list containing the Length elements. Length should be fixnum.

List*
List-PSIC

List* (Elt$_0$ Elt$_1$ ... Elt$_{CE - 1}$ Length) pushes a list* formed by the CE elements onto the stack. Length should be a fixnum.

List**
List*-PSIC

5.2.2. Stack Manipulation

Push (E) pushes CE onto the stack.

Push-PSIC*
Push-PSIC0
Push-PSIC1
Push-PSIC2
Push-PSIC3
Psuh-NSIC*
Push-AL*
Push-AL0
Push-AL1
Push-AL2
Push-AL3
Push-AL4
Push-AL5
Push-AL6

Push-AL7
Push-LongAL*
Push-C*
Push-LongC*
Push-S*
Push-LongS*

Pop (E) pops the stack into E.

Pop-AL*
Pop-AL0
Pop-AL1
Pop-AL2
Pop-AL3
Pop-AL4
Pop-AL5
Pop-AL6
Pop-AL7
Pop-LongAL*
Pop-S*
Pop-LongS*
Pop-Ignore*

Exchange () exchanges the top two elements of the stack.

Exchange*

Copy (E) copies the top of stack to E.

Copy*
Copy-AL

NPop (N). If N is positive, N items are popped off of the stack. If N is negative, NIL is pushed onto the stack -N times. N must be a fixnum.

NPop-Stack*
NPop-PSIC
NPop-NSIC

Bind-Null (E) pushes CE (which must be a symbol) and its current value onto the binding stack, and sets the value cell of CE to NIL.

```
              *
     Bind-Null
     Bind-Null-C
```

Bind (Value Symbol) pushes Symbol (which must be a symbol) and its current value onto the binding stack, and sets the value cell of Symbol to Value.

```
         *
     Bind
     Bind-C
```

Unbind (N) undoes the top N bindings on the binding stack.

```
           *
     Unbind
     Unbind-PSIC
```

5.2.3. List Manipulation

Cxxr (L). The cxxr of L is pushed onto the stack. L should be a list or NIL.

```
        *
     Car
     Car-AL
        *
     Cdr
     Cdr-AL
         *
     Cadr
     Cadr-AL
         *
     Cddr
     Cddr-AL
         *
     Cdar
     Cdar-AL
         *
     Caar
     Caar-AL
```

Set-Cxxr (E). The cxxr of CE is stored in E. CE should be either a list or NIL.

```
     Set-Cdr-AL
     Set-Cdr-S
     Set-Cddr-AL
     Set-Cddr-S
```

Set-Lpop (E). The car of CE is pushed onto the stack; the cdr of CE is stored in E. CE should be a list or NIL.

```
Set-Lpop-AL
Set-Lpop-S
```

Spread (E) pushes the elements of the list CE onto the stack in left-to-right order.

```
      *
Spread
Spread-AL
```

Replace-Cxr (List Value) sets the cxr of the List to Value and pushes Value on the stack.

```
           *
Replace-Car
Replace-Car-AL
           *
Replace-Cdr
Replace-Cdr-Al
```

Endp (X) pushes T if X is NIL, or NIL if X is a cons. Otherwise an error is signalled.

```
    *
Endp
Endp-AL
```

Assoc (List Item) pushes the first cons in the association-list List whose car is EQL to Item. If the = part of the EQL comparison bugs out (and it can if the numbers are too complicated), a Lisp-level Assoc function is called with the current cdr of the List. Assq pushes the first cons in the association-list List whose car is EQ to Item.

```
     *
Assoc
    *
Assq
```

Member (List Item) pushes the first cons in the list List whose car is EQL to Item. If the = part of the EQL comparison bugs out, a Lisp-level Member function is called with the current cdr of the List. Memq pushes the first cons in List whose car is EQ to the Item.

```
      *
Member
    *
Memq
```

GetF (List Indicator Default) searches for the Indicator in the list List, cddring down as the Common Lisp form GetF would. If Indicator is found, its associated value is pushed, otherwise Default is pushed.

GetF*

5.2.4. Symbol Manipulation

Get-Value (Symbol) pushes the value of Symbol (which must be a symbol) onto the stack. An error is signalled if CE is unbound.

Get-Value*

Set-Value (Symbol Value) sets the value cell of the symbol Symbol to Value. Value is left on the top of the stack.

Set-Value*

Get-Definition (Symbol) pushes the definition of the symbol Symbol onto the stack. If Symbol is undefined, an error is signalled.

Get-Definition*
Get-Definition-C

Set-Definition (Symbol Definition) sets the definition of the symbol Symbol to Definition. Definition is left on the top of the stack.

Set-Definition*
Set-Definition-C

Get-Plist (Symbol) pushes the property list of the symbol Symbol onto the stack.

Get-Plist*
Get-Plist-C

Set-Plist (Symbol Plist) sets the property list of the symbol Symbol to Plist. Plist is left on the top of the stack.

Set-Plist*
Set-Plist-C

Get-Pname (Symbol) pushes the print name of the symbol Symbol onto the stack.

Get-Pname*

Get-Package (Symbol) pushes the package cell of the symbol Symbol onto the stack.

Get-Package*

Set-Package (Symbol Package) sets the package cell of the symbol Symbol to Package. Package is left on the top of the stack.

Set-Package*

Boundp (Symbol) pushes T if the symbol Symbol is bound; NIL otherwise.

Boundp*
Boundp-C

FBoundp (Symbol) pushes T if the symbol Symbol is defined; NIL otherwise.

FBoundp*
FBoundp-C

5.2.5. Array Manipulation

Common Lisp arrays have many manifestations in Spice Lisp. The Spice Lisp data types Bit-Vector, Integer-Vector, String, General-Vector, and Array are used to implement the collection of data types the Common Lisp manual calls ``arrays.''

In the following instruction descriptions, ``simple-array'' means an array implemented in Spice Lisp as a Bit-Vector, I-Vector, String, or G-Vector. ``Complex-array'' means an array implemented as a Spice Lisp Array object. ``Complex-bit-vector'' means a bit-vector implemented as a Spice Lisp array; similar remarks apply for ``complex-string'' and so forth.

Vector-Length (Vector) pushes the length of the one-dimensional Common Lisp array Vector. G-Vector-Length, Simple-String-Length, and Simple-Bit-Vector-Length push the lengths of G-Vectors, Spice Lisp strings, and Spice Lisp Bit-Vectors respectively. Vector should be a vector of the appropriate type.

Vector-Length*

G-Vector-Length*

Simple-String-Length*

Simple-Bit-Vector-Length*

Get-Vector-Subtype (Vector) pushes the subtype field of the vector Vector as an integer. Vector should be a vector of some sort.

Get-Vector-Subtype*

Set-Vector-Subtype (Vector A) sets the subtype field of the vector Vector to A, which must be an fixnum.

Set-Vector-Subtype*

Get-Vector-Access-Code (Vector) pushes the access code of the I-Vector (or Bit-Vector) Vector as an integer.

Get-Vector-Access-Code*

Shrink-Vector (Vector Length) sets the length field and the number-of-entries field of the vector Vector to Length. If the vector contains Lisp objects, entries beyond the new end are set to Misc-Trap. Pushes the shortened vector. Length should be a fixnum. One cannot shrink array headers or function headers.

Shrink-Vector*

Typed-Vref (A Vector I) pushes the I'th element of the I-Vector Vector by indexing into it as if its access-code were A. A and I should be fixnums.

Typed-Vref*

Typed-Vset (A Vector I Value) sets the I'th element of the I-Vector Vector to Value indexing into Vector as if its access-code were A. A, I, and Value should be fixnums. Value is pushed onto the stack.

Typed-Vset*

Header-Length (Object) pushes the number of Lisp objects in the header of the function or array Object. This is used to find the number of dimensions of an array or the number of constants in a function.

Header-Length*

Header-Ref (Object I) pushes the I'th element of the function or array header Object. I must be a fixnum.

Header-Ref*

Header-Set (Object I Value) sets the I'th element of the function of array header Object to Value, and pushes Value. I must be a fixnum.

Header-Set*

The names of the instructions used to reference and set elements of arrays are somewhat based on the Common Lisp function names. The SVref, SBit, and SChar instructions perform the same operation as their Common Lisp namesakes -- referencing elements of simple-vectors, simple-bit-vectors, and simple-strings respectively. Aref1 references any kind of one dimensional array. The names of setting functions are derived by replacing ``ref'' with ``set'', ``char'' with ``charset'', and ``bit'' with ``bitset.''

Aref1 (Array I) pushes the I'th element of the one-dimensional array Array.

SVref pushes an element of a G-Vector; SChar an element of a string; Sbit an element of a Bit-Vector. I should be a fixnum.

```
          *
     Aref1
     Aref1-AL
          *
     SVref
     SVref-PSIC
     SVref-AL
     SVref-PSIC0
     SVref-PSIC1
     SVref-PSIC2
     SVref-PSIC3
     SVref-PSIC4
     SVref-PSIC5
          *
     SChar
     SChar-AL
         *
     SBit
```

Aset1 (Array I Value) sets the I'th element of the one-dimensional array Array to Value. SVset sets an element of a G-Vector; SCharset an element of a string; SBitset an element of a Bit-Vector. I should be a fixnum and Value is pushed on the stack.

```
          *
     Aset1
     Aset1-AL
          *
     SVset
     SVset-AL
             *
     SCharset
     SCharset-AL
            *
     SBitset
```

SVset* (Array Value I) sets the I'th element of the G-Vector Array to Value. The operands to the instruction are arranged so that the index can be specified as part of the effective address. This could not be done in general, of course, because order of evaluation must be preserved, but for constant indices (as used in structure accesses) this problem does not come up.

```
     SVset*-PSIC
     SVset*-PSIC0
     SVset*-PSIC1
     SVset*-PSIC2
     SVset*-PSIC3
     SVset*-PSIC4
     SVset*-PSIC5
```

CAref2 (Array I1 I2) pushes the element (I1, I2) of the two-dimensional array Array onto the stack. I1 and I2 should be fixnums. CAref3 pushes the element (I1, I2, I3).

    CAref2
    CAref3

CAset2 (Array I1 I2 Value) sets the element (I1, I2) of the two-dimensional array Array to Value and pushes Value on the stack. I1 and I2 should be fixnums. CAset3 sets the element (I1, I2, I3).

    CAset2
    CAset3

Bit-Bash (V1 V2 V3 Op). V1, V2, and V3 should be bit-vectors and Op should be a fixnum. The elements of the bit vector V3 are filled with the result of Op'ing the corresponding elements of V1 and V2. Op should be a Boole-style number (see the Boole instruction in section 5.2.7).

    Bit-Bash*

The rest of the instructions in this section implement special cases of sequence or string operations. Where an operand is referred to as a string, it may actually be an 8-bit I-Vector or system area pointer.

Byte-BLT (Src-String Src-Start Dst-String Dst-Start Dst-End) moves bytes from Src-String into Dst-String between Dst-Start (inclusive) and Dst-End (exclusive). Dst-Start - Dst-End bytes are moved. If the substrings specified overlap, ``the right thing happens,'' i.e. all the characters are moved to the right place. This instruction corresponds to the Common Lisp function REPLACE when the sequences are simple-strings.

    Byte-BLT*

Find-Character (String Start End Character) searches String for the Character from Start to End. If the character is found, the corresponding index into String is returned, otherwise NIL is returned. This instruction corresponds to the Common Lisp function FIND when the sequence is a simple-string.

    Find-Character*

Find-Character-With-Attribute (String Start End Table Mask) The codes of the characters of String from Start to End are used as indices into the Table, which is an I-Vector of 8-bit bytes. When the number picked up from the table bitwise ANDed with Mask is non-zero, the current index into the String is returned.

    Find-Character-With-Attribute*

SXHash-Simple-String (String Length) Computes the hash code of the first Length characters of String and pushes it on the stack. This corresponds to the Common Lisp function SXHASH when the object is a simple-string. The Length operand can be Nil, in which case the length of the string is calculated in microcode.

SXHash-Simple-String*

5.2.6. Type Predicates

Bit-Vector-P (Object) pushes T if Object is a Common Lisp bit-vector or NIL if it is not.

Bit-Vector-P*

Simple-Bit-Vector-P (Object) pushes T if Object is a Spice Lisp bit-vector or NIL if it is not.

Simple-Bit-Vector-P*

Simple-Integer-Vector-P (Object) pushes T if Object is a Spice Lisp I-Vector or NIL if it is not.

Simple-Integer-Vector-P*

StringP (Object) pushes T if Object is a Common Lisp string or NIL if it is not.

StringP*

Simple-String-P (Object) pushes T if Object is a Spice Lisp string or NIL if it is not.

Simple-String-P*

BignumP (Object) pushes T if Object is a bignum or NIL if it is not.

BignumP*

Long-Float-P (Object) pushes T if Object is a long-float or NIL if it is not.

Long-Float-P*

ComplexP (Object) pushes T if Object is a complex number or NIL if it is not.

ComplexP*

RatioP (Object) pushes T if Object is a ratio or NIL if it is not.

RatioP*

IntegerP (Object) pushes T if Object is a fixnum or bignum or NIL if it is not.

IntegerP*

RationalP (Object) pushes T if Object is a fixnum, bignum, or ratio.

RationalP*

FloatP (Object) pushes T if Object is a short-float or long-float.

FloatP*

NumberP (Object) pushes T if Object is a number or NIL if it is not.

NumberP*

General-Vector-P (Object) pushes T if Object is a Common Lisp general vector or NIL if it is not.

General-Vector-P*

Simple-Vector-P (Object) pushes T if Object is a Spice Lisp G-Vector or NIL if it is not.

Simple-Vector-P*

Compiled-Function-P (Object) pushes T if Object is a compiled function or NIL if it is not.

Compiled-Function-P*

ArrayP (Object) pushes T if Object is a Common Lisp array or NIL if it is not.

ArrayP*

VectorP (Object) pushes T if Object is a Common Lisp vector of NIL if it is not.

VectorP*

Complex-Array-P (Object) pushes T if Object is a Spice Lisp array or NIL if it is not.

Complex-Array-P*

SymbolP (Object) pushes T if Object is a symbol or NIL if it is not.

SymbolP*

ListP (Object) pushes T if Object is a cons or NIL or NIL if it is not.

ListP*

ConsP (Object) pushes T if Object is a cons or NIL if it is not.

ConsP*

FixnumP (Object) pushes T if Object is a fixnum or NIL if it is not.

FixnumP*

Short-Float-P (Object) pushes T if Object is a short-float or NIL if it is not.

Short-Float-P*

CharacterP (Object) pushes T if Object is a character or NIL if it is not.

CharacterP*

5.2.7. Arithmetic and Logic

Integer-Length (Object) pushes the integer-length (as defined in the Common Lisp manual) of the integer Object onto the stack.

Integer-Length*
Integer-Length-AL

Float-Short (Object) pushes a short-float corresponding to the number Object.

Float-Short*

Float-Long (Number) pushes a long float formed by coercing Number to a long float. This corresponds to the Common Lisp function Float when given a long float as its second argument.

Float-Long*

Realpart (Number) pushes the realpart of the Number.

Realpart*

Imagpart (Number) pushes the imagpart of the Number.

Imagpart*

Numerator (Number) pushes the numerator of the rational Number.

Numerator*

Denominator (Number) pushes the denominator of the rational Number.

Denominator*

Decode-Float (Number) performs the Common Lisp Decode-Float function, leaving 3 values and a Values-Marker on the stack.

```
                 *
     Decode-Float
```

Scale-Float (Number X) performs the Common Lisp Scale-Float function, pushing the result on the stack.

```
                *
     Scale-Float
```

= (X Y) pushes T if X and Y are numerically equal, or NIL if they are not. If an integer is compared with a flonum, the integer is floated first; if a short flonum is compared with a long flonum, the short one is first extended. Flonums must be exactly identical (after conversion) for a non-null comparison. < and > are similar.

```
      *
     =
     =-AL
     =-PSIC
      *
     <
     <-AL
     <-PSIC
      *
     >
     >-AL
     >-PSIC
```

Truncate (N X) performs the Common Lisp TRUNCATE operation. There are 3 cases depending on X:

- If X is fixnum 1, push three items: a fixnum or bignum representing the integer part of N (rounded toward 0), then either 0 if N was already an integer or the fractional part of N represented as a flonum or ratio with the same type as N, then Values-Marker 2 to mimic a multiple return of two values.

- If X and N are both fixnums or bignums and X is not 1, divide N by X. Push three items: the integer quotient (a fixnum or bignum), the integer remainder, and Values-Marker 2.

- If either X or N is a flonum or ratio, push a fixnum or bignum quotient (the true quotient rounded toward 0), then a flonum or ratio remainder, then push Values-Marker 2. The type of the remainder is determined by the same type-coercion rules as for +. The value of the remainder is equal to N - X * Quotient.

If Truncate uses the escape-to-macro mechanism (see section 6.3), it builds a multiple-value frame header rather than an escape header.

```
        *
Truncate
Truncate-AL
Truncate-PSIC
```

+ (N X) pushes N + X. -, *, and / are similar.

```
 *
+
+-AL
+-PSIC
+-PSIC1
 *
-
--AL
--PSIC
--PSIC1
 *
*
*-AL
*-PSIC
 *
/
/-AL
/-PSIC
```

1+ (E) stores CE + 1 into E.
```
1+-AL
```

1- (E) stores CE - 1 into E.
```
1--AL
```

Negate (N) pushes -N.

```
      *
Negate
Negate-AL
```

Abs (N) pushes |N|.

```
   *
Abs
Abs-AL
```

Logand (N X) pushes the bitwise and of the integers N and X. Logior and Logxor are analagous.

```
      *
Logand
      *
Logior
```

Logxor*

Lognot (N) pushes the bitwise complement of N.

Lognot*

Boole (Op X Y) performs the Common Lisp Boole operation Op on X, and Y. The Boole constants for Spice Lisp are these:

boole-clr	0
boole-set	1
boole-1	2
boole-2	3
boole-c1	4
boole-c2	5
boole-and	6
boole-ior	7
boole-xor	8
boole-eqv	9
boole-nand	10
boole-nor	11
boole-andc1	12
boole-andc2	13
boole-orc1	14
boole-orc2	15

Boole*

Ash (N X) performs the Common Lisp ASH operation on N and X.

Ash*
Ash-PSIC

Ldb (S P N). All args are integers; S and P are non-negative. Performs the Common Lisp LDB operation with S and P being the size and position of the byte specifier.

Ldb*

Mask-Field (S P N) performs the Common Lisp Mask-Field operation with S and P being the size and position of the byte specifier.

Mask-Field*

Dpb (V S P N) performs the Common Lisp DPB operation with S and P being the size and position of the byte specifier.

Dpb*

Deposit-Field (V S P N) performs the Common Lisp Deposit-Field operation with S and P as the size and position of the byte specifier.

Deposit-Field*

Lsh (N X) pushes a fixnum that is N shifted left by X bits, with 0's shifted in on the right. If X is negative, N is shifted to the right with 0's coming in on the left. Both N and X should be fixnums.

Lsh*
Lsh-PSIC

Logldb (S P N). All args are fixnums. S and P specify a ``byte'' or bit-field of any length within N. This is extracted and is pushed right-justified as a fixnum. S is the length of the field in bits; P is the number of bits from the right of N to the beginning of the specified field. P = 0 means that the field starts at bit 0 of N, and so on. It is an error if the specified field is not entirely within the 28 bits of N

Logldb*

Logdpb (V S P N). All args are fixnums. Pushes a number equal to N, but with the field specified by P and S replaced by the S low-order bits of V. It is an error if the field does not fit into the 28 bits of N.

Logdpb*

5.2.8. Branching and Dispatching

Branch instructions add or subtract a 1 or 2 byte a relative offset to the PC after the branch instruction and the offset bytes have been fetched. The opcode determines the direction of the branch and the number of offset bytes to be fetched.

Branch-Forward (Offset) adds the 1 byte Offset to the PC. Long-Branch-Forward adds a 2 byte Offset. Branch-Backward and Long-Branch-Backward subtract 1 or 2 byte Offsets.

Branch-Forward*

Long-Branch-Forward*

Branch-Backward*

Long-Branch-Backward*

Branch-Null (Offset) pops the top item off the stack and branches if it is NIL; Branch-Not-Null branches if it is not null.

Branch-Null-Forward*

Long-Branch-Null-Forward*

Branch-Not-Null-Forward*

Long-Branch-Not-Null-Forward*

Branch-Null-Backward*

Long-Branch-Null-Backward*

Branch-Not-Null-Backward*

Long-Branch-Not-Null-Backward*

Branch-Save-Not-Null (Offset) looks at the value in TOS. If it is Nil, the stack it is popped off the stack and we fall through. Otherwise the stack is left as is and we take the branch.

Branch-Save-Not-Null-Forward*

Long-Branch-Save-Not-Null-Forward*

Branch-Save-Not-Null-Backward*

Long-Branch-Save-Not-Null-Backward*

Dispatch (). The top of stack (TOS) is used as an index into a dispatch table located in the current code vector. The next byte in the instruction is a limit. If TOS is not a fixnum, a fixnum less than 0, or a fixnum greater than or equal to the limit, no branch happens and we fall through, continuing with the next instruction. If TOS is within the specified bounds, however, it is added to a 16-bit number formed by fetching the next 1 or 2 bytes from the instruction stream. This result is used as an index into the code vector, and a 16-bit word is fetched from that memory location. The offset into the current code vector is set to this word. The stack is popped whether or not we branch.

Dispatch*

Long-Dispatch*

5.2.9. Function Call and Return

Call (F). F must be some sort of executable function: a function object, a lambda-expression, or a symbol with one of these stored in its function cell. A call frame for this function is opened. This is explained in more detail in the next chapter.

Call*
Call-C
Call-AL

Call-0 (F). F must be an executable function, as above, but is a function of 0 arguments. Thus, there is no need to collect arguments. The call frame is opened and activated in a single instruction.

Call-0*
Call-0-C
Call-0-AL

Call-Multiple (F). Just like a Call instruction, but it sets bit 21 of the frame header word to indicate that multiple values will be accepted. See section 6.1.4.

Call-Multiple*
Call-Multiple-C
Call-Multiple-AL

Start-Call () closes the currently open call frame, and initiates a function call. See section 6.1.2 for details.

Start-Call*

Push-Last (X) pushes X as an argument, closes the currently open call frame, and initiates a function call. See section 6.1.2 for details.

Push-Last-AL
Push-Last-C
Push-Last-S
Push-Last-PSIC

Return (X). Return from the current function call. After the current frame is popped off the stack, X is pushed as the result being returned. See section 6.1.3 for more details.

Return*
Return-C
Return-AL

Escape-Return (X). If the current call frame has an escape frame header, this works like a normal return, but the value X is put in the destination indicated in the header rather than just being returned on the stack. If the current frame is not an escape frame, just return the single value on the stack as a normal return would.

Escape-Return*

Break-Return (). If the header of the current call frame indicates a break frame, do a Return, but push no return value on the stack. If the current frame is not an escape frame, return NIL.

Break-Return*

Catch () builds a catch frame. The top of stack should hold the tag caught by this catch frame, and the next entry on the stack should be a saved-format PC (which will come from the constants vector of the function). See section 6.2 for details.

Catch*

Catch-Multiple () builds a multiple-value catch frame. The top of stack should hold the tag caught by this catch frame, and the next entry on the stack should be a saved-format PC. See section 6.2 for details.

Catch-Multiple*

Catch-All () builds a catch frame whose tag is the special Catch-All object. The top of stack should hold the saved-format PC, which is the address to branch to if this frame is thrown through. See section 6.2 for details.

Catch-All*

Throw (X). X is the throw-tag, normally a symbol. The value to be returned, either single or multiple, is on the top of the stack. See section 6.2 for a description of how this instruction works.

Throw*
Throw-C

5.2.10. Miscellaneous

Eq (X Y) pushes T if X and Y are the same object, NIL otherwise.

Eq*
Eq-AL
Eq-C

Eql (X Y) pushes T if X and Y are the same object or if X and Y are numbers of the same type with the same value, NIL otherwise.

Eql*
Eql-AL

```
     Eql-C
```

Set-Null (E) sets CE to NIL.

```
             *
     Set-Null
     Set-Null-AL
```

Set-T (E) sets CE to T.

```
          *
     Set-T
     Set-T-AL
```

Set-0 (E) sets CE to 0.

```
          *
     Set-0
     Set-0-AL
```

Make-Predicate (X) pushes NIL if X is NIL or T if it is not.

```
                   *
     Make-Predicate
     Make-Predicate-AL
```

Not-Predicate (X) pushes T if X is NIL or NIL if it is not.

```
                  *
     Not-Predicate
     Not-Predicate-AL
```

Values-To-N (V). V must be a Values-Marker. Returns the number of values indicated in the low 24 bits of V as a fixnum.

```
     Values-To-N
```

N-To-Values (N). N is a fixnum. Returns a Values-Marker with the same low-order 24 bits as N.

```
     N-To-Values
```

Force-Values (). If the top of the stack is a Values-Marker, do nothing; if not, push Values-Marker 1.

```
     Force-Values
```

Flush-Values (). If the top of the stack is a Values-Marker, remove this marker; if not, do nothing.

```
     Flush-Values
```

5.2.11. System Hacking

Get-Type (Object) pushes the five type bits of the Object as a fixnum.

Get-Type*
Get-Type-AL

Get-Space (Object) pushes the two space bits of Object as a fixnum.

Get-Space*

Make-Immediate-Type (X A) pushes an object whose type bits are the integer A and whose other bits come from the immediate object or pointer X. A should be an immediate type code.

Make-Immediate-Type*

8bit-System-Ref (X I). If X is an I-Vector, pushes the I'th byte of X, indexing into X as an 8-bit I-Vector. If X is a system area pointer, pushes the I'th byte beyond X as a fixnum. I must be a fixnum.
8bit-System-Ref

8bit-System-Set (X I V). If X is an I-Vector, sets the I'th element of X to V, indexing into X as an 8-bit I-Vector. If X is a system area pointer, sets the I'th byte beyond X to V.
8bit-System-Set

16bit-System-Ref (X I). If X is an I-Vector, pushes the I'th 16-bit word of X, indexing into X as a 16-bit I-Vector. If X is a system area pointer, pushes the I'th word beyond X as a fixnum. I must be a fixnum.
16bit-System-Ref

16bit-System-Set (X I V). If X is an I-Vector, sets the I'th element of X to V, indexing into X as a 16-bit I-Vector. If X is a system area pointer, sets the I'th word beyond X to V.
16bit-System-Set

Collect-Garbage () causes a stop-and-copy GC to be performed.
Collect-Garbage

Newspace-Bit () pushes 0 if newspace is currently space 0 or 1 if it is 1.
Newspace-Bit

Set-Newspace-For-Type (type space) sets the next newspace free pointer for the type corresponding to the type number to the space corresponding to the space number. There is about one useful thing that you can do with this instruction; see section 4.3. There are a number of not-so-useful but very fun things that you can do with this instruction that are not documented here.
Set-Newspace-For-Type

Kernel-Trap (Argblock Code) is for communication with the Accent Kernel. Code is the type of trap desired (a fixnum), and Argblock is an I-Vector

containing assorted argument information. See section 6.5 for details.
    Kernel-Trap

Halt () stops the execution of Lisp. If continued, T is pushed on the stack.
    Halt

Arg-In-Frame (N F). N is a fixnum, F is a control stack pointer as returned by the Active-Call-Frame and Open-Call-Frame instructions. Pushes the item in slot N of the args-and-locals area of call frame F.
    Arg-In-Frame

Active-Call-Frame () pushes a control-stack pointer to the start of the currently active call frame. This will be of type Misc-Control-Stack-Pointer.
    Active-Call-Frame

Active-Catch-Frame () pushes the control-stack pointer to the start of the currently active catch frame. This is Nil if there is no active catch.
    Active-Catch-Frame

Set-Call-Frame (P). P must be a control stack pointer. This becomes the current active call frame pointer.
    Set-Call-Frame

Current-Open-Frame () pushes a control-stack pointer to the start of the currently open call frame. This will be of type Misc-Control-Stack-Pointer.
    Current-Open-Frame

Set-Open-Frame (P). P must be a control stack pointer. This becomes the current open frame pointer.
    Set-Open-Frame

Current-Stack-Pointer () pushes the Misc-Control-Stack-Pointer that points to the current top of the stack (before the result of this operation is pushed). Note: by definition, this points to the first unused word of the stack, not to the last thing pushed. The stack manipulation instructions make it appear as if the stack is all in contiguous virtual memory, despite the fact that the TOS register will be holding the top word of the stack.
    Current-Stack-Pointer

Current-Binding-Pointer () pushes a Misc-Binding-Stack-Pointer that points to the first word above the current top of the binding stack.
    Current-Binding-Pointer

Read-Control-Stack (F). F must be a control stack pointer. Pushes the Lisp object that resides at this location. If the addressed object is totally outside the current stack, this is an error.
    Read-Control-Stack

Write-Control-Stack (F V). F is a stack pointer, V is any Lisp object. Writes V into the location addressed. If the addressed cell is totally outside the current stack, this is an error. Obviously, this should only be used by carefully written and debugged system code, since you can destroy the world by

using this instruction.
     Write-Control-Stack

Read-Binding-Stack (B). B must be a binding stack pointer. Reads and returns the Lisp object at this location. An error if the location specified is outside the current binding stack.
     Read-Binding-Stack

Write-Binding-Stack (B V). B must be a binding stack pointer. Writes V into the specified location. An error if the location specified is outside the current binding stack.
     Write-Binding-Stack

6. Control Conventions

6.1. Function Calls

6.1.1. Starting a Function Call

The Call and Call-Multiple instructions open a call frame on the control stack, but do not transfer control to the called function. The arguments for the function are then evaluated and pushed on the stack, and the call is started by a Push-Last instruction. Call-Multiple sets bit 21, the multiple-values-accepted bit, of the call frame to indicate that it will accept multiple-values. Call-0 opens the call frame and does the equivalent of a Start-Call instruction (see below) to start the called function. All these instructions take the function to be called as CE.

If CE is a symbol, we fetch the contents of the symbol's definition cell. If it is a Misc-Trap or another symbol, we signal an error. Otherwise, we go on with this definition as the function. We do not allow chains of symbols defined as other symbols. If CE is a compiled function, we perform the following steps:

1. Note the current value of the Control-Stack-Pointer register.

2. Push a Call-Frame-Header on control stack (with bit 21 set if this is a Call-Multiple).

3. Push CE (the function being called).

4. Push the Active-Frame register.

5. Push the Open-Frame register.

6. Push Binding-Stack-Pointer.

7. Push Fixnum -1 or some other easy-to-generate value. This will later be filled with caller's PC.

8. Open-Frame <== Stack frame pointer saved in step 1.

The open frame is now ready to have arguments pushed.

If CE is a list, it is probably a lambda-expression or interpreted lexical closure. The call proceeds as above, with the list stored in the function slot of the new frame. The arguments are pushed normally, and %SP-Internal-Apply will be called when the Push-Last is executed. %SP-Internal-Apply verifies that this function is a lambda or lexical closure.

If CE is anything else, an Illegal-Function error is signalled.

6.1.2. Finishing a Function Call

Push-Last pushes a final argument X and starts the function responsible for the current open frame. Start-Call just starts the function. Call-0 opens the frame and performs the equivalent of a Start-Call immediately, since there are no arguments to push.

We look at the function entry of the current open frame. If this contains a compiled function object, proceed as follows:

1. Insert the current PC (points to the NEXT instruction of the caller's code vector) in the PC slot of the open frame.

2. Active-Function <== Called function (from slot 1 of open frame).

3. Active-Code <== Code vector for new active function.

4. Active-Frame <== Open-Frame

5. Note number of args pushed by caller. Let this be K. We must now compute the proper entry point in the called function's code vector as a function of K and the number of args the called function wants.

    a. If number of args < minimum, signal an error.

    b. If number of args > maximum and no &REST arg is allowed, signal an error.

    c. If number of args > maximum and a &REST arg is present, pop excess args into a list, push this list back on stack as the &REST arg, and start at offset 0.

    d. If number of args is between min and max (inclusive), get the starting offset from the appropriate slot of the called function's function object. This is stored as a fixnum in slot K - MIN + 6 of the function object.

6. Set up the new PC to point at the right place in the code vector and return to the macro-code execution loop to run the new function. This involves setting up PC, the BPC, and refilling the instruction buffer.

If the object in the function entry is a list instead of a function object, we must call %SP-Internal-Apply to interpret the function with the given arguments. We proceed as follows:

1. Note the number of args pushed in the current open frame (call this N) and the frame pointer for this frame (call it F). Also remember the lambda-expression in this frame (call it L).

2. Perform steps 1 - 4 of the sequence above for a normal Start-Call.

3. Perform the equivalent of a Call-Multiple instruction with the

symbol %SP-Internal-Apply as CE. (This symbol is in a fixed location known to the microcode. See section 2.9.)

4. Push L, N, and F in that order as the three arguments to %SP-Internal-Apply.

5. Perform the equivalent of a Push-Last-Stack to start the call.

%SP-Internal-Apply, a function of three arguments, now evaluates the call to the lambda-expression or interpreted lexical closure L, obtaining the arguments from the frame pointed to by F. These arguments are obtained using the Arg-In-Frame instruction. Prior to returning %SP-Internal-Apply sets the Active-Frame register to F, so that it returns from frame F.

6.1.3. Returning from a Function Call

Return returns from the current function, popping the stack frame and pushing some number of returned values. If CE is a Values-Marker but bit 21 is not on in the current call frame, only one value is returned. If bit 21 is on, either multiple values or a single value will be returned. The steps are as follows:

1. Pop binding stack back to value saved in slot 5 of the active control frame. For each symbol/value pair popped off the binding stack, restore that value for the symbol.

2. Temp <== Previous active frame from slot 3 of current frame.

3. Open-Frame <== Saved value in current frame.

4. PC <== Saved value in current frame. This requires setting up the internal PC, the BPC, and the instruction buffer.

5. Active-Function <== Saved value from previous frame. A pointer to this frame is in Temp.

6. Active-Code <== Code Vector obtained from entry in restored Active-Function object.

7. Pop current frame off stack:

```
Control-Stack-Pointer <== Active-Frame.
Active-Frame <== Temp.
Pop top of stack into TOS register.  Since the active frame
  is inside the barrier, make sure the new top frame has been
scavenged, or do it now.
```

8. Push the return value onto the stack.

9. Resume execution of function popped to.

6.1.4. Returning Multiple-Values

If bit 21 is on in the current frame and a Values-Marker indicating N values is on the top of the stack, we proceed as follows:

1. Note the value of the current stack pointer (after CE is popped off if it came from the stack) as OLDSP.

2. Perform steps 1 - 7 of the Return procedure described above.

3. Do a block transfer loop pushing the N words starting at (OLDSP) - N onto the stack as return values. Then push the original CE, which is Values-Marker N.

4. Resume execution of the caller.

To do (MULTIPLE-VALUE-LIST (FOO A B)), we could use this sequence of instructions:

```
(CALL-MULTIPLE (CONSTANT [FOO]))
(PUSH [A])
(PUSH-LAST [B])
(FORCE-VALUES)
(VALUES-TO-N STACK)
(LIST STACK)    ;Pop N from stack, then listify N things.
```

To do (MULTIPLE-VALUE-SETQ (X Y Z) (FOO A B)), we could use this code:

```
(CALL-MULTIPLE (CONSTANT [FOO]))
(PUSH [A])
(PUSH-LAST [B])
(FORCE-VALUES)
(VALUES-TO-N STACK)
(- (CONSTANT [3]))      ;Get number offered - number wanted.
(NPOP STACK)            ;Flush surplus returns or push NILs.
(POP [Z])               ;Now put the three values wherever they
(POP [Y])               ; are supposed to go.
(POP [X])
```

In tail recursive situations, such as in the last form of a PROGN, one function, FOO, may want to call another function, BAR, and return ``whatever BAR returns.'' Call-Multiple is used in this case. If BAR returns multiple values, they will all be passed to FOO. If FOO's caller wants multiple values, the values will be returned. If not, FOO's Return instruction will see that there are multiple values on the stack, but that multiple values will not be accepted by FOO's caller. So Return will return only the first value.

6.2. Non-Local Exits

The Catch and Unwind-Protect special forms are implemented using catch frames. Unwind-Protect builds a catch frame whose tag is the Catch-All object. The Catch instruction creates a catch frame for a given tag and PC to branch to in the current instruction. The Throw instruction looks up the stack by following the chain of catch frames until it finds a frame with a matching tag or a frame with the Catch-All object as its tag. If it finds a frame with a matching tag, that frame is ``returned from,'' and that function is resumed. If it finds a frame with the Catch-All object as its tag, that frame is ``returned from,'' and in addition, %SP-Internal-Throw-Tag is set to the tag being searched for. So that interrupted cleanup forms behave correctly, %SP-Internal-Throw-Tag should be bound to the Catch-All object before the Catch-All frame is built. The protected forms are then executed, and if %SP-Internal-Throw-Tag is not the Catch-All object, its value is thrown to. Exactly what we do is this:

1. Put the contents of the Active-Catch register into a register, A. Put NIL into another register, B.

2. If A is NIL, the tag we seek isn't on the stack. Signal an Unseen-Throw-Tag error.

3. Look at the tag for the catch frame in register A. If it's the tag we're looking for, go to step 4. If it's the Catch-All object and B is NIL, copy A to B. Set A to the previous catch frame and go back to step 2.

4. If B is non-NIL, we need to execute some cleanup forms. Return into B's frame and bind %SP-Internal-Throw-Tag to the tag we're searching for. When the cleanup forms are finished executing, they'll throw to this tag again.

5. If B is NIL, return into this frame, pushing the return value (or BLTing the multiple values if this frame has bit 21 set and there are multiple values).

If no form inside of a Catch results in a Throw, the catch frame needs to be removed from the stack before execution of the function containing the throw is resumed. For now, the value produced by the forms inside the Catch form are thrown to the tag. Some sort of specialized instruction could be used for this, but right now we'll just go with the throw. The branch PC specified by a Catch instruction is part of the constants area of the function object, much like the function's entry points. To do

```
(catch 'foo
  (baz)
  (bar))
```

we could use this code:

```
            (PUSH (CONSTANT [PC-FOR-TAG-1]))
            (PUSH (CONSTANT [FOO]))
            (CATCH STACK)
            (CALL-0 (CONSTANT [BAZ]))
            (POP IGNORE)
            (CALL-0 (CONSTANT [BAR]))
            (PUSH (CONSTANT [FOO]))
            (THROW STACK)
            TAG-1
```

To do

```
     (unwind-protect
        (baz)
        (bar))
```

we could use this code:

```
            (PUSH (SYMBOL %CATCH-ALL-OBJECT))
            (PUSH (CONSTANT %SP-INTERNAL-THROW-TAG))
            (BIND STACK)
            (PUSH (CONSTANT [PC-FOR-TAG-1]))
            (CATCH-ALL STACK)
            (CALL-0 (CONSTANT [BAZ]))
            (PUSH (SYMBOL %CATCH-ALL-OBJECT))
            (THROW STACK)
            TAG-1
            (CALL-0 (CONSTANT [BAR]))
            (POP IGNORE)
            (PUSH (SYMBOL %CATCH-ALL-OBJECT))
            (EQ (SYMBOL %SP-INTERNAL-THROW-TAG))
            (BRANCH-NOT-NULL TAG-2)
            (PUSH (SYMBOL %SP-INTERNAL-THROW-TAG))
            (THROW STACK)
            TAG-2
```

6.3. Escaping to Macrocode

Some instructions can be complex (e.g. * given a long-float and a bignum), and with limited microstore (and microprogrammer time) on the PERQ, we would like to handle these in Lisp code. Such cases could be handled by a full-scale microcode-to-macrocode subroutine call, which upon a return comes back to the designated return address in the microcode and restores any micro-state that may have been clobbered. This may ultimately be needed if we ever implement a micro-compiler for lisp, but for now we can get by with a simpler scheme. If the microcode for any macro-instruction decides that it has a case too difficult to handle, it can call a macrocoded function that does whatever the original macro-instruction was supposed to do. It does this by opening an escape-type frame on the control stack, pushing an appropriate set of arguments, and then starting the call as though a push-last had been done in macrocode.

When the macrocoded escape function returns (the Escape-Return instruction must be used for this return) the single returned value goes wherever the original macro-instruction was supposed to place its result, and the original instruction stream continues on as if the macrocode instruction had exited normally without an escape.

Instructions can place their return values in any of several destinations. The escape call must set up the frame header word to indicate which of these locations is to get the value returned by the macro-coded escape function. An appropriate effective-address code is stored in bits 16-17:

0 Stack	The result is pushed onto the stack.
1 AL	The result is put into the arguments/locals area of the current call frame. Bits 0-15 contain a 16-bit offset.
2 Symbol	The result is put into the value cell of a symbol in the symbols and constants area of the current function object. Bits 0-15 contain a 16-bit offset.
3 Ignore	The result is thrown away.

Given this information in the frame header, Escape-Return will do the right thing to make it appear that the original instruction had exited normally.

Some instructions, notably Truncate, may want to return multiple values from an escape function. These values will always be returned on the stack. In this case, the escape mechanism builds a multiple-value call frame rather than an escape call frame, then escapes in the usual way. The escape routine for Truncate is exited using a normal Return instruction.

A table of pointers to the Lisp-level escape functions is stored in a fixed location in virtual memory, and the address of the start of this table is known to the microcode. This means that microcode routines can select the desired function by means of a table index, and it is not necessary to assemble the addresses of all these functions into the microcode.

The escape mechanism is implemented by a micro-subroutine named ESCAPE, which can be called (or rather, jumped to, since ESCAPE never returns to the caller) by any microcode that wants to escape to macrocode. ESCAPE is passed the index of the macro-function to be called and from 0 to 4 lisp objects as arguments on the PERQ E-Stack. ESCAPE then performs the following steps:

1. It is determined where the currently executing instruction is going to place its result, and an appropriate escape-type call header word is generated.

2. A pointer to the desired function object is fetched from the table of escape functions, as determined by the index that was passed to ESCAPE.

3. The equivalent of a Call instruction is executed for this function object, but the header word determined in step 1 is used instead of

the normal header word.

4. The specified arguments, if any, are pushed onto the control stack. The new function is then started by executing the equivalent of a Push-Last instruction.

A second entry point, ESCAPE-MULTIPLE, does the same thing as ESCAPE but creates a multiple-value frame header instead of an escape frame header.

6.4. Errors

When an error occurs during the execution of an instruction, a call to %SP-Internal-Error is performed. This call is a break-type call, so if the error is proceeded (with a Break-Return instruction), no value will be pushed on the stack.

%SP-Internal-Error is passed a fixnum error code as its first argument. The second argument is a fixnum offset into the current code vector that points to the location immediately following the instruction that encountered the trouble. From this offset, the Lisp-level error handler can reconstruct the PC of the losing instruction, which is not readily available in the micro-machine. Following the offset, there may be 0 - 2 additional arguments that provide information of possible use to the error handler. For example, an unbound-symbol error will pass the symbol in question as the third arg.

A Lisp-Level error handler may want to provide a result for the instruction. It can find the losing instruction in the way described above, and look at it's opcode to find the destination. The error handler could then store the user-supplied result in the specified place and proceed executing the errorful function at the instruction after the losing instruction.

The following error codes are currently defined. Unless otherwise specified, only the error code and the code-vector offset are passed as arguments.

The following table is pretty bogus. After the microcode is written, and I know what errors are really generated, I'll make a newer table.

1 Control Stack Overflow
The control stack has exceeded the allowable size, currently 2^{24} words.

2 Control Stack Underflow
Can only result from a compiler bug or misuse of an instruction.

3 Binding Stack Overflow
The binding stack has exceeded the allowable size, currently 2^{24} words.

4 Binding Stack Underflow
Can only result from a compiler bug or misuse of an

instruction.

5 Virtual Memory Overflow
Some data space has exceeded the maximum size of its segment in virtual memory.

6 Unbound Symbol
Attempted access to the special value of an unbound symbol. Passes the symbol as the third argument to %Sp-Internal-Error.

7 Undefined Symbol
Attempted access to the definition cell of an undefined symbol. Passes the symbol as the third argument to %Sp-Internal-Error.

8 Unused.

9 Altering T or NIL
Attempt to bind or setq the special value of T or NIL.

10 Unused.

11 Write Into Read-Only Space
Self-explanatory.

12 Object Not Character
The object is passed as the third argument.

13 Object Not System Area Pointer
The object is passed as the third argument.

14 Object Not Control Stack Pointer
The object is passed as the third argument.

15 Objot Binding Stack Pointer
The object is passed as the third argument.

16 Object Not Values Marker
The object is passed as the third argument.

17 Object Not Fixnum
The object is passed as the third argument.

18 Object Not Vector-Like
The object is passed as the third argument.

19 Object Not Integer-Vector
The object is passed as the third argument.

20 Object Not Symbol
The object is passed as the third argument.

21 Object Not List

The object is passed as the third argument.

22 Object Not List or Nil
The object is passed as the third argument.

23 Object Not String
The object is passed as the third argument.

24 Object Not Number
The object is passed as the third argument.

25 Object Not Misc Type
The object is passed as the third argument.

26 Unused.

27 Illegal Allocation Space Value
Self explanatory.

28 Illegal Vector Size
Attempt to allocate a vector with negative size or size too large for vectors of this type. Passes the requested size as the third argument.

29 Illegal Immediate Type Code
Passes the code as the third argument.

30 Illegal Control Stack Pointer
Passes the illegal pointer as the third argument.

31 Illegal Binding Stack Pointer
Passes the illegal pointer as the third argument.

32 Illegal Instruction
Must be due to a compiler error or to using obsolete code that does not match the current microcode. No additional args.

33 Unused.

34 Illegal Divisor
The divisor is integer or floating 0. Returns the divisor and dividend as the third and fourth args.

35 Illegal Vector Access Type
The specified access type is returned as the third argument.

36 Illegal Vector Index
The specified index is out of bounds for this vector. The bad index is passed as the third argument.

37 Illegal Byte Pointer
Bad S or P value to LDB or related function. Returns S and P

as the third and fourth arguments.

38 Illegal Function
Bad object being called as a function. The object is passed as the third argument.

39 Too Few Arguments
Attempt to activate the call to a function with too few arguments on the stack. Returns the number of arguments passed as the third argument, the function being called as the fourth.

40 Too Many Arguments
Attempt to activate the call to a function with too few arguments on the stack. Returns the number of arguments passed as the third argument, the function being called as the fourth.

41 Unseen Throw Tag
Returns the tag as the third argument.

42 Null Open Frame
Attempt to activate a function call, but no frame has been opened. No additional args.

43 Undefined Type Code
Can only result from a bug in the micro-machine. Returns the strange object as the third argument.

44 Return From Initial Function
Self-explanatory.

45 GC Forward Not To Newspace
Can only result from internal errors in the micro-machine. No additional args.

46 Attempt To Transport GC Forward
Can only result from internal errors in the micro-machine. No additional args.

47 Object Not Integer
The object is passed as the third argument.

48 Short-float exponent overflow, underflow
No additional args.

49 Long-float exponent overflow, underflow
No additional args.

50 - 63 Unused.

In the Tops-20 virtual machine, the following codes are defined:

64 Illegal File Token

The bad token is passed as the third argument.

65 Illegal I/O Mode Specifier
The bad mode is passed as the third argument.

6.5. Trapping to the Accent Kernel

Most of the primitive calls to the Accent kernel are made through a single microcode entry point, SVCall, defined in Accent file process.mic. From Lisp level, these calls are generated by the Kernel-Trap instruction.

Kernel-Trap takes two operands, an argument block and a trap code, in that order. The trap code is a fixnum which specifies the sort of trap call desired. The argument block is an I-Vector which contains the argument information for the trap call. The size and format of the argument block depends on which trap code is called. The return codes and values from the trap are written into elements of the I-Vector by the kernel.

Internally, the trap code and a pointer to the data portion of the I-Vector are passed to Accent on the PERQ E-Stack, as follows:

ETOS	The trap code.
ETOS - 1	The low order 16 bits of the virtual address.
ETOS - 2	The high order 16 bits of the virtual address.

All of the kernel traps called by Lisp-level code use the virtual address as a pointer to an argument block. An argument block is stored at lisp level as an I-Vector of 16-bit quantities. The trap codes are defined in Accent file accenttype.pas, and the arguments to these calls are described in the Accent Kernel Interface Manual.

6.6. Interrupts

There are three kinds of asynchronous events that the Spice Lisp system must service: hardware interrupts, process breaks, and software interrupts.

Hardware interrupts must be serviced every 70 microinstructions. It is guaranteed that no process registers will be altered and no page faults will occur, so all a microprogrammer need do is check the Intr-Pending condition every now and then, and call the hardware interrupt service routine. Sometimes that routine will set the process break flag, and a process break should occur.

If there are other runnable processes on the machine, a process break will result in the de-scheduling of the Lisp process. Process registers will be saved by the kernel, and restored when the Lisp runs again. After a process break, all cached virtual-to-physical memory translations may be invalid and the instruction buffer will probably be filled with some other process's instructions. The caches must be flushed and the instruction buffer must be refilled after a process break.

After a process break, it is possible that the Lisp process will have received an ``emergency message'' from some other process. If so, the software interrupt flag will be set. To service this software interrupt, a break-type call frame is built to %SP-Software-Interrupt-Handler, which should receive the message and figure out what to do with it at Lisp level. The emergency message might, for example, report that an interrupt character has been typed, and the interrupt handler could enter a break loop or throw to the Lisp top level.

I. Fasload File Format

I.1. General

The purpose of Fasload files is to allow concise storage and rapid loading of Lisp data, particularly function definitions. The intent is that loading a Fasload file has the same effect as loading the ASCII file from which the Fasload file was compiled, but accomplishes the tasks more efficiently. One noticeable difference, of course, is that function definitions may be in compiled form rather than S-expression form. Another is that Fasload files may specify in what parts of memory the Lisp data should be allocated. For example, constant lists used by compiled code may be regarded as read-only.

In some Lisp implementations, Fasload file formats are designed to allow sharing of code parts of the file, possibly by direct mapping of pages of the file into the address space of a process. This technique produces great performance improvements in a paged time-sharing system. Since the Spice project is to produce a distributed personal-computer network system rather than a time-sharing system, efficiencies of this type are explicitly not a goal for the Spice Lisp Fasload file format.

On the other hand, Spice Lisp is intended to be portable, as it will eventually run on a variety of machines. Therefore an explicit goal is that Fasload files shall be transportable among various implementations, to permit efficient distribution of programs in compiled form. The representations of data objects in Fasload files shall be relatively independent of such considerations as word length, number of type bits, and so on. If two implementations interpret the same macrocode (compiled code format), then Fasload files should be completely compatible. If they do not, then files not containing compiled code (so-called "Fasdump" data files) should still be compatible. While this may lead to a format which is not maximally efficient for a particular implementation, the sacrifice of a small amount of performance is deemed a worthwhile price to pay to achieve portability.

The primary assumption about data format compatibility is that all implementations can support I/O on finite streams of eight-bit bytes. By "finite" we mean that a definite end-of-file point can be detected irrespective of the content of the data stream. A Fasload file will be regarded as such a byte stream.

I.2. Strategy

A Fasload file may be regarded as a human-readable prefix followed by code in a funny little language. When interpreted, this code will cause the construction of the encoded data structures. The virtual machine which interprets this code has a stack and a table, both initially empty. The table may be thought of as an expandable register file; it is used to remember quantities which are needed more than once. The elements of both the stack and the table are Lisp data objects. Operators of the funny language may take as operands following bytes of the data stream, or items popped from the stack. Results may be pushed back onto the stack or pushed onto the table. The table is an indexable stack that is never popped; it is indexed relative to the base, not the top, so that an item once pushed always has the same index.

More precisely, a Fasload file has the following macroscopic organization. It is a sequence of zero or more groups concatenated together. End-of-file must occur at the end of the last group. Each group begins with a series of seven-bit ASCII characters terminated by one or more bytes of all ones (FF_{16}); this is called the header. Following the bytes which terminate the header is the body, a stream of bytes in the funny binary language. The body of necessity begins with a byte other than FF_{16}. The body is terminated by the operation FOP-END-GROUP.

The first nine characters of the header must be "FASL FILE" in upper-case letters. The rest may be any ASCII text, but by convention it is formatted in a certain way. The header is divided into lines, which are grouped into paragraphs. A paragraph begins with a line which does not begin with a space or tab character, and contains all lines up to, but not including, the next such line. The first word of a paragraph, defined to be all characters up to but not including the first space, tab, or end-of-line character, is the name of the paragraph. A Fasload file header might look something like this:

```
FASL FILE >SteelesPerq>User>Guy>IoHacks>Pretty-Print.Slisp
Package Pretty-Print
Compiled 31-Mar-1988 09:01:32 by some random luser
Compiler Version 1.6, Lisp Version 3.0.
Functions: INITIALIZE DRIVER HACK HACK1 MUNGE MUNGE1 GAZORCH
           MINGLE MUDDLE PERTURB OVERDRIVE GOBBLE-KEYBOARD FRY-USER
           DROP-DEAD HELP CLEAR-MICROCODE %AOS-TRIANGLE
           %HARASS-READTABLE-MAYBE
Macros:    PUSH POP FROB TWIDDLE
<one or more bytes of FF  >
                        16
```

The particular paragraph names and contents shown here are only intended as suggestions.

I.3. Fasload Language

Each operation in the binary Fasload language is an eight-bit (one-byte) opcode. Each has a name beginning with "FOP-". In the following descriptions, the name is followed by operand descriptors. Each descriptor denotes operands that follow the opcode in the input stream. A quantity in parentheses indicates the number of bytes of data from the stream making up the operand. Operands which implicitly come from the stack are noted in the text. The notation "@z[@] stack" means that the result is pushed onto the stack; "@z[@] table" similarly means that the result is added to the table. A construction like "n(1) value(n)" means that first a single byte n is read from the input stream, and this byte specifies how many bytes to read as the operand named value. All numeric values are unsigned binary integers unless otherwise specified. Values described as "signed" are in two's-complement form unless otherwise specified. When an integer read from the stream occupies more than one byte, the first byte read is the least significant byte, and the last byte read is the most significant (and contains the sign bit as its high-order bit if the entire integer is signed).

Some of the operations are not necessary, but are rather special cases of or combinations of others. These are included to reduce the size of the file or to speed up important cases. As an example, nearly all strings are less than 256 bytes long, and so a special form of string operation might take a one-byte length rather than a four-byte length. As another example, some implementations may choose to store bits in an array in a left-to-right format within each word, rather than right-to-left. The Fasload file format may support both formats, with one being significantly more efficient than the other for a given implementation. The compiler for any implementation may generate the more efficient form for that implementation, and yet compatibility can be maintained by requiring all implementations to support both formats in Fasload files.

Measurements are to be made to determine which operation codes are worthwhile; little-used operations may be discarded and new ones added. After a point the definition will be "frozen", meaning that existing operations may not be deleted (though new ones may be added; some operations codes will be reserved for that purpose).

0 FOP-NOP No operation. (This is included because it is recognized that some implementations may benefit from alignment of operands to some operations, for example to 32-bit boundaries. This operation can be used to pad the instruction stream to a desired bounary.)

1 FOP-POP @z[@] table
One item is popped from the stack and added to the table.

2 FOP-PUSH index(4) @z[@] stack
Item number index of the table is pushed onto the stack. The first element of the table is item number zero.

3 FOP-BYTE-PUSH index(1) @z[@] stack
Item number index of the table is pushed onto the stack. The first element of the table is item number zero.

4 FOP-EMPTY-LIST @z[@] stack
The empty list (()) is pushed onto the stack.

5 FOP-TRUTH @z[@] stack
The standard truth value (T) is pushed onto the stack.

6 FOP-SYMBOL-SAVE n(4) name(n) @z[@] stack & table
The four-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

7 FOP-SMALL-SYMBOL-SAVE n(1) name(n) @z[@] stack & table
The one-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with

the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

8 FOP-SYMBOL-IN-PACKAGE-SAVE index(4) n(4) name(n) @z[@] stack & table
The four-byte index specifies a package stored in the table. The four-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

9 FOP-SMALL-SYMBOL-IN-PACKAGE-SAVE index(4) n(1) name(n) @z[@] stack & table
The four-byte index specifies a package stored in the table. The one-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

10 FOP-SYMBOL-IN-BYTE-PACKAGE-SAVE index(1) n(4) name(n) @z[@] stack & table
The one-byte index specifies a package stored in the table. The four-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

11 FOP-SMALL-SYMBOL-IN-BYTE-PACKAGE-SAVE index(1) n(1) name(n) @z[@] stack & table
The one-byte index specifies a package stored in the table. The one-byte operand n specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

12 Unused.

13 FOP-DEFAULT-PACKAGE index(4)
A package stored in the table entry specified by index is made the default package for future FOP-SYMBOL and FOP-SMALL-SYMBOL interning operations. (These package FOPs may change or disappear as the package system is determined.)

14 FOP-PACKAGE @z[@] table
An item is popped from the stack; it must be a symbol. The package of that name is located and pushed onto the table.

15   FOP-LIST   length(1)   @z[@]   stack

The unsigned operand length specifies a number of operands to be popped from the stack. These are made into a list of that length, and the list is pushed onto the stack. The first item popped from the stack becomes the last element of the list, and so on. Hence an iterative loop can start with the empty list and perform "pop an item and cons it onto the list" length times. (Lists of length greater than 255 can be made by using FOP-LIST* repeatedly.)

16   FOP-LIST*   length(1)   @z[@]   stack

This is like FOP-LIST except that the constructed list is terminated not by () (the empty list), but by an item popped from the stack before any others are. Therefore length+1 items are popped in all. Hence an iterative loop can start with a popped item and perform "pop an item and cons it onto the list" length+1 times.

17-24   FOP-LIST-1, FOP-LIST-2, ..., FOP-LIST-8

FOP-LIST-k is like FOP-LIST with a byte containing k following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of k.

25-32   FOP-LIST*-1, FOP-LIST*-2, ..., FOP-LIST*-8

FOP-LIST*-k is like FOP-LIST* with a byte containing k following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of k.

33   FOP-INTEGER   n(4)   value(n)   @z[@]   stack

A four-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

34   FOP-SMALL-INTEGER   n(1)   value(n)   @z[@]   stack

A one-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

35   FOP-WORD-INTEGER   value(4)   @z[@]   stack

A four-byte signed integer (in the range -2^{31} to $2^{31}-1$) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

36   FOP-BYTE-INTEGER   value(1)   @z[@]   stack

A one-byte signed integer (in the range -128 to 127) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

37 FOP-STRING n(4) name(n) @z[@] stack
The four-byte operand n specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

38 FOP-SMALL-STRING n(1) name(n) @z[@] stack
The one-byte operand n specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

39 FOP-VECTOR n(4) @z[@] stack
The four-byte operand n specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

40 FOP-SMALL-VECTOR n(1) @z[@] stack
The one-byte operand n specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

41 FOP-UNIFORM-VECTOR n(4) @z[@] stack
The four-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

42 FOP-SMALL-UNIFORM-VECTOR n(1) @z[@] stack
The one-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

43 FOP-INT-VECTOR n(4) size(1) count(1)
data(@z[T]n/count@z[U]@z[T]size*count/8@z[U]) @z[@] stack
The four-byte operand n specifies the length of a vector of unsigned integers to be constructed. Each integer is size bits big, and are packed in the data stream in sections of count apiece. Each section occupies an integral number of bytes. If the bytes of a section are lined up in a row, with the first byte read at the right, and successive bytes placed to the left, with the bits within a byte being arranged so that the low-order bit is to the right, then the integers of the section are successive groups of size bits, starting from the right and running across byte boundaries. (In other words, this is a consistent right-to-left convention.) Any bits wasted at the left end of a section are ignored, and any wasted groups in the last section are ignored. It is permitted for the loading implementation to use a vector format providing more precision than is required by size. For example, if size were 3, it

would be permitted to use a vector of 4-bit integers, or even vector of general LISP objects filled with integer LISP objects. However, an implementation is expected to use the most restrictive format that will suffice, and is expected to reconstruct objects identical to those output if the Fasload file was produced by the same implementation. (For the PERQ U-vector formats, one would have size an element of {1, 2, 4, 8, 16}, and count=32/size; words could be read directly into the U-vector. This operation provides a very general format whereby almost any conceivable implementation can output in its preferred packed format, and another can read it meaningfully; by checking at the beginning for good cases, loading can still proceed quickly.) The constructed vector is pushed onto the stack.

44 FOP-UNIFORM-INT-VECTOR n(4) size(1) value(@z[T]size/8@z[U]) @z[@] stack
The four-byte operand n specifies the length of a vector of unsigned integers to construct. Each integer is size bits big, and is initialized to the value of the operand value. The constructed vector is pushed onto the stack.

45 FOP-FLOAT n(1) exponent(@z[T]n/8@z[U]) m(1) mantissa(@z[T]m/8@z[U]) @z[@] stack
The first operand n is one unsigned byte, and describes the number of bits in the second operand exponent, which is a signed integer in two's-complement format. The high-order bits of the last (most significant) byte of exponent shall equal the sign bit. Similar remarks apply to m and mantissa. The value denoted by these four operands is
exponent-length(mantissa
mantissax2). A floating-point number shall be constructed which has this value, and then pushed onto the stack. That floating-point format should be used which is the smallest (most compact) provided by the implementation which nevertheless provides enough accuracy to represent both the exponent and the mantissa correctly.

46-51 Unused

52 FOP-ALTER index(1)
Two items are popped from the stack; call the first newval and the second object. The component of object specified by index is altered to contain newval. The precise operation depends on the type of object:

List	A zero index means alter the car (perform RPLACA), and index=1 means alter the cdr (RPLACD).
Symbol	By definition these indices have the following meaning, and have nothing to do with the actual

representation of symbols in a given implementation:

0	Alter value cell.
1	Alter function cell.
2	Alter property list (!).

Vector (of any kind)
Alter component number index of the vector.

String Alter character number index of the string.

53 FOP-EVAL @z[@] stack
Pop an item from the stack and evaluate it (give it to EVAL). Push the result back onto the stack.

54 FOP-EVAL-FOR-EFFECT
Pop an item from the stack and evaluate it (give it to EVAL). The result is ignored.

55 FOP-FUNCALL nargs(1) @z[@] stack
Pop nargs+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). Push the result back onto the stack.

56 FOP-FUNCALL-FOR-EFFECT nargs(1)
Pop nargs+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). The result is ignored.

57 FOP-CODE-FORMAT id(1)
The operand id is a unique identifier specifying the format for following code objects. The operations FOP-CODE and its relatives may not occur in a group until after FOP-CODE-FORMAT has appeared; there is no default format. This is provided so that several compiled code formats may co-exist in a file, and so that a loader can determine whether or not code was compiled by the correct compiler for the implementation being loaded into. So far the following code format identifiers are defined:

0	PERQ
1	VAX

58 FOP-CODE nitems(4) size(4) code(size) @z[@] stack
A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of FOP-CODE-FORMAT. The operand nitems specifies a

number of items to pop off the stack to use in the "boxed storage" section. The operand code is a string of bytes constituting the compiled executable code.

59 FOP-SMALL-CODE nitems(1) size(2) code(size) @z[@] stack

A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of FOP-CODE-FORMAT. The operand nitems specifies a number of items to pop off the stack to use in the "boxed storage" section. The operand code is a string of bytes constituting the compiled executable code.

60 FOP-STATIC-HEAP

Until further notice operations which allocate data structures may allocate them in the static area rather than the dynamic area. (The default area for allocation is the dynamic area; this default is reset whenever a new group is begun. This command is of an advisory nature; implementations with no static heap can ignore it.)

61 FOP-DYNAMIC-HEAP

Following storage allocation should be in the dynamic area.

62 FOP-VERIFY-TABLE-SIZE size(4)

If the current size of the table is not equal to size, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

63 FOP-VERIFY-EMPTY-STACK

If the stack is not currently empty, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

64 FOP-END-GROUP

This is the last operation of a group. If this is not the last byte of the file, then a new group follows; the next nine bytes must be "FASL FILE".

65 FOP-POP-FOR-EFFECT stack @z[@]

One item is popped from the stack.

66 FOP-MISC-TRAP @z[@] stack

A trap object is pushed onto the stack.

67 FOP-READ-ONLY-HEAP

Following storage allocation may be in a read-only heap. (For symbols, the symbol itself may not be in a read-only area, but its print name (a string) may be. This command is of an advisory nature; implementations with no read-only heap can ignore it, or use a static heap.)

68 FOP-CHARACTER character(3) @z[@] stack
The three bytes specify the 24 bits of a Spice Lisp character object. The bytes, lowest first, represent the code, control, and font bits. A character is constructed and pushed onto the stack.

69 FOP-SHORT-CHARACTER character(1) @z[@] stack
The one byte specifies the lower eight bits of a spice lisp character object (the code). A character is constructed with zero control and zero font attributes and pushed onto the stack.

70 FOP-RATIO @z[@] stack
Creates a ratio from two integers popped from the stack. The denominator is popped first, the numerator second.

71 FOP-COMPLEX @z[@] stack
Creates a complex number from two numbers popped from the stack. The imaginary part is popped first, the real part second.

72 FOP-LINK-ADDRESS-FIXUP nargs(1) restp(1) offset(4) @z[@] stack
Valid only for when FOP-CODE-FORMAT corresponds to the Vax. This operation pops a symbol and a code object from the stack and pushes a modified code object back onto the stack according to the needs of the runtime Vax code linker.

73 FOP-LINK-FUNCTION-FIXUP offset(4) @z[@] stack
Valid only for when FOP-CODE-FORMAT corresponds to the Vax. This operation pops a symbol and a code object from the stack and pushes a modified code object back onto the stack according to the needs of the runtime Vax code linker.

74 FOP-FSET
Pops the top two things off of the stack and uses them as arguments to FSET (i.e. SETF of SYMBOL-FUNCTION).

255 FOP-END-HEADER
Indicates the end of a group header, as described above.

II. The Opcode Definition File

Index