CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

An Analysis of the Spice Lisp Instruction Set

Joseph R. Ginder

18 January 1983

Spice Document Sxxx

Keywords and index categories: PE Lisp

Location of machine-readable file: STAT.MSS.275 @ CMU-20C

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

List of Tables

Abstract

In order to increase program locality (thereby decreasing page swapping on computers with small physical memory and increasing the cache hit-ratio on those computers which have caches) the size of object code programs should be as small as possible. By introducing a distinction between the instruction set used by the compiler and that generated by the assembler, it is possible to encode the instruction set of the target machine to reduce the size of object code programs without introducing added complexity into the compiler. Other improvements in the assembler's instruction set require that the compiler instruction set be updated. The micro-coded instruction set for Spice Lisp on the Perq is amenable to this sort of optimization.

1. Introduction

This paper is an analysis of the instruction set for Spice Lisp based on static instruction and operand frequencies. Dynamic execution statistics are being collected to be used in an analysis similar to those performed in [5] and [10]. The Spice Lisp instruction set was designed for a single user, user microcodable machine (currently the Perq).[1]

Most interesting computer programs require a large amount of virtual memory. On small, personal computers a major bottleneck for system performance is primary to secondary memory bandwidth. Therefore, it is advantageous to have a working set that can fit in primary memory, incurring as little swapping cost as possible. On those computers which have instruction caches, program size should be minimized in order to increase the locality of programs so that the cache hit-ratio is improved, thereby increasing performance. By refining the compiler to generate compact object programs, program runtime can be reduced. In refining the compiler in this manner, it is important to remember that instructions should not be designed so that many levels of instruction decoding are required by the microcode; this might actually increase program runtime. The goal of this analysis is to determine ways to reduce Spice Lisp object program size by adding optimizing instructions in a non-disruptive manner. Other optimizations have become apparent in the analysis and are included in this paper. A similar analysis of the Mesa instruction set with recommended improvements is described by Sweet and Sandman in [9].

From the compiler writer's point of view, a computer architecture should have a regularity of structure; orthogonality should not be violated; arbitrary composing of these regular, orthogonal notions should be allowed [11], [1]. However, since all instructions and addressing modes are not used equally in practice, certain optimizations are suggested. In order not to violate the above principles, one key criterion for any optimization involving the addition of instructions should be kept in mind. *Any instructions added for optimization should provide a special case of some more general construction.* [4] The main phase of the compiler will generate the general instruction; the assembler will choose the most efficient specialized form of that instruction. If no specialized form is available (presumably because it has not been warranted by analysis

[1] While designed for a personal machine [7], Spice Lisp adheres to the Common Lisp specification [2]; Common Lisp will be available on multi-user machines.

of instruction and operand frequencies) a general form is generated by the assembler[2].

This sort of optimization suggests a different sort of methodology to be used in designing the compiler and assembler. In order to make the compiler writer's job easier, the compiler is designing to generate instructions which are members of an idealized, general instruction set which is not limited by opcode space. The assembler translates these idealized instructions into those actually available for execution by the machine. It is important to note that these instruction sets are of approximately the same level of abstraction -- there is little or no semantic gap between them. The salient difference between them is that the compiler instruction set is designed for easy compilation -- the assembler instruction set for short object program size and fast execution speed.[3] In order to make room for new, specialized instructions, opcode space must be made available. Elimination of rarely-used instructions from the instruction set generated by the assembler will open up a substantial portion of the presently used opcodes for re-use. *An instruction generated by the compiler need not have a unique, corresponding opcode in the actual instruction set which the assembler generates.*

An argument against allocating opcodes in this fashion may be made by claiming that this places an undue burden on the compiler by infringing upon the regularity of the instruction set (see [11]). However, if we restrict the necessity for more complicated instruction selection to a particular, specialized phase of compilation (ie. assembly), the increase in object code compactness should more than offset the added complexity of instruction assembly. For example, the Spice Lisp compiler may continue to assume that an instruction such as *Check*[4] with operand *stack* exists and generate something of the form *(check stack)* as input to the assembler even if this instruction is not directly encoded. The assembler may then generate the opcode for *(pop ignore)*, which has the same effect. The essence of this argument is that there need not be a one-to-one correspondence between the instructions generated by the compiler and those opcodes generated by the assembler. Perhaps a more appropriate way to view the above mapping of *(check stack)* and *(pop ignore)* to one opcode is to see this as two instructions having the same assembled opcode. Another sort of

---

[2] An interesting side issue given this sort of specialized instruction set is whether or not the general form of an instruction should include those options available as specialized forms. From general information theory [6], it can be shown that to achieve greater program entropy (where entropy is a measure of the average information content -- see the thesis by Sweet [8] for a discussion of program entropy) the general form should be designed so that there is no redundancy with the specialized forms. For example, if one special form of a Push instruction pushed the constant 0, then the range of constants available for the general form should be realigned so as not to include 0. However, this approach increases the complexity of the assembler and can significantly reduce regularity between the general forms generated by the assembler. It should also be pointed out that once this approach is taken, all compilers must use the optimizing assembler.

[3] An additional advantage of this scheme is that the compiler will be much more portable. Only a new assembler need be written for each new machine.

[4] See Appendix I for a description of the Spice Lisp instruction set.

mapping that would be possible is that which occurs when one sort of compiler instruction (eg. *(Pop arg&local N)*) maps to several different opcodes depending on the value of N. It would also be possible for the compiler to generate rarely-used instructions for which there was no corresponding opcode to be generated by the assembler. In these instances, the assembler might substitute a short sequence of instructions which had the same effect. In no case is the compiler writer forced to take these optimizations into account when writing the main body of the compiler. Other optimization techniques (flow analysis or constant folding, for example) are orthogonal to this technique.

Another argument against this sort of opcode allocation is that the compiler could generate better object code if the compiler writer could make use of the knowledge that particular instructions were expressed in fewer bytes. It may be possible to tune the compiler to generate more compact code by doing complicated analysis of the number of references to particular operands, etc. This would make the compiler much more complex. This problem can be avoided by using the appropriate methodology in designed and refining the instruction set and compiler. The compiler should be made extremely regular -- for example, it should allocate arguments and local variables to particular slots in the control stack in a uniform manner. (One fairly simple approach to allocating slots on the control stack would be to allocate slots starting with the most frequently referenced local variable or argument.) If the compiler is designed to build functions and stack blocks in a uniform manner, patterns of reference into the vector of symbols and constants and into the arguments and local variables of the control stack can be recognized through analysis (both static *and dynamic*) of object code. Using these patterns, the instruction set may be modified and tuned to optimize the number of bytes needed to represent the most common instructions and operands. *All of these modifications to the instruction set may be made so as to require changes only to the assembler.* This process may be repeated throughout the lifetime of the compiler and instruction set. The aspect of this process which eliminates much of the expected complexity in the compiler is that the specialized opcodes and many-to-one instruction to opcode mappings are added only to the instruction set generated by the *assembler* after statistical analysis of the instructions generated by the compiler. This statistical analysis can also be used to suggest new instructions to be generated by the compiler (for an example, see 5.4). Similar methodologies have been used to some success and are described in [9] and [4].

In some instruction sets, instruction opcodes may vary in length (eg. 1 byte and 2 byte instructions). One or more of the short opcodes is interpreted as an escape code which indicates that another byte should be fetched and used as the instruction opcode. Another advantage to designing separate instruction sets for the compiler and the assembler is that the compiler need not divide its instructions into the long instruction and short instruction categories; this complication may be reserved for the assembler only.

After discussing those aspects of Spice Lisp architecture which are helpful in understanding the instruction

set, I will present statistics summarizing the findings of an analysis of static instruction and operand frequencies (section 3). In section 4 I will describe changes to the instruction set which will reduce decoding time for the most common sort of operand. Following this will be a discussion of changes which will enable the assembler to generate smaller object code programs. These changes fall into the following categories:

- allow some short instructions to imply a specific operand in the instruction opcode (section 5)
- convert heavily used long instructions to short instructions (section 5)
- convert rarely-used short instructions to long instructions (section 6)
- eliminate illegal instruction-opcode combinations (section 6)
- collapse frequent instruction pairs into single instructions in both the compiler instruction set and the assembler instruction set (section 5)
- collapse frequent instruction pairs into single instructions in only the assembler instruction set (section 5)
- eliminate infrequently used instructions that are easily expressed as a sequence of two or more other instructions (section 6)

Appendix I is an edited description of the instruction set of the current Spice Lisp architecture taken from *The Internal Design of Spice Lisp* [3]. Appendix II is a parallel version of these descriptions which is intended as a description of the instruction set of a new assembler for Spice Lisp, including the recommended optimizations. Appendix III is a description of the instruction set which should be generated by the main phase of the compiler; it is these general instructions which the new assembler should translate into the specialized instruction set of Appendix II.

2. Some Notes on Spice Lisp Architecture

2.1 The Execution Environment

The Spice Lisp implementation architecture is stack based. The control stack is used to stack function call frames. Each frame contains pointers to the previously-active frame, the most recent open frame, and the point to which the binding stack is to be popped upon function return. Each frame also contains storage locations for the function's arguments and local variables. Spice Lisp instructions may specify an argument or local variable as operand. Frames also include a slot for a pointer to the function object which contains the compiled code for that function. A function frame on the control stack is arranged as follows:

0 Header word.
1 Function object or EXPR for this call.
2 Closure List (or NIL if not a closure).
3 Pointer to previous active frame.
4 Pointer to previous open frame.
5 Pointer to previous binding stack.
6 Saved PC of caller. (An integer address)
7 **Arguments-and-local-variables** block starts here. (Entry 0)
...
N Frame Barrier. Push after the args and locals.

2.2 Function Objects

Each compiled function is represented in the machine as a Function Object. The function object contains a vector of information needed by the function-calling mechanism. This vector includes a pointer to the vector that holds the actual code, the number of required and optional arguments, and a few other things. Following this information is a vector of symbol pointers (for symbols that are used as special variables in the code) and constants. One addressing mode for Spice Lisp instructions may access this vector. A constant is any Lisp object that is used but not altered by the function. Integer constants in the range of -128 to +127 can be expressed as immediate operands, and so do not need to be represented here as full-word constants.

2.3 Instruction Set and Addressing Modes

The instruction set used for Spice Lisp is based on that used by the MIT Lisp Machine.[5] However, while

[5]Since we have no locative pointers and no CDR coding in the current Spice Lisp, some of the instructions used on the Lisp Machine have no counterparts in the Spice Lisp instruction set.

the Lisp Machine instruction set was designed to fit into 16-bit words, there is a clear advantage to fitting Spice Lisp instructions into 8-bit bytes on the Perq and several other machines to which Spice Lisp may be ported, even if more bytes must usually be fetched for operands. Therefore Spice Lisp object programs are organized into a stream of 8-bit bytes. Each instruction (including operand specification fields) may vary from 1 byte to 5 bytes in length. Several operations have been added in order to correct weaknesses noticed in the MIT instruction set (eg. operations for accessing vectors and strings). Operands are of the following types:

- top-of-stack
- indirect through top-of-stack
- immediate short constant
- offset into the arguments and local variables of the active stack frame
- offset into the vector of symbols and constants of the current function object
- offset indirect through the vector of symbols and constants of the current function object

Currently, the addressing mode of an instruction is specified by a 2-bit field, the A-field. While 2-bits do not carry enough information to distinguish between all of the above addressing modes, ambiguity can be resolved by additional operand bytes. These addressing modes are tailored specifically to facilitate implementing Lisp; no attempt is made to address the general "primitives vs. solutions" problem expressed by Wulf in [11]. Specific "solution" type instructions for implementing a Lisp system are implemented in microcode. In particular, addressing modes assume a certain stack frame layout.

There are two classes of instructions available for Spice Lisp. Those instructions specifiable in 1 byte (without operands) are designated *short* instructions. Four 1-byte short instructions are escape codes that are interpreted to mean that a second byte should be fetched to specify an operation. These 2-byte instructions are designated *long* instructions. Long instructions normally are used in conjunction with arguments which are set up explicitly on the stack. For example, the *Cons* long instruction expects the two arguments for consing to be at the top of the stack. For more detailed information, see [3].

Spice Lisp instructions may also be divided into to classes according to whether the instruction is a branch instruction or not. This distinction is conceptually orthogonal to the short/long distinction.

2.4 Indicators

Conceptually, the result produced by each instruction is used to set a group of indicators, which can be tested by subsequent conditional-jump instructions. These indicators are NULL, ATOM, and ZERO. In the description of instructions below, if it is unclear what the natural "result" is, it will be stated explicitly what value goes into the indicators; in some cases, instructions leave the indicators unchanged.

Under some conditions branching depending upon the setting of an indicator may require that instructions be generated for the sole purpose of setting the indicators; when this is the case, a branch architecture which does not use indicators would require fewer bytes of instructions. For example, (*Check* X) (*Branch-Null* N) might be expressed in a "no-indicator" architecture as (*Branch-Null* X N). The former requiring 4 bytes to express, the latter 3. However, when the quantity being tested for a conditional branch has already (or would have already) set the indicators, an indicator architecture requires fewer bytes of instructions. For example, (> X) (*Branch-Zero* N) might be expressed in a no-indicator architecture as (> X) (*Branch-Zero* X N); the former requiring 4 bytes to express, the latter 5. Statistics collected for this evaluation show that having an indicator architecture is much more often an advantage than a disadvantage. However, the addition of several branch instructions which do not branch based on the indicators allows the compiler to generate indicator-using branches or non-indicator-using branches as the situation warrants. This possibility is explored in section 5.3.

2.5 Design Philosophy

This instruction set was designed so that there would be a direct mapping from the S-expression source to the instruction set. Because of this direct mapping, no assembly code for Spice Lisp should ever be written by hand; it should only be generated by the compiler. Because only the compiler should generate these instructions, various specialized instructions may be added to the instruction set in order to reduce compiled program size with the only cost being a small amount of added complexity in the compiler (and, of course, opcode space.)

The division of the original instruction set into short and long instructions was based on the designer's intuitions as to which instructions would be generated and executed most frequently. While this intuition is shown to be largely correct by the statistics collected to date, substantial improvement in code size may be achieved by rearranging the instructions and by optimizing operand specification based on a statistical analysis.

3. Statistical Results

The code available for statistical analysis consisted entirely of Spice Lisp system implementation code. While this sample cannot be construed as representative of user code, the instruction frequencies found in this sample should be sufficient for a first pass at refining the Spice Lisp instruction set.

Statistics were gathered by two MacLisp programs which scan compiled Spice Lisp code. One program counted occurrences of each instruction and kept counts of the types of operands used by each instruction. The other MacLisp program collected information about specific operand values for those instructions for which this detailed information was deemed useful.

The object code programs analyzed in this study contained over 1 million instructions. These object programs had been optimized by a peephole optimizer which typically achieves at least a 25% reduction in the number of instructions over unoptimized code. These 1,072,306 instructions occupied 2,022,694 bytes for an average of 1.89 bytes per instruction. Table 3-1 summarizes the statistical results collected by the first program.

Instruction Type	Number	% of Instructions	Bytes	% of Bytes	Bytes/Instruction
Short Non-Branch	842,306	78.6	1,534,331	75.8	1.82
Branch	133,595	12.5	276,752	13.6	2.07
Long Non-Branch	96,405	9.0	211,611	10.5	2.20
Totals	1,072,306		2,022,694		1.89

Table 3-1: Instruction Type Statistics

Each instruction makes use of an operand as summarized in Table 3-2. The first part of the table summarizes operand frequencies for non-branch instructions. For each of these instructions an operand is specified from the group listed in the first part of the table. Some of these instructions implicitly make use of the top of the stack (particularly long instructions); these instances of implicit stack operands are not included in this table. The second portion of the table summarizes operand frequencies for branch instructions. Every branch instruction makes use of a PC-relative offset. One branch instruction (Branch-if-arg-supplied, see Appendix A) makes use of an additional argument. It is also possible for branch instructions to pop the stack. Optimizations based on these statistics are suggested in section 4.

The 15 instructions which occur most frequently are listed in Table 3-3 along with operand statistics. It is interesting to note that Push and Push-Last account for 389,187 instructions, or fully 36% of all instructions. This translates into 758,217 bytes of code (1.95 bytes/instruction), or 38% of the bytes of object programs. This implies that substantial saving may be possible if the average number of bytes per Push/Push-Last

Non-branch operand type	Number	% of Total
Stack	175,964	18.7
Short constant	115,530	12.3
Ignore	52,871	5.6
Symbols & Constants	154,822	16.5
Arguments & Local Variables	402,873	42.9
Symbol Pointers	36,651	3.9
Total	938,711	100.0

Branch operand	Number	% of Branch Instructions Using
PC-relative Offset	133,595	100.0
Arguments & Local Variables	3954	3.0

Table 3-2: Operand Frequencies

Instruction	Stack	Short Constant/ Ignore	Symbols & Constants	Arguments & Locals	Symbol Pointers	Total
Push	0	61,959	63,652	177,803	16,969	320,383
Pop	0	21,969	0	55,778	1846	79,593
Push-Last	21,069	1462	9365	33,281	3627	68,804
Call	1234	0	44,342	5543	146	51,265
Return	37,468	133	1606	7593	202	47,002
=	1243	20,802	725	6169	25	28,964
Set-Null	14,981	556	0	5680	511	21,728
Car	257	0	580	15,844	3213	19,894
V-Access (long)	13,476	2915	0	3078	0	19,469
Eq	725	103	6001	10,912	655	18,396
Set-0	12,891	0	0	5004	72	17,967
V-Store (long)	439	17,924	0	0	0	17,733

Branch Instruction	1-byte offsets	2-byte offsets	Total
Branch	38,163	3909	42,072
Branch-Not-Null	36,906	4376	41,282
Branch-Null	36,856	932	37,788

Table 3-3: Top 15 Instructions

instruction can be reduced. This possibility is explored in section 5. Branch instruction optimizations are discussed in section 6.3. The 15 most frequent instructions account for 832,340 of the 1,072,306 total instructions generated, or 77.6%. This translates into 1,565,532 bytes of code, or 77.4% of all the bytes of object code.

It is also interesting to note that 400,177 instructions (or 779,096 bytes) were used for the purpose of

manipulating the stack[6]. This means that 37.3% of all instructions, or 38.5% of all instruction bytes, were for stack manipulation. A useful experiment would be to compare the code generated for this stack architecture with code generated for other non-stack architectures in order to determine the relative overheads of the competing architectures.

The 15 most frequently occurring long instructions are summarized in Table 3-4. Optimizations involving these and other long instructions are described in section 5.2.

Long Instruction	Stack	Ignore	Symbols & Constants	Arguments & Locals	Symbol Pointers	Total
V-Access	13,476	2915	0	3078	0	19,469
V-Store	439	17,294	0	0	0	17,733
Get-Definition	7871	0	0	656	0	8527
Type	4680	16	0	3644	0	8340
Cons	2510	0	0	1966	470	4946
Make-Immediate-Type	2907	0	0	717	7	3631
Logldb	2085	96	0	492	0	2673
Get-Vector-Subtype	1536	0	0	839	0	2375
Get-Vector-Length	1012	0	0	1280	0	2292
Typed-V-Access	1582	36	0	402	0	2020
Negate	1307	0	0	684	19	2010
Typed-V-Store	38	1962	0	0	0	2000
Ldb	1427	174	0	283	0	1884
Rplacd	798	1019	0	60	0	1877
Get-Value	173	0	0	1452	0	1625

Table 3-4: 15 Most Common Long Instructions

The 15 most common long instructions account for 81,402 of the 96,405 long instruction occurrences. Thus 15 of the 77 long instructions (19.5%) account for 84.4% of the long instruction occurrences.

Another type of statistics collected was the frequency of occurrence of instruction pairs. The 10 most common instruction pairs are listed in Table 3-5. Optimizations suggested by analysis of these instruction pair frequencies are discussed in section 5.4.

---

[6]This includes the *Push*, *Push-Under*, and *Pop* instructions, but not the *Push-Last* instruction because *Push-Last* also serves to close a stack frame and end the function call process.

Instruction Pair	Number of Occurrences
Call, Call	3683
Check, Branch-Not-Atom	3611
Push, <	3425
<, Branch-Null	3390
List, Push	3338
1+, Branch	3318
Bind-Pop, Push	3308
Push, Copy	3177
Car, Pop	3147
Push, 1+	3050

Table 3-5: 10 Most Common Instruction Pairs

4. Optimization of Operand Source Decoding

4.1 A-Field Re-Alignment

In the current encoding of the A-field of non-branch instructions there is a significantly skewed frequency of occurrence for the four A-field values (see Table 4-1).

A-Code	Current Operand Type	Percentage
0	stack	18.7
1	short constant, ignore	17.9
2	arguments and locals, symbols and constants	59.4
3	symbol pointers	3.9

Table 4-1: A-field Values

By far the most frequent case is when A is 2, designating that the operand is either in the arguments and local variables section of the current stack frame or in the symbols and constants vector of the code object. Another byte is fetched to determine which of these areas is accessed. If the sign bit of this byte is 0, the remaining 7 bits of the byte is an unsigned offset into the vector of symbols and constants. If the sign bit is 1, then the remaining 7 bits of the byte is an unsigned offset into the arguments and local variables area of the stack frame. In analyzing the operand type frequencies, we notice that the most frequent type of operand is argument or local variable (see Table 4-2). By redefining the A-codes as in Figure 4-3, we may achieve a savings of one level of decoding for the most frequent type of operand. In the microcoded implementation of Spice Lisp on the Perq instructions may take varying lengths of time to execute; therefore this savings can be realized. An additional advantage of this scheme is that the range of arguments and local variables specifiable in a one byte operand is doubled.

Operand Type	Percentage
stack	18.7
short constant	12.3
ignore	5.6
symbols and constants	16.5
arguments and locals	42.9
symbol pointers	3.9

Table 4-2: Operand Frequencies

A-Code	Revised Operand Type	Percentage
0	stack	18.7
1	short constant, ignore	17.9
2	arguments and locals	42.9
3	symbols and constants, symbol pointers	20.4

Table 4-3: New A-field Frequencies

4.2 Two-Byte Offsets

Any time an instruction specifies an arguments and locals or symbols and constants operand (63.3% of all instructions use these types of operands) the instruction decoding microcode must check for a two-byte offset. While the option of having stack frames and vectors of symbols and constants which require more than one byte of offset should be retained, there is no need to spend microcycles checking for this possibility when decoding every instruction. Out of over 1 million instructions generated for the current Spice Lisp implementation, only 475 make use of a two-byte offset (all of these are source operand offsets into the vector of symbols and constants). In order to retain the use of two-byte offsets while eliminating the need to check for them when decoding most instructions, one new short instructions and three new long instructions should be added to the assembler's instruction set. These are:

- Push-Long-AL
- Push-Long-SC (short instruction)
- Pop-Long-AL
- Pop-Long-SP[7]

When the compiler generates instructions which require a two-byte offset, the assembler may simply substitute the same instruction with the stack as operand either preceded or followed by one of the four new instructions, as appropriate. For example, *(+ (AL LONG-OFFSET))* may be replaced by *(Push-Long-AL LONG-OFFSET), (+ Stack).* In the current instruction set, this action requires 4 bytes to express (one byte for the opcode, one for the escape-to-long-offset code, and two for the long offset) ; in the recommended instructions set, this action would require 5 bytes (two for the long opcode, two for the long offset, and one for the stack-using instruction opcode). If the long-offset were a source offset into the vector of symbols and constants, the replacement would only require four bytes because *Push-Long-SC* is a short instruction. The

[7] Only pointers through the vector of symbols and constants may be written!

net effect of this optimization is to save one level of decoding for 63.3% of all instructions at no increase in code size for the current sample.

5. Adding New Instructions

5.1 Assembler Instructions With Implied Operands

In keeping with the concept of designing the instruction set of the compiler and assembler separately, it should be made clear that the following instructions are intended to be added to the assembler's instruction set -- *not* the compiler's.

Many of the most common instructions use particular operands frequently. In some instances, an additional byte of object code is generated to specify the operand of these instructions; substantial savings in code size may be obtained by adding new instructions to the instruction set which imply an operand without need of an additional operand byte. These more specific forms of a general instruction may be generated by an assembler with no added complexity in the main body of the compiler. It should be noted that this is a savings only in the number of bytes of object code, not in the number of instructions of object code. The optimizations discussed below, as a rule, include only those that would save at least 0.1% in code size per opcode. These optimizations alone would require about 80 of the 256 8-bit short instruction opcodes (about 31%). To save 0.1% in object code size, the addition of an opcode must save at least 2022 bytes if the estimated savings is based on the current object code sample. In the following tables listing operand statistics for various instructions, detailed operand information is shown only for those types of operands which occur frequently enough to allow effective optimization; other operand statistics are summarized.

5.1.1 Push Instructions

The *push* instruction occurs 320,383 times in 1,072,306 total instructions. These 320,383 instructions account for 641,678 bytes of object code, an average of slightly over 2.0 bytes per *Push* instruction. Table 5-1 shows the operand usage of the push instruction. The frequency of occurrence for particular operands is shown for short integer constants, symbols and constants, arguments and local variables, and special symbols. These statistics are in the form of *S.I.C.-number* pairs where *S.I.C.* is an integer constant and *number* is the number of times that integer appeared in the object code under study or *offset-number* pairs where *offset* is an index into whatever structure holds the type of operands being discussed and *number* is the number of occurrences of that offset. (For instance, offset 0 under the arguments and local variables section of the table refers to the first entry in the block of arguments and local variables on the control stack. Offset 0 under the symbols and constants section refers to the first entry in the vector of symbols and constants of the current function object.)

Several specialized instructions are suggested by analysis of these statistics. The most obvious new

Short Integer Constants

S.I.C.	Number	S.I.C.	Number	S.I.C.	Number	S.I.C.	Number		Total
								-128 to -1	760
0	0	1	13,221	2	5970	3	7280	0 to 3	26,471
4	5225	5	2600	6	1967	7	1771	4 to 7	11,563
8	2709	9	808	10	1006	11	1068	8 to 11	5591
12	792	13	576	14	480	15	630	12 to 15	2478
								16 to 19	5548
								20 to 23	3585
								24 to 27	3752
								28 to 97	2126
								98 to 127	85
								Total:	61,959

Symbols and Constants

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	2705	1	5121	2	9491	3	7670	0 to 3	24,987
4	6164	5	4497	6	3492	7	2431	4 to 7	16,584
8	2077	9	2272	10	1571	11	1298	8 to 11	7218
12	1080	13	1352	14	907	15	686	12 to 15	4025
								16 to 20	2872
								21 to 26	2398
								27 to 35	2326
								36 to 88	2045
								89 to 127	741
								Total:	63,196

Table 5-1: *Push* Statistics, part 1

instruction is one which would push only arguments-and-locals 0 through 3. This instruction would have the same sort of 6-bit opcode as the normal short instructions, but the 2-bit A field would be used to designate an index into the arguments and local variables block of the control stack. This is the same as adding 4 instructions with 8-bit opcodes which imply the operand. It is desirable to conceptualize these additions as 8-bit opcodes because this allows the addition of some number of new *Push* instructions which is not a multiple of four. Instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-3. The new push instructions are given names of the form *Push-XXn* where *XX* is a mnemonic abbreviation for short integer constants (SIC), symbols and constants (SC), arguments and local variables (AL), or Special Symbols (S). *n* is an integer offset. The addition of these instructions will save 262,714 bytes reducing the average number of bytes per instruction from 2.0 to 1.2.

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	33,456	1	34,567	2	24,278	3	18,269	0 to 3	110,570
4	13,110	5	9397	6	6196	7	5715	4 to 7	34,418
8	6252	9	4853	10	4175	11	3432	8 to 11	18,712
12	2225	13	1615	14	957	15	1120	12 to 15	5917
16	1310	17	1304	18	652	19	662	16 to 19	3928
								20 to 23	3946
								24 to 127	312
								Total :	177,803

Specials

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	4673	1	2829	2	1119	3	2176	0 to 3	10,797
4	1647	5	862	6	523	7	526	4 to 7	3558
								8 to 20	2112
								21 to 127	502
								Total :	16,969

Stack :	0
Extended Symbols & Constants :	456
Total Number of *Push* Instructions :	320,383

Table 5-2: *Push* Statistics, part 2

5.1.2 Push-Last Instructions

The *Push-Last* instruction occurs 68,804 times in 1,072,306 total instructions accounting for 116,539 bytes of object code, an average of 1.7 bytes per *Push-Last* instruction. Table 5-4 shows the operand usage of the *Push-Last* instruction.

Several specialized instructions are suggested upon analysis of these statistics. Instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-5. The new *Push-Last* instructions are given names of the form *Push-Last-ALn* where *AL* is a mnemonic abbreviation for arguments and local variables. *n* is an integer offset. These specialized *Push-Last* instructions will reduce the number of bytes per instruction from 1.6 to 1.3.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Push-SIC1	13,221	0.654
Push-SIC3	7280	0.360
Push-SIC2	5970	0.295
Push-SIC4	5225	0.258
Push-SIC8	2709	0.134
Push-SIC5	2600	0.129
Push-SIC18	2129	0.105
Push-SIC19	2057	0.102
Push-SC2	9491	0.469
Push-SC3	7670	0.379
Push-SC4	6164	0.305
Push-SC1	5121	0.253
Push-SC5	4497	0.222
Push-SC6	3492	0.173
Push-SC0	2705	0.134
Push-SC7	2431	0.120
Push-SC9	2272	0.112
Push-SC8	2077	0.103
Push-AL1	34,567	1.709
Push-AL0	33,456	1.654
Push-AL2	24,278	1.200
Push-AL3	18,269	0.903
Push-AL4	13,110	0.648
Push-AL5	9397	0.465
Push-AL8	6252	0.309
Push-AL6	6196	0.306
Push-AL7	5715	0.283
Push-AL9	4853	0.240
Push-AL10	4175	0.206
Push-AL11	3432	0.170
Push-AL12	2225	0.110
Push-S0	4673	0.231
Push-S1	2829	0.140
Push-S3	2176	0.108
Total Savings (34 opcodes)	262,714	12.987

Table 5-3: New *Push* Instructions

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	15,042	1	5743	2	2464	3	2701	0 to 3	25,950
4	882	5	1001	6	1353	7	656	4 to 7	3892
								8 to 17	2217
								18 to 127	1222
								Total :	33,281
								Stack :	21,069
								Short Integer Constants :	1462
								Symbols & Constants :	9365
								Specials :	3627
								Total Number of *Push-Last* Instructions :	68,804

Table 5-4: *Push-Last* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Push-Last-AL0	15,042	0.744
Push-Last-AL1	5743	0.284
Push-Last-AL3	2701	0.134
Push-Last-AL2	2464	0.122
Total Savings (4 opcodes)	25,950	1.283

Table 5-5: New *Push-Last* Instructions

5.1.3 Call Instructions

The *Call* instruction occurs 51,265 times in 1,072,306 total instructions accounting for 101,296 bytes of object code, for an average of 2.0 bytes per *Call* instruction. Table 5-6 shows the operand usage of the *Call* instruction.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-7. These new *Call* instructions are given names of the form *Call-SCn* where *SC* is a mnemonic abbreviation for symbols and constants. *n* is an integer offset. These specialized *Call* instructions will reduce the number of bytes per instruction from 2.0 to 1.3.

Symbols and Constants

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	10,442	1	5989	2	3097	3	3714	0 to 3	23,242
4	4512	5	2166	6	2664	7	2191	4 to 7	11,533
8	1608	9	1071	10	575	11	866	8 to 11	4120
								12 to 16	2384
								17 to 26	2104
								27 to 127	959
								Total :	44,342

Stack :	1234
Short Integer Constants :	0
Arguments & Local Variables :	5543
Specials :	146
Total Number of *Call* Instructions :	51,265

Table 5-6: *Call* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Call-SC0	10,442	0.516
Call-SC1	5989	0.296
Call-SC4	4512	0.223
Call-SC3	3714	0.184
Call-SC2	3097	0.153
Call-SC6	2664	0.132
Call-SC7	2191	0.108
Call-SC5	2166	0.107
Total Savings (8 opcodes)	34,775	1.719

Table 5-7: New *Call* Instructions

5.1.4 Pop Instructions

The *Pop* instruction occurs 79,593 times in 1,072,306 total instructions accounting for 137,217 bytes of object code, for an average of 1.7 bytes per *Pop* instruction. Table 5-8 shows the operand usage of the *Pop* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-9. These new *Pop* instructions are given names of the form *Pop-ALn* where *AL* is a

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	1248	1	7070	2	6696	3	7397	0 to 3	22,411
4	5489	5	4301	6	3600	7	4005	4 to 7	17,395
8	2994	9	1915	10	1473	11	1390	8 to 11	7772
12	1337	13	976	14	759	15	714	12 to 15	3786
								16 to 21	2662
								22 to 127	1752
								Total :	55,778
								Stack :	0
								Ignore :	21,969
								Symbols & Constants :	0
								Specials :	1846
								Total Number of *Pop* Instructions :	79,593

Table 5-8: *Pop* Statistics

mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These specialized *Pop* instructions will reduce the number of bytes per *Pop* instruction from 1.7 to 1.2.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Pop-AL3	7397	0.366
Pop-AL1	7070	0.349
Pop-AL2	6696	0.331
Pop-AL4	5489	0.271
Pop-AL5	4301	0.213
Pop-AL7	4005	0.198
Pop-AL6	3600	0.178
Pop-AL8	2994	0.148
Total Savings (8 opcodes)	41,552	2.054

Table 5-9: New *Pop* Instructions

5.1.5 Check Instructions

The *Check* instruction occurs 17,128 times in 1,072,306 total instructions accounting for 34,256 bytes of object code, for an average of 2.0 bytes per *Check* instruction. Table 5-10 shows the operand usage of the *Check* instruction. The frequency of occurrence for particular operands is shown for arguments and local variables and special symbols.

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	1893	1	1658	2	2470	3	1599	0 to 3	7620
4	2087	5	1029	6	809	7	1579	4 to 7	5504
8	498	9	508	10	215	11	147	8 to 11	1368
								12 to 127	764
								Total :	15,256

Stack :	0
Short Integer Constants :	0
Symbols & Constants :	0
Specials :	1872
Total Number of *Check* Instructions :	17,128

Table 5-10: *Check* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-11. These new *Check* instructions are given names of the form *Check-ALn* where *AL* is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These specialized *Check* instructions will reduce the number of bytes per *Check* instruction from 2.0 to 1.7.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Check-AL2	2470	0.122
Check-AL4	2087	0.103
Total Savings (2 opcodes)	4557	0.225

Table 5-11: New *Check* Instructions

5.1.6 Call-Maybe-Multiple Instructions

The *Call-Maybe-Multiple* instruction occurs 16,257 times in 1,072,306 total instructions accounting for 32,114 bytes of object code, for an average of 2.0 bytes per *Call-Maybe-Multiple* instruction. Table 5-12 shows the operand usage of the *Call-Maybe-Multiple* instruction. The frequency of occurrence for particular operands is shown only for symbols and constants.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-13. These new *Call-Maybe-Multiple* instructions are given names of the form *Call-Maybe-Multiple*-SC*n* where SC is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These new *Call-Maybe-Multiple* instructions will reduce the number of bytes per *Call-Maybe-Multiple* instruction from 2.0 to 1.3.

Symbols and Constants

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	2672	1	5216	2	1062	3	2528	0 to 3	11,478
4	1260	5	472	6	525	7	485	4 to 7	2742
								8 to 127	1377
								Total :	15,597
								Stack :	400
								Short Integer Constants :	0
								Arguments & Local Variables :	235
								Specials :	25
								Total Number of *Call-Maybe-Multiple* Instructions :	16,257

Table 5-12: *Call-Maybe-Multiple* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Call-Maybe-Multiple-SC1	5216	0.258
Call-Maybe-Multiple-SC0	2672	0.132
Call-Maybe-Multiple-SC3	2528	0.125
Total Savings (3 opcodes)	10,416	0.515

Table 5-13: New *Call-Maybe-Multiple* Instructions

5.1.7 Cdr Instructions

The *Cdr* instruction occurs 8758 times in 1,072,306 total instructions accounting for 16,234 bytes of object code for an average of 1.9 bytes per *Cdr* instruction. Table 5-14 shows the operand usage of the *Cdr* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-15. These new *Cdr* instructions are given names of the form *Cdr*-AL*n* where AL is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These specialized *Cdr* instructions will reduce the number of bytes per *Cdr* instruction from 1.9 to 1.5.

5.1.8 Car Instructions

The *Car* instruction occurs 19,894 times in 1,072,306 total instructions accounting for 39,531 bytes of object code, for an average of 2.0 bytes per *Car* instruction. Table 5-16 shows the operand usage of the *Car* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	2704	1	775	2	609	3	540	0 to 3	4628
4	219	5	473	6	254	7	431	4 to 7	1377
								8 to 127	856
								Total :	6861
								Stack :	1282
								Short Integer Constants :	0
								Symbols & Constants :	580
								Specials :	35
								Total Number of *Cdr* Instructions :	8758

Table 5-14: *Cdr* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Cdr-AL0	2704	0.134
Total Savings (1 opcode)	2704	0.134

Table 5-15: New *Cdr* Instructions

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	3098	1	2714	2	3360	3	2035	0 to 3	11,207
4	866	5	864	6	584	7	468	4 to 7	2782
								8 to 127	1855
								Total :	15,844
								Stack :	257
								Short Integer Constants :	0
								Symbols & Constants :	580
								Specials :	3213
								Total Number of *Car* Instructions :	19,894

Table 5-16: *Car* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-17. These new *Car* instructions are given names of the form *Car*-AL*n* where AL is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These new *Car* instructions will reduce the number of bytes per *Car* instruction from 2.0 to 1.4.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Car-AL2	3360	0.166
Car-AL0	3098	0.153
Car-AL1	2714	0.134
Car-AL3	2035	0.101
Total Savings (4 opcodes)	11,207	0.554

Table 5-17: New *Car* Instructions

5.1.9 Cadr Instructions

The *Cadr* instruction occurs 9238 times in 1,072,306 total instructions accounting for 16,156 bytes of object code, for an average of 1.7 bytes per *Cadr* instruction. Table 5-18 shows the operand usage of the *Cadr* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	3508	1	109	2	476	3	153	0 to 3	4246
4	122	5	0	6	50	7	0	4 to 7	172
								8 to 127	25
								Total :	4443
								Stack :	2320
								Short Integer Constants :	0
								Symbols & Constants :	0
								Specials :	2475
								Total Number of *Cadr* Instructions :	9238

Table 5-18: *Cadr* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-19. These new *Cadr* instructions are given names of the form *Cadr*-AL*n* where AL is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These new *Cadr* instructions will reduce the number of bytes per *Cadr* instruction from 1.7 to 1.4.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Cadr-AL0	3508	0.173
Total Savings (1 opcode)	3508	0.173

Table 5-19: New *Cadr* Instructions

5.1.10 = Instructions

The = instruction occurs 28,964 times in 1,072,306 total instructions accounting for 56,723 bytes of object code, for an average of 1.8 bytes per = instruction. Table 5-20 shows the operand usage of the = instruction.

Short Integer Constants

S.I.C.	Number	S.I.C.	Number	S.I.C.	Number	S.I.C.	Number		Total
								-128 to -1	34
0	1370	1	2101	2	3423	3	2545	0 to 3	9439
4	2958	5	1606	6	532	7	2377	4 to 7	7473
8	269	9	1104	10	1032	11	1213	8 to 11	3618
								12 to 127	238
								Total :	20,802
								Stack :	1243
								Symbols & Constants :	706
								Arguments & Local Variables :	6169
								Extended Symbols & Constants :	19
								Specials :	25
								Total Number of = Instructions :	28,964

Table 5-20: = Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-21. These new = instructions are given names of the form =-SIC*n* where SIC is a mnemonic abbreviation for short integer constant operands and *n* is a short integer constant. These new = instructions will reduce the number of bytes per = instruction from 1.8 to 1.5.

5.1.11 Eq Instructions

The *Eq* instruction occurs 18,396 times in 1,072,306 total instructions accounting for 36,067 bytes of object code, for an average of 2.0 bytes per *Eq* instruction. Table 5-22 shows the operand usage of the *Eq* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
=-SIC2	3423	0.169
=-SIC4	2958	0.146
=-SIC3	2545	0.126
=-SIC7	2377	0.118
=-SIC1	2101	0.104
Total Savings (5 opcodes)	13,404	0.663

Table 5-21: New = Instructions

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	354	1	1162	2	928	3	2943	0 to 3	5387
								4 to 6	3319
								7 to 16	2206
								17 to 127	0
								Total :	10,912
								Stack :	725
								Short Integer Constants :	103
								Symbols & Constants :	6001
								Specials :	655
								Total Number of *Eq* Instructions :	18,396

Table 5-22: *Eq* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-23. These new *Eq* instructions are given names of the form *Eq*-AL*n* where AL is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These new *Eq* instructions will reduce the number of bytes per *Eq* instruction from 2.0 to 1.8.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Eq-AL3	2943	0.145
Total Savings (1 opcode)	2943	0.145

Table 5-23: New *Eq* Instructions

5.1.12 Bind-Pop Instructions

The *Bind-Pop* instruction occurs 6178 times in 1,072,306 total instructions accounting for 12,156 bytes of object code, for an average of 2.0 bytes per *Bind-Pop* instruction. Table 5-24 shows the operand usage of the *Bind-Pop* instruction.

Symbols and Constants

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	2684	1	932	2	497	3	444	0 to 3	4557
4	441	5	255	6	158	7	61	4 to 7	915
								8 to 127	431
								Total :	5903
								Stack :	200
								Short Integer Constants :	0
								Arguments & Local Variables :	75
								Specials :	0
								Total Number of *Bind-Pop* Instructions :	6178

Table 5-24: *Bind-Pop* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-25. These new *Bind-Pop* instructions will reduce the number of bytes per *Bind-Pop* instruction from 2.0 to 1.5.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Bind-Pop-SC0	2684	0.133
Total Savings (1 opcode)	2684	0.133

Table 5-25: New *Bind-Pop* Instructions

5.1.13 Unbind Instructions

The *Unbind* instruction occurs 3732 times in 1,072,306 total instructions accounting for 7464 bytes of object code, for an average of 2.0 bytes per *Unbind* instruction. Table 5-26 shows the operand usage of the *Unbind* instruction.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-27. These new *Unbind* instructions will reduce the number of bytes per *Unbind* instruction from 2.0 to 1.1.

Short Integer Constants

wS.I.C.	Number	S.I.C.	Number	S.I.C.	Number	S.I.C.	Number		Total
								-128 to -1	0
0	0	1	3424	2	174	3	58	0 to 3	3656
4	25	5	0	6	0	7	0	4 to 7	25
								8 to 127	26
								Total :	3707

Stack :	0
Symbols & Constants :	0
Arguments & Local Variables :	25
Specials :	0
Total Number of *Unbind* Instructions :	3732

Table 5-26: *Unbind* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Unbind-SIC1	3424	0.169
Total Savings (1 opcode)	3424	0.169

Table 5-27: New *Unbind* Instructions

5.1.14 List Instructions

The *List* instruction occurs 12,979 times in 1,072,306 total instructions accounting for 25,780 bytes of object code, for an average of 2.0 bytes per *List* instruction. Table 5-28 shows the operand usage of the *List* instruction.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-29. These new *List* instructions will reduce the number of bytes per *List* instruction from 2.0 to 1.1.

5.1.15 List* Instructions

The *List** instruction occurs 4242 times in 1,072,306 total instructions accounting for 8484 bytes of object code, for an average of 2.0 bytes per *List** instruction. Table 5-30 shows the operand usage of the *List** instruction.

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-31. These new *List** instructions will reduce the number of bytes per *List** instruction from 2.0 to 1.4.

Short Integer Constants

S.I.C.	Number	S.I.C.	Number	S.I.C.	Number	S.I.C.	Number		Total
								-128 to -1	0
0	0	1	3335	2	4880	3	2707	0 to 3	10,922
4	1249	5	600	6	0	7	2	4 to 7	1851
								8 to 127	28
								Total :	12,801

Stack :	178
Symbols & Constants :	0
Arguments & Local Variables :	0
Specials :	0
Total Number of *List* Instructions :	12,979

Table 5-28: *List* Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
List-SIC2	4880	0.241
List-SIC1	3335	0.165
List-SIC3	2707	0.134
Total Savings (3 opcodes)	10,922	0.54

Table 5-29: New *List* Instructions

Short Integer Constants

S.I.C.	Number	S.I.C.	Number	S.I.C.	Number	S.I.C.	Number		Total
								-128 to -1	0
0	0	1	1157	2	2376	3	476	0 to 3	4009
4	80	5	70	6	55	7	2	4 to 7	207
								8 to 127	26
								Total :	4242

Stack :	0
Symbols & Constants :	0
Arguments & Local Variables :	0
Specials :	0
Total Number of *List Instructions :**	4242

Table 5-30: *List** Statistics

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
List*-SIC2	2376	0.117
Total Savings (1 opcode)	2376	0.117

Table 5-31: New *List** Instructions

5.1.16 Long-Escape Instructions

The *Long-Escape* instruction occurs 96,405 times in 1,072,306 total instructions accounting for 211,611 bytes of object code, for an average of 2.2 bytes per *Long-Escape* instruction. Table 5-32 shows the operand usage of the *Long-Escape* instruction. The frequency of occurrence for particular operands is shown only for arguments and local variables.

Arguments and Local Variables

Offset	Number	Offset	Number	Offset	Number	Offset	Number		Total
0	42	1	1625	2	3118	3	3472	0 to 3	8257
4	3060	5	1836	6	1095	7	720	4 to 7	6711
8	493	9	723	10	442	11	458	8 to 11	2116
								12 to 127	1164
								Total :	18,248
								Stack :	50,583
								Ignore :	27,021
								Symbols & Constants :	0
								Specials :	553
								Total Number of *Misc* Instructions :	96,405

Table 5-32: *Misc* Statistics

Specialized instructions whose addition would save more than the threshold number of bytes (2022) are listed in Table 5-33. These new *Long-Escape* instructions are given names of the form *Long-Escape*-AL*n* where AL is a mnemonic abbreviation for arguments and local variables and *n* is an integer offset. These new *Long-Escape* instructions will reduce the number of bytes per *Long-Escape* instruction from 2.2 to 2.1.

New Instruction	Number of Bytes Eliminated	% of Code Eliminated
Misc-AL3	3472	0.172
Misc-AL2	3118	0.154
Misc-AL4	3060	0.151
Total Savings (3 opcodes)	9650	0.477

Table 5-33: New *Misc* Instructions

5.1.17 Specialized Instruction Summary

In the preceding sections, we have seen how object code size may be reduced by adding 80 specialized instructions. These specialized instructions eliminate 442,786 bytes of code. The instructions in the preceding sections accounted for 1,493,306 bytes of code in 762,216 occurrences for an average of 2.0 bytes per instruction. Eliminating 442,786 bytes brings this average for the optimized instructions down to 1.4. Thus the addition of these 80 specialized instructions reduces code size by 21.9%.

5.2 Converting Long instructions to Short Instructions

Several of the instructions which were originally defined to be long instructions occur frequently enough to warrant their inclusion as short instructions in the new assembler instruction set. This re-alignment will result in a 1-byte saving per occurrence of a converted long instruction. The long instructions which should be converted are listed in Table 5-34. It is interesting to note that these 11 long instructions account for 62,263 (64.6) out of 96,405 total long instruction occurrences. Thus less than 36% of the existing long instructions occurrences will remain once these are converted to short instructions.

Long Instruction	Number of Bytes Eliminated	% of Code Eliminated
(V-Store Ignore)	17,294	0.855
(V-Access Stack)	13,476	0.666
(Get-Definition Stack)	7871	0.389
(Type Stack)	4680	0.231
(Type A&L)	3644	0.180
(V-Access A&L)	3078	0.152
(V-Access Ignore)	2915	0.144
(Make-Immediate-Type Stack)	2907	0.144
(Cons Stack)	2510	0.124
(Cons A&L)	1966	0.097
(Typed-V-Store Ignore)	1922	0.095
Total (11 opcodes)	**62,263**	**3.078**

Table 5-34: New Short instructions from Old Long instructions

5.3 Non-Indicator Branch Instructions

The addition of a branch instruction which branches based upon the value of an operand specified by the branch instruction itself rather than an indicator setting can reduce code size by a significant amount. Table 5-35 shows pairs of instructions generated by the current compiler in which the first instruction does nothing but set the indicators for a branch, which is specified by the second instruction of the pair.

Instruction Pair	Number of Occurrences
Check, Branch-Not-Atom	3611
<, Branch-Null	3390
<, Branch-Not-Null	2425
>, Branch-Null	2163
Check, Branch-Atom	1636
	13,225

Table 5-35: Branch Instruction Pairs

However, only one new branch instruction is justified based upon these instruction pair occurrences. These instruction pair occurrences are misleading because they do not specify where the operand for the first instruction comes from. In reality, each instruction of the pairs in this table represent four opcodes.[8] In order to calculate the expected savings for a new branch instruction, we must determine what proportion of each instruction' occurrences use a particular operand source. For example, using the *Check, Branch-Not-Atom* pair, 89% of *Check* instructions use an argument or local variable operand. 98.5% of *Branch-Not-Atom* instructions are short and do not pop the stack. Therefore, (3611 * .89 * .985) = 3166 (0.157%) is the expected savings due to adding an *NI-Branch-Not-Atom-AL* instruction. The addition of no other branch instruction saves greater than 0.1% in code size.

5.4 Compiler Instructions to Replace Instruction Pairs

New instructions are suggested if particular instruction pairs occurred frequently (ie. combine the two into one instruction). It should be noted that these optimizations are of a different character than those which add specialized instructions with implied operands. These optimizations would most conveniently be added by changing the main body of the compiler such that these were generated under the appropriate circumstances rather than contorting the assembler to reduce some instruction pairs to one instruction.

One instruction pair that might be combined into a single instruction is *Branch-If-Arg-Supplied, Set-Null.*

[8] Four 8-bit opcodes are represented by the first to account for the A-code which specifies the operand source; four 8-bit opcodes are represented by the branch to account for the short-long, pop-nopop options.

By combining these two instructions to arrive at a new instruction, code size may be reduced. A diagram of the byte layout for this instruction pair is presented in Figure 5-1.

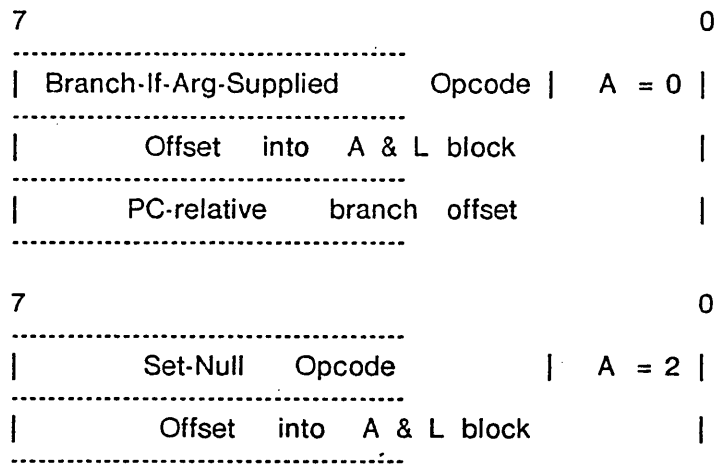


Figure 5-1: Byte Configuration for *Branch-If-Arg-Supplied, Set-Null* Pair

Notice that the operand of the Set-Null instruction is *always* an offset into the arguments and local variables block (A = 2); and the Branch-If-Arg-Supplied Instruction is *always* takes a short (1 byte) PC-relative offset and never pops the stack (A = 0). For more detailed information about these instructions, see Appendix I.

The following optimizations are possible:

1. One byte per occurrence of this instruction pair could be saved by simply adding a new 1-byte opcode that used all the same operands as the instruction pair because one of the two opcode bytes could be eliminated. This would eliminate (1 * 1782) = 1782 bytes for a .088% reduction in code size. It is not necessary to introduce four 1-byte opcodes to correspond to the 4 possible Set-Null opcodes; only one of the A-code values is in use in this pairing -- the arguments and locals variables combination. Therefore the cost of this optimization is only one opcode.

2. Further savings would result from eliminating one operand byte altogether. This is possible because the *Set-Null* instruction of the pair certainly refers to the same argument as the *Branch-If-Arg-Supplied* instruction in the arguments and local variables part of the stack frame in order to initialize it. We need only specify one offset for both references. So, (2 * 1782) = 3564 bytes could be eliminated from existing compiled code by adding a new instruction *Set-Null-Unless-Arg-Supplied* were added to the instruction set. This is a .176% reduction in code size.

3. Further savings may be possible here. Depending upon how the compiler generates code for initializing unsupplied arguments, even the branch PC-relative offset may be eliminated. If the branch is used only to bypass the Set-Null instruction, then no branching is needed when the new

instruction is used. This would eliminate (3 * 1782) = 5346 bytes resulting a 0.264% reduction in code size.

It would seem that the most reasonable choice is number 3 above; the resulting byte layout for the new instruction would be as in Figure 5-2.

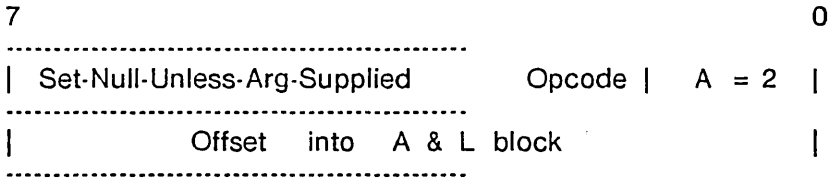


Figure 5-2: Byte Layout for Set-Null-Unless-Arg-Supplied Instruction

Another instruction pair which is easily converted into a single instruction is *Branch-Null, Return.* The new instruction resulting from combining this pair would be *(Return-Unless-Null Stack. Return Stack* accounts for 79.7% of all *Return* occurrences, and this pair occurs 5722 times; so this would result in a savings of (5722 * .797) = 4561 bytes of code, a 0.225% reduction in code size. Adding other operand sources for this instruction do not save the minimum 0.1% in code size required.

Another instruction pair that may be converted to a single instruction is the pair *Car, Scdr.* From the nature of these instructions we may assume that both use the same operand. The new instruction resulting from this pair would be *(Push-Car-Scdr-ALI).* Arguments and local variables are used as the operand in 88.7% of all *Scdr* instructions. This pair occurs 3670 times; so the expected savings is (3670 * .887) = 3255 bytes, a 0.161% reduction in code size.

5.5 Summary of New Opcodes

From the preceding sections we see that with the addition of 95 opcodes to the assembler's instruction set, a 25.8% savings in object code size may be obtained. This is a reduction from 2,022,694 bytes of object code to 1,501,277 bytes, a savings of 521,417 bytes. These new instructions will reduce the average number of bytes per instruction from 1.9 to 1.4.

6. Deleting Instructions

In order to add the specialized instructions recommended in other sections of this paper, opcode space must be available in the instruction set of the assembler. In this section we explore options for making opcode space for short instructions available. These options include:

- converting some short instructions to long instructions
- eliminating illegal instruction-operand combinations from the opcode space
- eliminating some rarely-used instructions by replacing them with sequences of more common instructions

In no case does any change intended to open up opcode space affect any code generation except that done by the assembler (ie. the compiler is unchanged).

6.1 Illegal Non-Branch Short Instruction Opcodes

56 non-branch short instructions are presently defined, accounting for 224 opcodes. Of these 224 opcodes 20 are illegal and can never be generated by the assembler. These 20 re-claimable opcodes are described in Table 6-1. The name of the instruction is listed following its 6-bit instruction opcode . This 6-bit opcode, concatenated with the 2-bit A-field (which describes the operand source), make up a unique 8-bit opcode which fills the first byte of every object-code instruction. All of these illegal opcodes occur when A is 1. When A is 1 and the operand is used only as a source, then the operand is a short integer constant taken from the next byte of object code. If the operand is used as a destination, the result is ignored when A is 1 (however, it is used to set the indicators). For a description of these instructions, see Appendix I.

6-Bit Opcode	Instruction	6-bit Opcode	Instruction
1	*Call*	21	*Caar*
2	*Call-0*	22	*Scdr*
3	*Call-Multiple*	23	*Scddr*
4	*Call-Maybe-Multiple*	37	*1+*
16	*Car*	38	*1-*
17	*Cdr*	39	*Bind-Null*
18	*Cadr*	40	*Bind-T*
19	*Cddr*	41	*Bind-Pop*
20	*Cdar*	50	*Set-Lpush*
51	*Set-Lpop*	54	*Spread*

Table 6-1: Illegal Instructions when A = 1

6.2 Little-Used Non-Branch Short Instruction Opcodes

Many of the legal short instructions have not been generated by the current Spice Lisp compiler. While this might be attributed to a poor selection of instructions by the compiler, it seems more likely (after analysis of the unused instructions) that they are simply not useful. In this section, it is proposed that many of these opcodes be re-allocated to more useful instructions.

Several opcodes which should be considered for re-use are described below. With each description is included the instruction or sequence of instructions which could be generated by the assembler to replace the instruction being described.

2 Call-0, A = 0 or 3

No function of 0 arguments has yet been called using the stack or a special symbol as the source of the function in existing code. While these combinations may certainly occur in future code, there is no need to allocate two opcodes to them. They may be handled in the assembler by generating a *Call* and a *Push-Last*. >>>>>NOTE: Slguts is unclear about this. Push-Last must not try to cross control stack boundaries if this is checked for. It should simply start the call without munging the stack when given stack as argument.<<<<<

8 Push, A = 0 This is a no-op unless it is being used to set the indicators. Several instructions perform this same operation; they should be mapped onto the same opcode. These instructions include *Pop* with A = 0 and *Copy* with A = 1. Moreover, none of these combinations has yet occurred; therefore, this might economically be converted a long instruction.

10 Push-Under, A = 1-3

Push-Under has never been generated by with these operands using the current compiler. These combinations may be eliminated both from the set of possible instructions generated by the compiler and from the opcode set generated by the assembler.

11 Check, A = 0 This instruction has the same effect as *Pop* with A = 1 in the current instruction set. This is not intuitive. In the revised instruction set this combination should *not* pop the stack; it should have the same effect as *Push* with A = 0 as above.

12 Pop, A = 0 See *Push* above.

13 Copy, A = 1 See *Push* above for a description of the effect of this instruction in the current instruction set. However, this is simply an unintuitive and redundant combination -- the compiler should never generate it.

50 Set-Lpush, A = 0,2,3

This instruction has never been generated. It may be replaced by *(Push x), (Cons Stack)*[9], *(Pop x).* The *Push* and *Pop* instructions would have the same operand as the original *Set-Lpush* would have had. While intended for use when compiling Spice Lisp DO's, this instruction is not used and does not seem an economical use of the opcode space -- consing will dominate the cost of the instruction fetch here anyway. If not totally eliminated from the instruction set, this should at least become a long instruction.

51 Set-Lpop, A = 0,2,3

This instruction has not been used. It may be deleted in much the same way as *Set-Lpush* above (in either the assembler or both the compiler and assembler). It would be replaced by *(Car x), (Cdr x), (Pop x).* This instruction was intended to be used when compiling Spice Lisp DO's; however, this instruction might be used if the compiler were smarter about compiling DO's; therefore, this instruction should be retained.

From above, we see that 12 opcodes may be re-claimed with no apparent cost. If the *Set-Lpush* instruction is retained as a long instruction, the cost is only three long instruction opcodes.

While several opcodes would be freed by the preceding suggestions, many more could be eliminated at little cost in added code size to the current sample. These eliminations are all of the same sort. All of the instructions in the existing instruction set are encoded to use operands from any of four sources (therefore 4 8-bit opcodes). Many of these instructions are never used in conjunction with one or more of the possible operand sources for which opcode space is used. It would be a simple matter for the assembler to translate these instructions so as to use the stack as operand. It would generate a *Push* instruction to precede each of these instructions when an uncommon source operand was specified by the compiler. When an uncommon destination operand was specified, a *Pop* instruction would follow the original instructions. If this were done, these rarely used opcodes could be re-assigned to frequently occurring, specialized instructions. The likely candidates for this re-assignment are listed in Table 6-2. Each of these would cost one byte of increased code size per occurrence to eliminate. These instructions account for 53 opcodes (out of 256 possible in 1 byte) and occur only 4688 times (out of 1,072,306 total instructions). Thus they take up 20.7% of the available opcode space but account for only 0.44% of the instructions present in the current sample. 23 of the opcodes in table 6-2 do not appear in the code sample and may thus be eliminated with no resulting increase in code size. Eliminating all 53 of these opcodes would increase code size by 0.23%.

---

[9]The *(Cons A B)* long instruction expects *A* and *B* to be on the stack. *A* would already be on the stack (see the definition of *Set-Lpush* in I).

Instruction and Operand	Occurrences	Instruction and Operand	Occurrences
(Call-Multiple A&L)	4	*(Bit-Or S&C -- Special)*	28
(Call-Maybe-Multiple A&L)	235	*(Eql Short-Constant)*	0
(Return Ignore)	133	*(Eql A&L)*	76
(Throw Short-Constant)	0	*(Eql S&C -- Special)*	0
(Throw A&L)	122	*(> S&C -- Special)*	217
(Throw S&C -- Special)	368	*(Eq Short-Constant)*	103
(Check Short-Constant)	0	*(1+ S&C -- Special)*	387
(Make-Predicate A&L)	141	*(1- S&C -- Special)*	0
(Make-Predicate S&C -- Special)	0	*(Not-Predicate S&C -- Special)*	0
(Not-Predicate A&L)	99	*(Bind-Null A&L)*	25
(Cadr A&L)	0	*(Bind-T A&L)*	0
(Cddr S&C -- Special)	0	*(Bind-T S&C -- Special)*	12
(Cdar A&L)	0	*(Bind-Pop A&L)*	75
(Cdar S&C -- Special)	0	*(Set-0 Ignore)*	0
(Caar S&C -- Special)	105	*(Set-0 S&C -- Special)*	72
(Trunc S&C -- Special)	16	*(Set-T S&C -- Special)*	229
(+ S&C -- Special)	277	*(Npop A&L)*	0
(- S&C -- Special)	321	*(Npop S&C -- Special)*	0
(S&C -- Special)*	305	*(Unbind A&L)*	25
(/ Short-Constant)	206	*(Unbind S&C -- Special)*	0
(/ S&C -- Special)	14	*(Set-Lpop S&C -- Special)*	0
(Bit-And Short-Constant)	316	*(List A&L)*	0
(Bit-And A&L)	307	*(List S&C -- Special)*	0
(Bit-Xor A&L)	151	*(List* A&L)*	0
(Bit-Xor S&C -- Special)	0	*(List* S&C -- Special)*	0
(Bit-Or Short-Constant)	0	*(Spread S&C --Special)*	72
(Bit-Or A&L)	247	**Total** (53 Opcodes)	4688

Table 6-2: Rarely Used Instructions and Operands

6.3 Little-Used Branch Opcodes

Several of the branch instruction opcodes are never or rarely used. These may be eliminated in order to open up opcode space for more frequently occurring operations. Existing branch instructions have a format similar to that of existing non-branch short instructions. A 6-bit opcode is used to denote what sort of branch is to be executed. A 2-bit A-field is used to represent whether the branch is short or long and whether to pop the stack or not if the branch is taken. For a complete description of existing branch instructions, see Appendix I. Many of the 8-bit combinations possible in the present encoding are unneeded in the assembler's instruction set. All of the short branches (1-byte PC-relative offset) which do not pop the stack are used enough to warrant their inclusion. Those which may be eliminated are described below. Rather

than introduce the confusing A-code encoding of the pop -- no-pop and short -- long offset possibilities, the instructions below will be described in the following form : (branch-mnemonic *X Y*) where *X* is either *long* or *short* and *Y* is either *pop* or *nopop*.

- *(Branch X Pop)*
- *(Branch-If-Arg-Supplied X Pop)*
- *(Branch-Atom X Pop)*
- *(Branch-Not-Atom X Pop)*
- *(Branch-Zero X Pop)*
  Few branches with a pop if the branch is not taken have been generated. 10 opcodes may be freed if we eliminate all but the 6 most frequent types of branch instructions which pop if the branch is not taken. Those eliminated may be replaced by the same sort of branch instruction followed by an explicit *Pop*; thus the stack would be popped only if the branch was not taken, as specified for the compiler instruction set. This will not increase the current object code size.
- *(Branch-If-Arg-Supplied long nopop)*
  This combination has never occurred. It may be eliminated or converted to a long instruction with no increase in current object code size. While elimination of this long offset form of the *Branch-If-Arg-Supplied* instruction might restrict the compiler when it is generating code to handle optional arguments in Spice Lisp, it seems doubtful that more bytes of code than can be specified in one byte of offset will ever be needed to initialize an unsupplied argument. Nevertheless, since plenty of opcode space is available for long instructions, the extended form should be retained as a long instruction -- just in case!
- *(Branch-Null Long Pop)*
- *(Branch-Not-Null Long Pop)*
- *(Branch-Atom Long NoPop)*
- *(Branch-Not-Atom Long NoPop)*
- *(Branch-Zero Long NoPop)*
- *(Branch-Not-Zero Long NoPop)*
  These branch instructions occur so seldom that they could be eliminated or converted to long instructions. Since the two instructions in this group which pop the stack if the branch is not taken may be easily simulated by the assembler with an instruction sequence requiring no more

bytes to express than a long instruction which would have the same effect, they may be eliminated entirely. The other three instructions should be retained for flexibility, but converted to long instructions if changed at all. The cost of converting these to be long instructions (ie. use an escape code) is given in table 6-3 in section 6.4.

Instruction and Operand	Number of Occurrences	Cost in Bytes
(Branch-If-Arg-Supplied Long NoPop)	0	0
(Branch-Null Long Pop)	193	193
(Branch-Not-Null Long Pop)	18	18
(Branch-Atom Long NoPop)	231	231
(Branch-Not-Atom Long NoPop)	66	66
(Branch-Zero Long NoPop)	48	48
(Branch-Not-Zero Long NoPop)	0	0
Total (7 Opcodes)	556	556

Table 6-3: Conversion of Short Branch Instructions to Long Branch Instructions

6.4 Converting Short Instructions to Long Instructions

Several other opcodes may be reclaimed by converting them to long instructions. These are listed in Table 6-4. These are in addition to the long instructions added above. Calculating the cost of converting some of these instructions to long instructions is not straightforward. If we convert only these to long instructions and eliminate none of the above instructions (see section {unused-macrops}) which depend on the presence of a similar instruction with stack as operand, then the cost is simply one byte per occurrence of the converted instruction. However, if we both delete those above instructions and convert these to long instructions, then we must also add one byte per occurrence of the deleted instruction above. This total cost is listed in the third column of the table.

These instructions account for 17 opcodes (out of 256 possible in 1 byte) and occur 1470 times (out of 1,072,306 total instructions). Thus they take up 9.0% of the available opcode space but account for only 0.14% of the instructions present in the current sample. Five of the opcodes in this table do not appear in the code sample and may thus be converted with no resulting increase in code size. All 17 may be converted at the cost of a 0.22% increase (2366 bytes) in code size; this cost estimate assumes that the relevant instructions from table 6-2 are being eliminated (otherwise the cost of converting these to long instructions would be lower).

Instruction and Operand	Number of Occurrences	Cost in Total Bytes
(Throw Stack)	0	490
(Push-Under Stack)	201	201
(Check Stack)	0	0
(Copy Stack)	0	0
(Car Stack)	257	257
(Caar Stack)	120	225
(Scdr Stack)	0	0
(Scddr Stack)	0	0
(Trunc Stack)	93	109
(Eql Stack)	118	194
(Bind-Null Stack)	150	175
(Bind-T Stack)	25	37
(Bind-Pop Stack)	200	275
(Unbind Stack)	0	25
(List Stack)	178	178
(List Stack)*	0	0
(Spread Stack)	128	200
Total (17 Opcodes)	1470	2366

Table 6-4: Conversion of Short Instructions to Long Instructions

6.5 Summary of Available Opcode Space

A large portion of the opcode space for 1-byte opcodes is available for re-assignment if the above changes are made. 16 opcodes are presently unassigned.[10] From section 6.1 we see that 20 opcodes are assigned but illegal. From section 6.2 we see that 65 opcodes may be reclaimed: 35 opcodes at no cost and 30 at the cost of 4688 bytes of extra code. From section 6.3 we see that 17 opcodes may be reclaimed: 10 opcodes at no cost, 2 at the cost of 2 long instructions, and 5 at the cost of 3 long instructions and 556 bytes of additional object code. From section 6.4 we see that 17 opcodes may be reclaimed: 7 at the cost of 7 long instructions, and 10 at the cost of 2366 bytes of extra code and 10 long instructions.[11] The total number of opcodes which may be saved is 135 out of 256, or 52.7%. The cost of saving all 135 of these is 22 long instructions and 7610 bytes of additional code. This cost amounts to 2.1% of the long instruction opcode space (1024 long opcodes) and a 0.38% increase in object code size. However, only 95 new opcodes are needed for the new short instructions recommended in sections 5 and 4. We see from the preceding sections on deleting instructions that 81

[10] 6-bit opcodes 0 and 45-47 are unused; each has a 2-bit A-field.

[11] The above figures imply that 90 opcodes, 81 of which are assigned, have never been generated! These 81 opcodes make up 33.7% of the 240 assigned opcodes. 35.2% of the opcode space (90 out of 256) has been essentially unused.

opcodes are available at no cost in additional long instructions or added object code length. Of course, the compiler could generate instructions in the future which would require extra code bytes to express even if only these 81 were reclaimed. The additional 14 opcodes should be reclaimed in such a way as to incur as little cost as possible. We note that 9 more opcodes may be reclaimed at the cost of adding 9 Long instructions to the instruction set, but incur no increase in object code size (given the current sample). The other 5 may be reclaimed at the cost of 1 long instruction and a 64 byte increase in object code size. The remaining 40 may be reclaimed upon further analysis as necessary.

To summarize, the 95 needed opcodes may be reclaimed at the cost of 10 additional long instructions and 64 bytes of additional object code[12].

[12] From 2,022,694 to 2,022,758 bytes before optimization!

7. Conclusions

A re-alignment of the operand source field is described in Section 4. This re-alignment should save one level of decoding for each occurrence of an argument and local variables operand.

In Section 5 it is shown that 25.8% of existing code may be eliminated with the addition of 95 new instructions to the instruction set generated by the assembler. This is accomplished, by adding only those opcodes which would save at least 0.1% of existing code. We see from Section 6 that 81 opcodes are available at no cost (given the current code sample) and 135 are available at minimal cost. Reusing only 96 of these 135 opcodes leaves 40 open for future use. Some of these might be allocated for use as additional escape codes for long instructions. The 95 new instructions are listed in Table 7-1 in order of the savings expected due to their inclusion in the instruction set. After all changes to the instruction set are made, the object code size is reduced from 2,022,694 to 1,501,341; a 25.8% decrease.

The new assembler instruction set is described in Appendix II. Only those instructions whose elimination would cost nothing have been deleted. No short instructions have been converted to long instructions; although this is a prime area for future optimization. The new compiler instruction set is described in Appendix III.

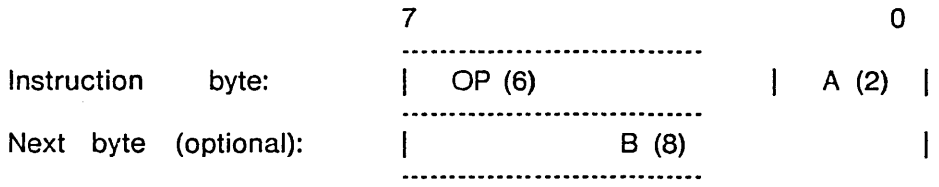
New Instruction	Bytes Saved	Percent Savings	New Instruction	Bytes Saved	Percent Savings
Push-AL1	34,567	1.709	*Unbind-SIC1*	3424	0.169
Push-AL0	33,456	1.654	*Car-AL2*	3360	0.166
Push-AL2	24,278	1.200	*List-SIC1*	3335	0.165
Push-AL3	18,269	0.903	*Push-Car-Scdr-AL1*	3255	0.161
V-Store-Ignore	17,294	0.855	*NI-Branch-Not-Atom-AL*	3166	0.157
Push-Last-AL0	15,042	0.744	*Misc-AL2*	3118	0.154
V-Access-Stack	13,476	0.666	*Call-SC2*	3097	0.153
Push-SIC1	13,221	0.654	*Car-AL0*	3098	0.153
Push-AL4	13,110	0.648	*V-Access-AL*	3078	0.152
Call-SC0	10,442	0.516	*Misc-AL4*	3060	0.151
Push-SC2	9491	0.469	*Pop-AL8*	2994	0.148
Push-AL5	9397	0.465	*=-SIC4*	2958	0.146
Get-Definition-Stack	7871	0.389	*Eq-AL3*	2943	0.145
Push-SC3	7670	0.379	*Make-Immediate-Type-Stack*	2907	0.144
Pop-AL3	7397	0.366	*V-Access-Ignore*	2915	0.144
Push-SIC3	7280	0.360	*Push-S1*	2829	0.140
Pop-AL1	7070	0.349	*Push-SIC8*	2709	0.134
Pop-AL2	6696	0.331	*Push-SC0*	2705	0.134
Push-AL8	6252	0.309	*Push-Last-AL3*	2701	0.134
Push-AL6	6196	0.306	*Car-AL1*	2714	0.134
Push-SC4	6164	0.305	*Cdr-AL0*	2704	0.134
Call-SC1	5989	0.296	*List-SIC3*	2707	0.134
Push-SIC2	5970	0.295	*Bind-Pop-SC0*	2684	0.133
Push-Last-AL1	5743	0.284	*Call-SC6*	2664	0.132
Push-AL7	5715	0.283	*Call-Maybe-Multiple-SC0*	2672	0.132
Pop-AL4	5489	0.271	*Push-SIC5*	2600	0.129
Set-Null-Unless-Arg-Supplied-AL	5346	0.264	*=-SIC3*	2545	0.126
Call-Maybe-Multiple-SC1	5216	0.258	*Call-Maybe-Multiple-SC3*	2528	0.125
Push-SIC4	5225	0.258	*Cons-Stack*	2510	0.124
Push-SC1	5121	0.253	*Push-Last-AL2*	2464	0.122
List-SIC2	4880	0.241	*Check-AL2*	2470	0.122
Push-AL9	4853	0.240	*Push-SC7*	2431	0.120
Type-Stack	4680	0.231	*=-SIC7*	2377	0.118
Push-S0	4673	0.231	*List*-SIC2*	2376	0.117
Return-Unless-Null-Stack	4561	0.225	*Push-SC9*	2272	0.112
Call-SC4	4512	0.223	*Push-AL12*	2225	0.110
Push-SC5	4497	0.222	*Call-SC7*	2191	0.108
Pop-AL5	4301	0.213	*Push-S3*	2176	0.108
Push-AL10	4175	0.206	*Call-SC5*	2166	0.107
Pop-AL7	4005	0.198	*Push-SIC18*	2129	0.105
Call-SC3	3714	0.184	*=-SIC1*	2101	0.104
Type-AL	3644	0.180	*Push-SC8*	2077	0.103
Pop-AL6	3600	0.178	*Check-AL4*	2087	0.103
Push-SC6	3492	0.173	*Push-SIC19*	2057	0.102
Cadr-AL0	3508	0.173	*Car-AL3*	2035	0.101
Misc-AL3	3472	0.172	*Cons-AL*	1966	0.097
Push-AL11	3432	0.170	*Typed-V-Store-Ignore*	1922	0.095
=-SIC2	3423	0.169	**Total : (95 opcodes)**	521,417	25.778

Table 7-1: Incremental Savings Per New Instruction

I. Current Instruction Set Summary

I.1 Introduction

The majority of the macro-instructions in the current (June, 1982) Spice Lisp set are of the following form:



Most instructions read from or write to an "effective address", and possibly also push or pop 32-bit words on the stack. When the OP field indicates that an effective address is to be read from, it is computed from the A field and (sometimes) from the subsequent byte B as follows:

A = 0 — The operand is popped off the stack. Then the operation takes place, in some cases popping a second (distinct) argument off the stack and/or pushing something onto the stack. No B byte is fetched.

A = 1 — The next byte is fetched and is converted (with sign extension) to a signed fixnum in the range -128 to +127. This is used as the operand.

A = 2 — The next byte is fetched. If its sign bit is 0, the remaining 7 bits are used as an unsigned offset (0 - 127) into the vector of symbols and constants in the code object of the current function. If the sign bit is 1, the other 7 bits are used instead as an unsigned offset (0 - 126) into the arguments and local variables area of the currently-active stack frame. The contents of this cell are used as the operand. If the fetched byte is all ones (377 octal), the next two bytes are fetched to form a 16-bit offset. The sign bit of this extended offset controls where the operand comes from, as in the 8-bit offsets. In fetching this double offset, the low-order byte comes in first.

A = 3 — The next byte (or set of bytes) is fetched and is used as an offset into the code object, as above; this will never be used with an offset into the stack frame. Instead of being used directly, the constant addressed is supposed to be a symbol pointer, and the operand is fetched from its value cell. If the value is Misc-Trap, an UNBOUND error is signalled.

If the effective address is being used as a place to write, the following descriptions apply:

A = 0 — The result is pushed on the stack.

A = 1 — The result sets the indicators, then is thrown away.

A = 2 If the offset indicates a stack frame destination, the result is put there; if it points into the code object, this destination is illegal, since the code object should not be altered.

A = 3 This writes into the value cell of the symbol pointed to, forwarding the write through an EVC-Forward pointer if one is present in the value cell.

In the following listing, the effective address is called "E" and its contents are called "CE".

I.2 Short Instructions

Note: In the following descriptions, the number in the left margin is the 6-bit opcode, in decimal notation.

0 Unused

1 Call CE must be some sort of executable function: a code object, a lambda-expression in list space, a closure, or a symbol with one of these stored in its function cell. A call block for this function is opened, and computation proceeds to gather the arguments into the call block. The state of the indicators after CALL is undefined.

2 Call-0 CE must be an executable function, as above, but is a function of 0 arguments. Thus, there is no need to collect arguments. The call block is opened and activated in a single operation. The indicators are left in an undefined state.

3 Call-Multiple Just like a Call instruction, except that the function being called should return multiple values.

4 Call-Maybe-Multiple
If the function being called returns multiple values, this is identical to Call-Multiple. If not, this is identical to Call.

5 Return Return from the current function call. After the current function's frame is popped off the stack, CE is pushed as the result being returned. CE also sets the indicators.

6 Throw CE is the throw-tag, normally a symbol. The value to be returned, either single or multiple, is on the top of the stack.

7 Unused.

8 Push CE is pushed onto the stack and sets the indicators. If A = 0, this is a NOOP, except that the indicators are set according to the value of the item on top of the stack.

9 Push-Last CE is pushed onto the stack as the last operand for the most recent currently-open call

block. The call is then activated: the call block is finished and becomes the current stack-frame. If A = 0, the effect of this operation is just to start the call. The indicators are undefined at the start of the called function; they are set by the returned value when execution resumes in the calling function.

10 Push-Under CE is pushed onto the stack as the second item and sets the indicators; the top item of the stack is unchanged. If A = 0, this swaps the top two items on the stack. Push-Under causes an error if the stack is empty or if A = 0 and the stack contains only one item.

11 Check CE is used to set the indicators, but is not put anywhere. If A = 0, the net effect is to pop the stack by one word, setting indicators.

12 Pop Pop the top item off the stack and store it in E, setting the indicators.

13 Copy Copy the item on top of the stack into E, setting the indicators, without popping the stack.

14 Make-Predicate
If the NULL indicator is on, put NIL in E. Else, put T in E. The NIL or T also sets the indicators.

15 Not-Predicate If the NULL indicator is not on, put NIL in E. Else, put T in E. The NIL or T also sets the indicators.

16 Car CE had better be either a pointer to a list or NIL. Its Car is pushed on the stack and sets the indicators.

17 Cdr The Cdr of CE is pushed on the stack and sets the indicators.

18 Cadr The Cadr of CE is pushed on the stack and sets the indicators.

19 Cddr The Cddr of CE is pushed on the stack and sets the indicators.

20 Cdar The Cdar of CE is pushed on the stack and sets the indicators.

21 Caar The Caar of CE is pushed on the stack and sets the indicators.

22 Scdr Get the Cdr of CE and store it in E and the indicators. Useful for Cdr'ing down lists. CE must be a list cell or NIL.

23 Scddr Get the Cddr of CE and store it in E and the indicators. Useful for Cddr'ing down property lists. CE must be a list cell or NIL.

24 Trunc Performs the equivalent of the TRUNC function as described in the Spice Lisp Manual.

After obtaining CE, take one value off the top of the stack to determine what is to be returned setting the indicators.

25 + CE is added to the value popped off the stack. The result is pushed back onto the stack and sets the indicators.

26 - Analogous, but CE is subtracted from TOS.

27 * Analogous, CE is multiplied by TOS.

28 / The TOS is divided by CE; the quotient goes back to TOS.

29 Bit-And Bitwise boolean AND of CE and top of stack. The result goes onto the stack and sets the indicators. The operands must be fixnums or bignums.

30 Bit-Xor Bitwise XOR.

31 Bit-Or Bitwise OR.

32 Eql CE is compared to the value popped off the stack. If these arguments are EQ or if they are both numbers of identical type and value, T sets the indicators; if not, NIL sets the indicators. Nothing is pushed back onto the stack.

33 = CE is compared arithmetically to the value popped off the stack. If they are equal, T sets the indicators; if not, NIL sets the indicators. Nothing is pushed back onto the stack. This works for mixed number-types: 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.

34 > Analogous, but non-null if TOS > CE.

35 < Analogous, but non-null if TOS < CE.

36 EQ CE is compared to the value popped off the stack. If these objects are identical 32-bit Lisp objects, T sets the indicators; if not, NIL sets the indicators.

37 1+ Add 1 to CE, store result back into E.

38 1- Subtract 1 from CE, store result back into E.

39 Bind-Null CE must be a symbol. This is rebound and set to NIL. The NULL indicator is set.

40 Bind-T CE must be a symbol. This is rebound and set to T, which also sets the indicators.

41 Bind-Pop — CE must be a symbol. This is rebound and is set to a value popped off the stack. This value also sets the indicators.

42 Set-Null — Store NIL in E.

43 Set-0 — Store fixnum 0 in E.

44 Set-T — Store T in E.

45 - 47 Unused.

48 NPop — CE is a fixnum N. If N is non-negative, N items are popped off the stack. If N is negative, NIL is pushed onto the stack |N| times. The indicators are unchanged.

49 Unbind — CE is a non-negative fixnum indicating how many bindings are to be popped off the binding stack and restored to their previous values. Used in exiting open-coded PROGs and LAMBDAs. The indicators are unchanged by this instruction.

50 Set-Lpush — Pop TOS, cons it onto CE (in the space indicated by the value of the symbol ALLOCATION-SPACE), store result back into E. The new CE sets the indicators.

51 Set-Lpop — CAR of CE is pushed onto the stack and sets the indicators; CDR of CE is stored back into E.

52 List — CE is a non-negative fixnum N. Beginning with a list of NIL, N items are popped off the stack and CONSed onto this list, so that the last item popped ends up as the CAR of the list. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed on the stack and sets the indicators.

53 List* — CE is a non-negative fixnum N. One item is popped off the stack, to begin the list L. Then N other items are popped and CONSed onto the front of L in succession, so that the last item popped becomes the CAR of L. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed onto the stack and sets the indicators.

54 Spread — CE is a list. Its elements are pushed onto the stack in left-to-right order. The last item pushed sets the indicators.

55 Long-Escape — This is used for calling a large number of microcoded functions. The next byte in the instruction stream is fetched, and this is used to indicate which of 256 long instructions is to be called. This operation will in general pop some arguments off the stack, compute a single result, then place this result in the location indicated by the effective address E, computed as usual from the A field of the first byte. Note that if one or more offset bytes

are needed for the effective address computation, these bytes are fetched *after* the byte telling which instruction is to be called. For more information about the long instructions, see [slguts] under "Misc instructions". (Many fewer than 256 long instructions are defined.)

56 Branch Unconditional branch relative to the current byte-PC (which has been incremented to point past the current instruction). The next byte or two bytes is fetched. This, treated as a signed integer, is added to the PC. The indicators are unchanged. For all of the branch instructions, the bits of the A field are interpreted as follows:

Bit 0 = 0 Fetch one byte for branches of -128 to +127 bytes.

Bit 0 = 1 Fetch two bytes for longer branches. The low-order byte comes in first.

Bit 1 = 0 Do not pop stack.

Bit 1 = 1 Pop stack if the (conditional) branch is *not* taken.

57 Branch-If-Arg-Supplied
This is a special conditional branch that is used by the machinery that computes default values for optional function arguments that were not supplied by the caller. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question contains Misc-Unsupplied-Arg, do not branch; otherwise, take the branch. The branch is executed normally, using the A-field of the instruction to control the usual branch options. The branch offset byte(s) will follow the argument offset byte in the instruction stream.

58 Branch-Null Branch if the NULL indicator is on. Does not alter indicators (nor do any of the other branches).

59 Branch-Not-Null
Branch if the NULL indicator is not on.

60 Branch-Atom Branch if the ATOM indicator is on.

61 Branch-Not-Atom
Branch if the ATOM indicator is not on.

62 Branch-Zero Branch if the ZERO indicator is on.

63 Branch-Not-Zero
Branch if the ZERO indicator is not on.

I.3 Long Instructions

The following instructions are long format instructions. Each of these expects a fixed number of arguments to have been pushed on the stack in the order indicated (leftmost arg pushed first). These arguments are popped and a single return value is generated. This sets the indicators and goes to the E location of the long instruction. The numbers in the left margin are the 8-bit codes corresponding to each instruction, in decimal format. These descriptions are intended only to give the reader some idea of what instructions are available; they are not intended to be a manual.

0 Cons (X Y) Conses up a list cell with X as CAR and Y as CDR.

1 Alloc-Symbol (N)
Allocates one symbol and returns a pointer to it. The symbol is not interned by this operation -- that is done in macrocode.

2 Alloc-B-Vector (N I)
Allocates a B-Vector of N entries and returns a pointer to it. I is the initial value with which the vector is filled.

3 Alloc-U-Vector (N A)
Allocate a local U-Vector with access-code A and a length of N items, and return a pointer to it. All entries are initialized to 0.

4 Alloc-Remote-Vector (N A P)
Allocate a remote U-Vector with N entries and access-code A, returning a pointer to it. P is the pointer to the data area in system-table space.

5 Alloc-String (N) Allocate a string of length N, initialized to all 0's, and return a pointer to it.

6 Alloc-Function (N)
Allocate a function object (like a B-Vector) of length N, not counting the 1-word header.

7 Alloc-Array (N) Allocate an array-header for an array of N dimensions. Returns a pointer to the array header.

8 Alloc-Xnum (N X)
Allocate an xnum N bytes in length, with sub-type code X. N and X must be fixnums. All entries of the XNUM vector are initialized to 0.

9 Alloc-Ynum (N X)
Allocate a Ynum N lisp-objects in length, with sub-type code X. N and X must be fixnums. All entries of the YNUM vector are initialized to Misc-Trap codes.

10 Misc-Subtype (X)

X must be of type MISC. Returns the subtype field (bits 24-27) of X right-justified in a fixnum.

11 Type (X) Returns the 4-bit type-code of X as a fixnum.

12 Make-Immediate-Type (OBJ TYPE)

OBJ can be any lisp object, TYPE is a fixnum in the range 0 - 2, which correspond to the type-codes of immediate objects. Returns an object whose type-code bits are TYPE, but whose other bits are those of OBJ.

13 Get-Vector-Subtype (V)

Returns the 4-bit subtype field of a vector-like object V (B-vector, U-Vector, Array, Xnum, String, Function). Returned as a fixnum.

14 Set-Vector-Subtype (V X)

Stores the low order 4 bits of fixnum X as the subtype code of vector-like thing V. Returns V.

15 Get-Vector-Length (V)

V is any vector-like thing. Returns the length of this vector.

16 Get-Value (S) Gets the contents of the value cell of the symbol S.

17 Set-Value (S V)

Set the value cell of symbol S to V. Returns V.

18 Get-Definition (S)

Returns the contents of the functional definition cell of symbol S.

19 Set-Definition (S D)

Puts D into the functional definition cell of symbol S. Returns D.

20 Get-Plist (S) Returns the property list of symbol S.

21 Set-Plist (S P) Sets the property list of symbol S to P. P should be NIL or a List object. Returns P.

22 Get-Pname (S) Returns the pname of symbol S.

23 Set-Pname (S P)

Sets the pname of symbol S to P. P should be a string. Returns P.

24 Get-Package (S)

Gets the contents of the package cell of symbol S.

25 Set-Package (S P)
Sets the package of symbol S to P. Returns P.

26 Get-Hash (S) Gets the contents of the hash cell of the symbol S.

27 Set-Hash (S H) Set the hash cell of symbol S to H. Returns H.

28 Boundp (S) S must be a symbol. Boundp returns T if the value cell of the symbol contains a value, NIL otherwise.

29 Fboundp (S) S must be a symbol. Fboundp returns T if the definition cell of the symbol contains a definition, NIL otherwise.

30 Rplaca (L X) Replaces car of L with X, returning the modified L.

31 Rplacd (L X) Replaces cdr of L with X, returning the modified L.

32 Unused.

33 S-Float (X) Turns any number X into a short flonum.

34 L-Float (X) Turns any number X into a long flonum.

35 Negate (X) For any number X, return the negative.

36 Lsh (N B) Both args are fixnums. Returns a fixnum that is N shifted left by B bits, with 0's shifted in on the right. If B is negative, N is shifted to the right with 0's coming in on the left.

37 Get-Vector-Access-Type (V)
V must be a U-Vector. Returns its access-type code.

38 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 returned right-justified in 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.

39 Logdpb (V S P N)
All args are fixnums. Returns a number equal to N, but with the field specified by P and S replaced by the S low-order bits of V.

40 Abs (N) N is any kind of number. Returns the absolute value of N.

41 Subspace (X) X is any lisp object. Returns the 2-bit allocation space code as a fixnum. Returns NIL if the object is immediate.

42 Close-Over (L) L is a list of symbols. Creates and returns a closure-list for these symbols in the current environment.

43 Activate-Closure (C)

C must be a closure list, as returned by the Close-Over operation. Activate-Closure restores the environment in which the closure list was created for the symbols closed over. Returns C unchanged.

44 Typed-V-Access (A V I)

A and I are fixnums, V points to a U-Vector or Xnum. This returns entry I of the V as a fixnum, but uses the low-order three bits of A as the access-type code instead of whatever code is stored in the vector itself. This is illegal if V is a string.

45 Typed-V-Store (A V I X)

Like a V-Store, but stores X in entry I of V using A as the low-order 3 bits of the access-type code, as above. Returns X. Illegal for strings.

46 Unused.

47 Freeze () Freezes all read-only spaces by moving the FREEZE pointers up to meet the FREE-STORAGE pointers. Returns NIL.

48 New-Pure-Page (X)

X can be an item of any non-immediate data type. The type of X is examined, and the current read-only page for that type of storage is closed. Returns X.

49 Shrink-Vector (V N)

V is any B-Vector, U-Vector, String, Function object, or Array header. N is the new number of entries, a fixnum, which must be less than or equal to the current number of entries. Returns V, the vector which has been shortened.

50 Call-Break (F) Just like the Call operation, but starts the new frame in such a way that when the called function ultimately returns, no return value is left on the stack. Returns NIL, though this will normally be called with destination IGNORE.

51 Values-To-N (V)

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

52 N-To-Values (N)

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

53 Arg-In-Frame (N F)

N is a fixnum, F is a control stack pointer as returned by the CURRENT-STACK-FRAME and CURRENT-OPEN-FRAME operators. Returns the item in slot N of the args-and-locals area of stack frame F.

54 Current-Stack-Frame ()

Returns a control-stack pointer to the start of the currently active stack frame.

55 Set-Stack-Frame (P)

P must be a control stack pointer. This becomes the current active frame pointer. Returns NIL.

56 Current-Open-Frame ()

Returns a control-stack pointer to the start of the currently open stack frame.

57 Set-Open-Frame (P)

P must be a control stack pointer. This becomes the current open frame pointer. Returns NIL.

58 Current-Stack-Pointer ()

Returns the 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.

59 Current-Binding-Pointer ()

Returns a pointer to the first word above the current top of the binding stack.

60 Read-Control-Stack (F)

F must be a control stack pointer. Returns the lisp object that resides at this location.

61 Write-Control-Stack (F V)

F is a stack pointer, V is any Lisp object. Writes V into the location addressed. Returns V.

62 Read-Binding-Stack (B)

B must be a binding stack pointer. Reads and returns the lisp object at this location.

63 Write-Binding-Stack (B V)

B must be a binding stack pointer. Writes V into the specified location. Returns V.

64 Ldb (S P N) All args are fixnums or bignums; S and P are non-negative. S and P specify a "byte" or bit-field of any length within N. This is extracted and is returned right-justified as a positive integer. 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.

65 Mask-Field (S P N)

Like LDB, except that the extracted field is returned in the same position it occupies in N, not moved to the right. The result is a positive fixnum or bignum with 0 in all positions except that specified by the S-P field.

66 Dpb (V S P N) All args are fixnums or bignums; P and S are non-negative. Returns a number equal to N, and with the same sign as N, but with the field specified by P and S replaced by the S low-order bits of V.

67 Deposit-Field (V S P N)

Like DPB, except that the bits to be put in N are extracted from the corresponding field of V, not from the rightmost S bits of V.

68 Ash (N C) N and C are fixnums or bignums. Shift N left C places, shifting in zeros on the right. If C is negative, shift N right -C places, preserving the sign of N.

69 Haulong (N) N is a fixnum or bignum. Returns the number of significant bits of N.

70 V-Access (V I) V is any vector or vector-like object (B-Vector, U-Vector, String, Xnum, Array, or Function Object). I is a Fixnum. Returns entry I of V.

71 V-Store (V I X)

V is any vector or vector-like object. I is a fixnum. X is the value to be stored into slot I of vector V. X is returned.

72 - 79 Unused. Reserved for I/O operators.

80 Force-Values ()

If the top of the stack is a multiple-value marker, do nothing; if not, push a multiple-value marker indicating 1 value. Returns NIL.

81 Flush-Values ()

If the top of the stack is a multiple-value marker, remove this marker; if not, do nothing. Returns NIL.

82 Mark-Catch-Frame ()

Mark the header word of the current *open* frame, indicating that this is a catch-tag frame. Returns NIL.

83 Get-Newspace-Bit ()

Returns a fixnum 0 or 1, indicating whether the current newspace is Dynamic-0 or Dynamic-1.

Other long instructions are still to be defined.

II. Recommended Assembler Instruction Set

II.1 Effective Address Specification

Most instructions read from or write to an "effective address", and possibly also push or pop 32-bit words on the stack. The following is a description of the various types of operands that can be used as the effective address for these instructions. The item in parentheses after the operand source name is the suffix which is appended to instruction names to signify the sort of operand.

Stack (Stack) The operand is taken from the stack; usually the stack is popped (if not, the instruction description explicitly states what happens). Then the operation takes place, in some cases popping a second (distinct) argument off the stack and/or pushing something onto the stack. No operand bytes are fetched.

Short Integer Constant (SIC)

A byte is fetched and is converted (with sign extension) to a signed fixnum in the range -128 to +127. This is used as the operand.

Arguments & Locals (AL, ALn)

In most cases, one byte is fetched and used as an unsigned offset (0 - 254) into the arguments and local variables area of the currently active stack frame. The contents of this cell are used as the operand. For several instructions, two bytes are fetched to form a 16-bit offset. In fetching this double offset, the low-order byte comes in first. 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 denoted the implied offset.

Symbols & Constants (SC SCn Sn)

In most cases, one byte is fetched (SC). The low order 7 bits are used as an unsigned offset into the vector of symbols and constants in the code object of the current function. If the sign bit is 0, the the constant is used directly. If the sign bit is 1, instead of being used directly, the constant addressed is supposed to be a symbol pointer, and the operand is fetched from its value cell. If the value is Misc-Trap, an UNBOUND error is signalled. For some instructions, the next two bytes are fetched to form a 16-bit offset. The sign bit of this extended offset controls the interpretation of the operand, as in the 8-bit offsets. In fetching this double offset, the low-order byte comes in first. (Note: When using a one byte offset for symbol pointers (sign bit is 1), the offset nee no longer be restricted to the range 0 - 126 (it may be 0 - 127) to avoid an all 1's byte since long offsets are now implicitly specified.) Sometimes an instruction implies an offset into the symbols and constants without the need of another byte for the offset. In those instances when the symbol or constant is to be used directly, the instruction will have the suffix SCn where n is an integer

denoting the offset. If the symbol or constant accessed is to be used as a symbol pointer (indirectly), then the suffix is Sn.

If the effective address is being used as a place to write, the following descriptions apply:

Stack (Stack) The result is pushed on the stack.

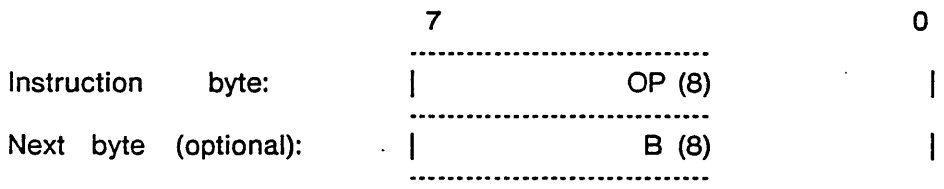
Ignore (Ignore) The result sets the indicators, then is thrown away.

Arguments and Locals (AL ALn)
The result is written into the appropriate cell of the arguments and local variables area of the currently active stack frame, as specified above.

Symbols (S Sn) Offset bytes are fetched and interpreted as a symbol pointer. The result written into the value cell of the symbol pointed to. Note that this can not be an offset directly into the vector of symbols and constants of a code object -- the code object should not be altered.

II.2 Short Instruction Descriptions

Many of the macro-instructions in the assembler instruction set are of the following form:



The OP field of the following instructions indicates what sort of effective address is to be read from or written to according to the descriptions above. In the following descriptions, the effective address is called "E" and its contents are called "CE". Instruction names are given a suffix which is a mnemonic abbreviation for this address. For example, XXX-AL denotes that the instruction's operand is an argument or local variable. A B-byte should be fetched which gives an offset into the block of arguments and locals on the control stack. If the instruction name is of the form XXX-AL3, then E is the third argument or local variable. (No offset byte is fetched - the offset is *implied* by the opcode). The legal suffixes are listed alongside the descriptions in Section II.1.

Some short instructions expect a fixed number of arguments to have been pushed on the stack. The arguments for these instructions are listed beside the instruction in the order in which they are expected (leftmost arg pushed first). These arguments are popped when the instruction is executed.

Note: In the following descriptions, the opcode listed for each instruction is an 8-bit opcode, in decimal notation.

Call

CE must be some sort of executable function: a code object, a lambda-expression in list space, a closure, or a symbol with one of these stored in its function cell. A call block for this function is opened, and computation proceeds to gather the arguments into the call block. The state of the indicators after CALL is undefined. The following *Call* instructions are implemented :

0	Call-Stack	6	Call-SC3
1	Call-AL	7	Call-SC4
2	Call-SC	8	Call-SC5
3	Call-SC0	9	Call-SC6
4	Call-SC1	10	Call-SC7
5	Call-SC2		

Call-0

CE must be an executable function, as above, but is a function of 0 arguments. Thus, there is no need to collect arguments. The call block is opened and activated in a single operation. The indicators are left in an undefined state. Only one *Call-0* instruction is implemented :

11 Call-0-SC

Call-Multiple

Just like a Call instruction, except that the function being called should return multiple values. See also the long format version of *Call-Multiple.*

12 Call-Multiple-Stack
13 Call-Multiple-SC

Call-Maybe-Multiple

If the function being called returns multiple values, this is identical to Call-Multiple. If not, this is identical to Call. Five *Call-Maybe-Multiple* instructions are implemented :

14	Call-Maybe-Multiple-Stack	17	Call-Maybe-Multiple-SC0
15	Call-Maybe-Multiple-AL	18	Call-Maybe-Multiple-SC1
16	Call-Maybe-Multiple-SC	19	Call-Maybe-Multiple-SC3

Return

Return from the current function call. After the current function's frame is popped off the stack, CE is pushed as the result being returned. CE also sets the indicators.

20 Return-Stack
21 Return-SIC
22 Return-AL
23 Return-SC

Return-Unless-Null

Return from the current function call unless the Null indicator is set. If the Return is successful, the current function's frame is popped off the stack, CE is pushed as the result being returned. CE also sets the indicators.

24 Return-Unless-Null-Stack

Throw

CE is the throw-tag, normally a symbol. The value to be returned, either single or multiple, is on the top of the stack. Currently, only one *throw* instruction is implemented which must use the Stack as E.

25 Throw-Stack
26 Throw-SC
27 Throw-AL

Push

CE is pushed onto the stack and sets the indicators. For Push-Long-SC, a two byte offset into the vector of symbols and constants is used to specify CE. See also the long format Push instruction for two-byte offsets into the arguments and local variables. (Note : *Push Stack* is not a legal combination. To set the indicators according to TOS, *Check Stack* should be used.) Many *push* instructions are available :

28	Push-SIC	47	Push-AL9
29	Push-SIC1	48	Push-AL10
30	Push-SIC2	49	Push-AL11
31	Push-SIC3	50	Push-AL12
32	Push-SIC4	51	Push-SC
33	Push-SIC5	52	Push-Long-SC
34	Push-SIC8	53	Push-SC0
35	Push-SIC18	54	Push-SC1
36	Push-SIC19	55	Push-SC2
37	Push-AL	56	Push-SC3
38	Push-AL0	57	Push-SC4
39	Push-AL1	58	Push-SC5
40	Push-AL2	59	Push-SC6
41	Push-AL3	60	Push-SC7
42	Push-AL4	61	Push-SC8
43	Push-AL5	62	Push-SC9
44	Push-AL6	63	Push-S0
45	Push-AL7	64	Push-S1
46	Push-AL8	65	Push-S3

Push-Last

CE is pushed onto the stack as the last operand for the most recent currently-open call block. The call is then activated: the call block is finished and becomes the current

stack-frame. If E is the stack, the effect of this operation is just to start the call. The indicators are undefined at the start of the called function; they are set by the returned value when execution resumes in the calling function. Several *Push-Last* instructions are supplied :

66	Push-Last-Stack	70	Push-Last-AL1
67	Push-Last-SIC	71	Push-Last-AL2
68	Push-Last-AL	72	Push-Last-AL3
69	Push-Last-AL0	73	Push-Last-SC

Push-Under

CE is pushed onto the stack as the second item and sets the indicators; the top item of the stack is unchanged. If A = 0, this swaps the top two items on the stack. Push-Under causes an error if the stack is empty or if A = 0 and the stack contains only one item.

74 Push-Under-Stack

Check

CE is used to set the indicators, but is not put anywhere. If E is the stack, the indicators are set; the stack is **unchanged.**[13] This is the operation one should use rather than *Pop Stack*, *Push Stack* or *Copy Ignore.* See also the long format *Check* instruction.

75 Check-AL
76 Check-AL2
77 Check-AL4
78 Check-SC

Pop

Pop the top item off the stack and store it in E, setting the indicators. *Pop stack* is not useful. It would set the indicators leaving the stack unchanged if it existed. *Check Stack* is the proper instruction for this action. See also the long format Pop instructions which allow two-byte operand offsets.

79	Pop-Ignore	85	Pop-AL5
80	Pop-AL	86	Pop-AL6
81	Pop-AL1	87	Pop-AL7
82	Pop-AL2	88	Pop-AL8
83	Pop-AL3	89	Pop-SC
84	Pop-AL4		

Copy

Copy the item on top of the stack into E, setting the indicators, without popping the stack. *Copy Ignore* is not a useful instruction; *Check Stack* should be used if this effect is desired. See also the long format version of *Copy.*

---

[13] This is different from the semantics of the current *(Check Stack)* instruction!

90 Copy-AL
91 Copy-S

Make-Predicate If the NULL indicator is on, put NIL in E. Else, put T in E. The NIL or T also sets the indicators.

92 Make-Predicate-Stack
93 Make-Predicate-Ignore
94 Make-Predicate-AL

Not-Predicate If the NULL indicator is not on, put NIL in E. Else, put T in E. The NIL or T also sets the indicators.

95 Not-Predicate-Stack
96 Not-Predicate-Ignore
97 Not-Predicate-AL

Car CE had better be either a pointer to a list or NIL. Its Car is pushed on the stack and sets the indicators.

98 Car-Stack
99 Car-AL
100 Car-AL0
101 Car-AL1
102 Car-AL2
103 Car-AL3
104 Car-SC

Cdr The Cdr of CE is pushed on the stack and sets the indicators.

105 Cdr-Stack
106 Cdr-AL
107 Cdr-AL0
108 Cdr-SC

Cadr The Cadr of CE is pushed on the stack and sets the indicators.

109 Cadr-Stack
110 Cadr-AL
111 Cadr-AL0
112 Cadr-SC

Cddr The Cddr of CE is pushed on the stack and sets the indicators.

113 Cddr-Stack
114 Cddr-AL

Cdar — The Cdar of CE is pushed on the stack and sets the indicators.

115 Cdar-Stack

Caar — The Caar of CE is pushed on the stack and sets the indicators.

116 Caar-Stack
117 Caar-AL
118 Caar-SC

Scdr — Get the Cdr of CE and store it in E and the indicators. Useful for Cdr'ing down lists. CE must be a list cell or NIL. See also the long version of this instruction.

119 Scdr-AL
120 Scdr-SC

Push-Car-Scdr — Push the Car of CE on the stack and store the Cdr of CE in E and the indicators. Useful for Cdr'ing down lists. CE must be a list cell or NIL.

121 Push-Car-Scdr-AL

Scddr — Get the Cddr of CE and store it in E and the indicators. Useful for Cddr'ing down property lists. CE must be a list cell or NIL. See also the long version of this instruction.

122 Scddr-AL
123 Scddr-SC

Trunc — Performs the equivalent of the TRUNC function as described in the Spice Lisp Manual. After obtaining CE, take one value off the top of the stack to determine what is to be returned setting the indicators. See also the long format *Trunc* instruction.

124 Trunc-Stack
125 Trunc-Ignore
126 Trunc-AL

\+ — CE is added to the value popped off the stack. The result is pushed back onto the stack and sets the indicators.

127 +Stack
128 +SIC
129 +AL
130 +SC

\- — Analogous, but CE is subtracted from TOS.

131 -Stack
132 -SIC
133 -AL
134 -SC

* Analogous, CE is multiplied by TOS.

135 *Stack
136 *SIC
137 *AL
138 *SC

/ The TOS is divided by CE; the quotient goes back to TOS. See also the long format / instruction. See also the long format / instruction for symbols and constants.

139 /Stack
140 /SIC
141 /AL

Bit-And Bitwise boolean AND of CE and top of stack. The result goes onto the stack and sets the indicators. The operands must be fixnums or bignums.

142 Bit-And-Stack
143 Bit-And-SIC
144 Bit-And-AL
145 Bit-And-SC

Bit-Xor Bitwise XOR.

146 Bit-Xor-Stack
147 Bit-Xor-SIC
148 Bit-Xor-AL

Bit-Or Bitwise OR.

149 Bit-Or-Stack
150 Bit-Or-AL
151 Bit-Or-SC

Eql CE is compared to the value popped off the stack. If these arguments are EQ or if they are both numbers of identical type and value, T sets the indicators; if not, NIL sets the indicators. Nothing is pushed back onto the stack.

152 Eql-Stack

153 Eql-AL

=

CE is compared arithmetically to the value popped off the stack. If they are equal, T sets the indicators; if not, NIL sets the indicators. Nothing is pushed back onto the stack. This works for mixed number-types: 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.

154 =Stack
155 =SIC
156 =SIC1
157 =SIC2
158 =SIC3
159 =SIC4
160 =SIC7
161 =AL
162 =SC

>

Analogous, but non-null if TOS > CE.

163 >Stack
164 >SIC
165 >AL
166 >SC

<

Analogous, but non-null if TOS < CE.

167 <Stack
168 <SIC
169 <AL
170 <SC

EQ

CE is compared to the value popped off the stack. If these objects are identical 32-bit Lisp objects, T sets the indicators; if not, NIL sets the indicators.

171 Eq-Stack
172 Eq-SIC
173 Eq-AL
174 Eq-AL3
175 Eq-SC

1+

Add 1 to CE, store result back into E.

176 1+Stack
177 1+AL
178 1+SC

1-

Subtract 1 from CE, store result back into E.

179 1-Stack

180 1-AL

Bind-Null CE must be a symbol. This is rebound and set to NIL. The NULL indicator is set. See also the long format *Bind-Null* instruction.

181 Bind-Null-Stack
182 Bind-Null-AL
183 Bind-Null-SC

Bind-T CE must be a symbol. This is rebound and set to T, which also sets the indicators.

184 Bind-T-Stack

Bind-Pop CE must be a symbol. This is rebound and is set to a value popped off the stack. This value also sets the indicators.

185 Bind-Pop-Stack
186 Bind-Pop-AL
187 Bind-Pop-SC
188 Bind-Pop-SC0

Set-Null Store NIL in E.

189 Set-Null-Stack
190 Set-Null-Ignore
191 Set-Null-AL
192 Set-Null-SC

Set-Null-Unless-Arg-Supplied
This is a special conditional instruction that is used by the machinery that computes default values for optional function arguments that were not supplied by the caller. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question contains Misc-Unsupplied-Arg, set the entry to NIL; otherwise, do nothing.

193 Set-Null-Unless-Arg-Supplied-AL

Set-0 Store fixnum 0 in E.

194 Set-0-Stack
195 Set-0-AL
196 Set-0-SC

Set-T Store T in E.

197 Set-T-Stack
198 Set-T-Ignore
199 Set-T-AL
200 Set-T-SC

NPop — CE is a fixnum N. If N is non-negative, N items are popped off the stack. If N is negative, NIL is pushed onto the stack |N| times. The indicators are unchanged.

201 NPop-Stack
202 NPop-SIC

Unbind — CE is a non-negative fixnum indicating how many bindings are to be popped off the binding stack and restored to their previous values. Used in exiting open-coded PROGs and LAMBDAs. The indicators are unchanged by this instruction. See also the long format *Unbind* instruction.

203 Unbind-SIC
204 Unbind-SIC1
205 Unbind-AL

Set-Lpop — CAR of CE is pushed onto the stack and sets the indicators; CDR of CE is stored back into E. See also the long format instruction *Set-Lpush.*

206 Set-Lpop-Stack
207 Set-Lpop-AL
208 Set-Lpop-SC

List — CE is a non-negative fixnum N. Beginning with a list of NIL, N items are popped off the stack and CONSed onto this list, so that the last item popped ends up as the CAR of the list. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed on the stack and sets the indicators.

209 List-Stack
210 List-SIC
211 List-SIC1
212 List-SIC2
213 List-SIC3

List* — CE is a non-negative fixnum N. One item is popped off the stack, to begin the list L. Then N other items are popped and CONSed onto the front of L in succession, so that the last item popped becomes the CAR of L. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed onto the stack and sets the indicators. See also the long format *List** instruction.

214 List*SIC
215 List*SIC2

Spread

CE is a list. Its elements are pushed onto the stack in left-to-right order. The last item pushed sets the indicators.

216 Spread-Stack
217 Spread-AL
218 Spread-SC

Long-Escape

This is used for calling a large number of microcoded functions. The next byte in the instruction stream is fetched, and this is used to indicate which of 256 long instructions is to be called. This operation will in general pop some arguments off the stack, compute a single result, then place this result in the location indicated by the effective address E, denoted as usual by the *Long-Escape* instruction suffix. Note that if one or more offset bytes are needed for the effective address computation, these bytes are fetched *after* the byte telling which instruction is to be called. For more information about long instructions, see [slguts] under Misc codes.

219 Long-Escape-Stack
220 Long-Escape-Ignore
221 Long-Escape-AL
222 Long-Escape-AL2
223 Long-Escape-AL3
224 Long-Escape-AL4
225 Long-Escape-SC

Cons (X TOS)

Conses up a list cell with X as CAR and TOS as CDR. X should be pushed first, then the second argument is pushed to become TOS; both are popped and used as arguments to *Cons*. The new cons is stored in E. See also the long format *Cons* instruction.

226 Cons-Stack
227 Cons-AL

Type (TOS)

Stores the 4-bit type-code of whatever it pops from the TOS in E as a fixnum. See also the long format *Type* instruction.

228 Type-Stack
229 Type-AL

Make-Immediate-Type (OBJ TYPE)

OBJ can be any lisp object, TYPE is a fixnum in the range 0 - 2, which correspond to the type-codes of immediate objects; these arguments are taken from the stack. Returns an object whose type-code bits are TYPE, but whose other bits are those of OBJ. See also the long format version of *Make-Immediate-Type*.

230 Make-Immediate-Type-Stack

Get-Definition (TOS)

The TOS had better be a symbol. This is popped and *Get-Definition* stores the contents of its functional definition cell in E. See also the long format version of *Get-Definition*.

231 Get-Definition-Stack

Typed-V-Store (A V I X)

Like a V-Store, but stores X in entry I of V using A as the low-order 3 bits of the access-type code, as above. Returns X. Illegal for strings. See also the long format version of this instruction.

232 Typed-V-Store-Ignore

V-Access (V TOS)

V is any vector or vector-like object (B-Vector, U-Vector, String, Xnum, Array, or Function Object). TOS is a Fixnum. Stores entry TOS of V in E. See also the long format version of *V-Access*.

233 V-Access-Stack
234 V-Access-Ignore
235 V-Access-AL

V-Store (V I TOS)

V is any vector or vector-like object. I is a fixnum. TOS is the value to be stored into slot I of vector V. TOS is stored in E. *V-Store* is also available in a long format version.

235 V-Store-Ignore

When a branch instruction is recognized, either one or two bytes are fetched and used as a PC-relative offset for branching, depending on whether the branch is short or long. Some conditional branch instructions may pop the stack if the branch is not taken. Branch instruction names are suffixed by either *-short* or *-long*; they may be suffixed by *pop* if the stack is to be popped when the branch is not taken.

Branch

Unconditional branch relative to the current byte-PC (which has been incremented to point past the current instruction). The next byte or two bytes is fetched. This, treated as a signed integer, is added to the PC. The indicators are unchanged. For all of the branch instructions, the bits of the A field are interpreted as follows:

236 Branch-Short
237 Branch-Long

Branch-If-Arg-Supplied

This is a special conditional branch that is used by the machinery that computes default values for optional function arguments that were not supplied by the caller. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question contains

Misc-Unsupplied-Arg, do not branch; otherwise, take the branch. The branch is executed normally, using the A-field of the instruction to control the usual branch options. The branch offset byte(s) will follow the argument offset byte in the instruction stream. See also the long format of this instruction.

238 Branch-If-Arg-Supplied-Short

Branch-Null Branch if the NULL indicator is on. Does not alter indicators (nor do any of the other branches).

240 Branch-Null-Short
241 Branch-Null-Long
242 Branch-Null-Short-Pop
243 Branch-Null-Long-Pop

Branch-Not-Null Branch if the NULL indicator is not on.

244 Branch-Not-Null-Short
245 Branch-Not-Null-Short-Pop

Branch-Atom Branch if the ATOM indicator is on.

246 Branch-Atom-Short
247 Branch-Atom-Long

Branch-Not-Atom

Branch if the ATOM indicator is not on.

248 Branch-Not-Atom-Short
249 Branch-Not-Atom-Long

Branch-Zero Branch if the ZERO indicator is on.

250 Branch-Zero-Short
251 Branch-Zero-Long

Branch-Not-Zero Branch if the ZERO indicator is not on.

252 Branch-Not-Zero-Short

NI-Branch-Not-Atom-AL

This is a special conditional branch that does not use the indicators. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question would set the Atom indicator if given to the *Check* instruction, do not branch; otherwise, take the branch. The branch is

executed without popping the stack and may only be used with a short offset. The branch offset byte will follow the argument offset byte in the instruction stream.

253 NI-Branch-Not-Atom-AL-Short

II.3 Long Instructions

Each long instruction expects a fixed number of arguments to have been pushed on the stack in the order indicated (leftmost arg pushed first). These arguments are popped and a single return value is generated. This sets the indicators and goes to the E location of the long instruction. The long instructions of the assembler's instruction set are similar to those of the existing set except that some of the old long instructions are available are short instructions; and some previously short instructions are now available only as long instructions. For example, *(Cons X Y)* with E either the stack or the block of arguments and locals is available as a short instruction. However, to *Cons* something where E is indirect through the symbols and constants vector of the current function object, the long instruction must be used. Other long instructions which are similar to a short instruction are used in the same manner. Those long instructions which have been changed or added for the recommended assembler instruction set are described below, numbered starting with opcode 200 (decimal) to avoid conflicts with existing long instructions. For a description of other long instructions, see section I.3 of appendix I.

Push-Long CE is pushed onto the stack setting the indicators. Offsets for specifying CE are two bytes long.

200 Push-Long-AL

Pop-Long Pop the item off the stack and store it in E, setting the indicators. Offsets for specifying CE are two bytes long.

201 Pop-Long-AL
202 Pop-Long-SP

Cons (X TOS) Conses up a list cell with X as CAR and TOS as CDR. X should be pushed first, then the second argument is pushed to become TOS; both are popped and used as arguments to *Cons*. The new cons is stored in E. See also the short format *Cons* instruction.

203 Cons-Ignore
204 Cons-AL
205 Cons-SC

Trunc Performs the equivalent of the TRUNC function as described in the Spice Lisp Manual. After obtaining CE, take one value off the top of the stack to determine what is to be returned setting the indicators. See also the short format *Trunc* instruction.

206 Trunc-SC

Branch-If-Arg-Supplied

This is a special conditional branch that is used by the machinery that computes default values for optional function arguments that were not supplied by the caller. The next byte is read from the instruction stream and is taken as an offset (range 0 - 255) into the args-and-locals area of the stack frame. If the stack frame entry in question contains Misc-Unsupplied-Arg, do not branch; otherwise, take the branch. The branch is executed normally, using the A-field of the instruction to control the usual branch options. The branch offset byte(s) will follow the argument offset byte in the instruction stream. See also the short format of this instruction.

207 Branch-If-Arg-Supplied-Long

Type (TOS) Stores the 4-bit type-code of whatever it pops from the TOS in E as a fixnum. See also the short format *Type* instruction.

208 Type-Ignore
209 Type-SC

Make-Immediate-Type (OBJ TYPE)

OBJ can be any lisp object, TYPE is a fixnum in the range 0 - 2, which correspond to the type-codes of immediate objects; these arguments are taken from the stack. Returns an object whose type-code bits are TYPE, but whose other bits are those of OBJ. See also the short format version of *Make-Immediate-Type.*

210 Make-Immediate-Type-Ignore
211 Make-Immediate-Type-AL
212 Make-Immediate-Type-SC

Get-Definition (TOS)

The TOS had better be a symbol. This is popped and *Get-Definition* stores the contents of its functional definition cell in E. See also the short format version of *Get-Definition.*

213 Get-Definition-Ignore
214 Get-Definition-AL
215 Get-Definition-SC

Rplacd (L TOS) Replaces cdr of L with TOS; stores the modified L in E. L should be pushed before the second argument. *Rplacd* is also available in a short format.

216 Rplacd-Stack
217 Rplacd-AL

218 Rplacd-SC

V-Access (V TOS)

V is any vector or vector-like object (B-Vector, U-Vector, String, Xnum, Array, or Function Object). TOS is a Fixnum. Stores entry TOS of V in E. See also the short format version of *V-Access.*

219 V-Access-SC

V-Store (V I TOS)

V is any vector or vector-like object. I is a fixnum. TOS is the value to be stored into slot I of vector V. TOS is stored in E. *V-Store* is also available in a short format version.

220 V-Store-Stack
221 V-Store-AL
222 V-Store-SC

Check CE is used to set the indicators, but is not put anywhere. If E is the stack, the indicators are set; the stack is **unchanged.**[14] This is the operation one should use rather than *Pop Stack*, *Push Stack* or *Copy Ignore.* See also the short format version of *Check.*

223 Check-Stack

Copy Copy the item on top of the stack into E, setting the indicators, without popping the stack. *Copy Ignore* is not a useful instruction; *Check Stack* should be used if this effect is desired. See also the short format *Copy* instruction.

224 Copy-Stack

Scdr Get the Cdr of CE and store it in E and the indicators. Useful for Cdr'ing down lists. CE must be a list cell or NIL. See also the short format *Scdr* instruction.

225 Scdr-Stack

Scddr Get the Cddr of CE and store it in E and the indicators. Useful for Cddr'ing down property lists. CE must be a list cell or NIL. See also the short format of this instruction.

226 Scddr-Stack

Unbind CE is a non-negative fixnum indicating how many bindings are to be popped off the binding stack and restored to their previous values. Used in exiting open-coded PROGs and LAMBDAs. The indicators are unchanged by this instruction. See also the short format *Unbind* instruction.

[14] This is different from the semantics of the current *(Check Stack)* instruction!

227 Unbind-Stack

List* CE is a non-negative fixnum N. One item is popped off the stack, to begin the list L. Then N other items are popped and CONSed onto the front of L in succession, so that the last item popped becomes the CAR of L. The consing is done in the space specified by the value of ALLOCATION-SPACE. The resulting list is pushed onto the stack and sets the indicators. See also the short format *List** instruction.

228 List*Stack

III. Recommended Compiler Instruction Set

Instructions generated by the compiler as input to the assembler are of the form (instruction-name operand-source N) where N is optional. The possible forms are listed in Table III-1 below. These operand sources are described in section I.1 of appendix I.

(instruction-name *Stack*)
(instruction-name *SIC* N)
(instruction-name *Ignore*)
(instruction-name *A&L* N)
(instruction-name *S&C* N)
(instruction-name *S* N)

Table III-1: Compiler Instruction Formats

The instruction set generated by the compiler is like that described in appendix I, except that no distinction is made between long instructions and short instructions and several additional instructions are included as specified in sections 5.3 and 5.4 to incorporate a new type of branch instruction and replace commonly occurring instruction pairs. Several of the legal compiler instructions have no counterpart in the assembler's instruction set. For instance, *(Cadr S&C)* is translated into *(Push S&C)* followed by *(Cadr Stack)*. These translations, however, are transparent to the compiler. The compiler instruction set is designed to be extremely regular and complete in order to make the compiler as easy to write as possible.

References

[1] Gerrit A. Blaauw and Frederick P. Brooks, Jr.
Computer Architecture.
, 1982, to be published.

[2] Guy L. Steele Jr.
Common Lisp Manual
Carnegie-Mellon University, Department of Computer Science, to be published.

[3] Scott E. Fahlman, Guy L. Steele Jr., Gail E. Kaiser, Walter van Roggen.
Internal Design of Spice Lisp
5-23-82 edition, Carnegie-Mellon University, Department of Computer Science, 1982.

[4] Stephen C. Johnson.
A 32-Bit Processor Design.
Computer Science Technical Report 80, Bell Laboratories, April, 1979.

[5] Gene McDaniel.
An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies.
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, 1982.

[6] C. E. Shannon.
A Mathematical Theory of Communication.
Bell System Technical Journal 27, 1948.

[7]
Proposal for a Joint Effort in Personal Scientific Computing.
Carnegie-Mellon University, Department of Computer Science.

[8] Richard E. Sweet.
Empirical Estimates of Program Entropy.
Technical Report CSL-78-3, Xerox Palo Alto Research Center, September, 1978.

[9] Richard E. Sweet and James G. Sandman, Jr.
Empirical Analysis of the Mesa Instruction Set.
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, 1982.

[10] Cheryl A. Wiecek.
A Case Study of Vax-11 Instruction Set Usage for Compiler Execution.
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, 1982.

[11] William A. Wulf.
Compilers and Computer Architecture.
Computer, July, 1981.