SPUR Lisp: Design and Implementation

Benjamin Zorn
Paul Hilfinger
Kinson Ho
James Larus

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley California

24 September 1987

This research was funded by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

Abstract

This document describes SPUR Lisp, a Common Lisp superset designed and implemented at U.C. Berkeley. Function calling sequences, system data structures, memory management policies, etc. are all described in detail. Reasons for the more important decisions are given. SPUR Lisp is implemented on BARB, a software simulator for SPUR hardware. In addition to describing the design of SPUR Lisp, this paper provides documentation for the BARB simulator, the SPUR Lisp compiler, and associated tools.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Origins

SPUR Lisp is a Common Lisp [Ste84] superset that will run on SPUR, a multiprocessor workstation being designed and built at U. C. Berkeley [HLE*85]. SPUR will consist of 6–12 processors connected to 32 megabytes of memory through a shared system bus. One of the many interesting features of SPUR is that each CPU contains special hardware for executing Lisp programs. This document describes the parts of the SPUR architecture that are relevant to the Lisp implementation. A more complete architectural description is provided in [THL*86].

SPUR Lisp is a derivative of Spice Lisp developed at Carnegie-Mellon University. The internal data structures and system organization of Spice Lisp are well documented [WFG85]. Because SPUR Lisp is derived from Spice Lisp, some of its structures are similar to those of Spice Lisp. We will refer the reader to the Spice Lisp document whenever the systems are essentially the same. On the other hand, Spice Lisp is implemented for the Perq, a microcoded workstation, so there are many differences between the two Lisp systems.[1]

SPUR Lisp is still being designed and implemented. We expect that as Common Lisp evolves, our Lisp will also evolve to keep up with the standard. An important part of the SPUR Lisp system—language support for multiprocessing—has undergone several design revisions and has not been implemented. This document will concentrate on uniprocessor SPUR Lisp. A later document will describe the details of our multiprocessor implementation.

Simulations have shown that SPUR workstations will be high performance Lisp machines [THL*86]. When SPUR hardware is available at Berkeley, we expect people to use SPUR Lisp for a variety of exploratory programming projects. The design of SPUR Lisp will continue to evolve as new ideas are tested and proven.

This document is written for two purposes. The first is to provide a single reference for information about the internal design of SPUR Lisp. To achieve this goal, we have divided the document into two parts. The main body of this report describes the internal data structures and conventions of SPUR Lisp. The appendices are a collection of short pieces that describe the current state of SPUR Lisp and its associated tools.

The second purpose of this document is to provide a written rationale for the decisions made as we designed SPUR Lisp. A rationale is important because although design decisions are implicitly documented by an implementation, the reasons for the decisions are often lost. We hope to collect our reasoning in this document for future Lisp designers and implementors.

[1]Readers familiar with the Spice Lisp internal design document [WFG85] will note that this document has a similar organization. Being quite familiar with the Spice Lisp documentation, we found it well organized and informative. We hope that this document is as usable as theirs.

1.2 Notation

We use standard terminology throughout this document. Lisp objects are usually manipulated indirectly through references to them. An *object reference* is a typed pointer to an object (we use the term *pointer* for short). In SPUR Lisp, object references have three parts: a 32-bit virtual memory *address*, a 6-bit *type tag*, and a 2-bit *generation number* that is used for garbage collection. We refer to the 32-bit portion of a pointer as the *data* of the pointer. The 8 bits of combined type tag and generation number are manipulated by the hardware as a unit, and are called the *typegen* of the pointer. We use the term *cell* to refer to a location in memory in which a pointer is stored.

Numbers in this document appear in several radices. With no radix specified, numbers are decimal. Numbers in other radices will be represented as in Common Lisp. For instance, hexadecimal numbers are prefixed with #x, octal numbers with #o, and binary numbers with #b.

In this document *words* are 32-bit quantities. *Doublewords* are 64-bit quantities. The numbering of bits and bytes within a word corresponds to the SPUR hardware description [Tay85]. Bits in a word are numbered least significant to most significant, right to left, from 0 to 31. Bytes are numbered right to left from 0 to 3.

1.3 Document Overview

Section 2 presents basic machine data types followed by the Lisp system data types. Section 3 describes the organization of system data structures. Section 4 presents function calling sequences. Section 5 describes SPUR Lisp memory management. Section 6 describes the runtime operation of SPUR Lisp. Section 7 presents an overview of the existing SPUR Lisp implementation. Each of the appendices provides specific information about a major component of SPUR Lisp.

2 Data Types

2.1 Machine Data Types

The SPUR CPU contains 40-bit tagged registers, as shown in figure 1. The 32-bit address/data portion is distinct from the 8-bit typegen portion. Separate instructions manipulate the 8-bit typegen portion as a unit.

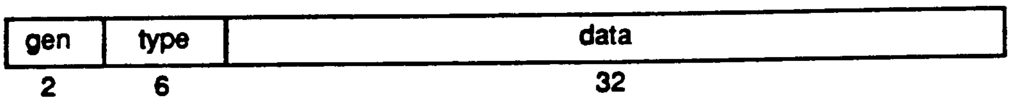


Figure 1: SPUR CPU Register Organization

The memory for SPUR is standard 32-bit memory. Pointers are placed in two consecutive 32-bit locations. The address portion is placed in the first 32 bits and the typegen is placed in the lower byte of the next 32 bits (figure 2).[2] All pointers in memory must be aligned to doubleword boundaries.

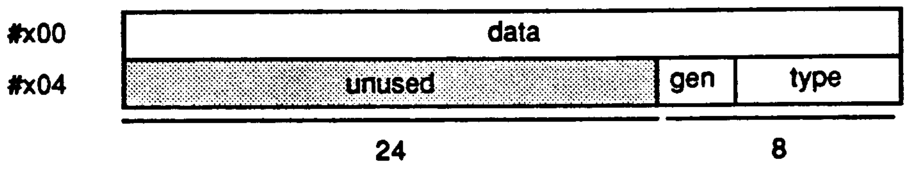


Figure 2: SPUR Lisp Pointers in Memory

Arithmetic in SPUR is handled both in the CPU and in a floating point coprocessor (FPU) . The SPUR CPU contains a 32-bit ALU capable of adding two 32-bit integers. Multiplication of integers and addition and multiplication of floating point numbers requires the FPU. The coprocessor performs IEEE standard floating point operations on three sizes of operands: short float (32-bit), long float (64-bit), and extended float (80-bit). The FPU coprocessor is described in detail in [ABPW85].

2.2 Lisp Data Types

Lisp data falls into two general categories: immediate values that fit in the 32-bit data portion of a pointer and indirect data that is pointed to by these bits. In SPUR Lisp,

[2]In the memory containing pointers, 3 bytes out every 8 bytes are unused. To avoid wasting this memory, a SPUR workstation would need 40-bit memory, which would require the design of custom memory boards. Because SPUR is a prototype research machine, funds and design time are limited. We chose to concentrate on more interesting architectural issues rather than custom memory design.

there are two immediate types: fixnum (the Lisp term for "small" integers) and character. All other data types are indirect. Objects of indirect data types are created in memory. Because pointers in memory are aligned to doubleword boundaries, all objects that contain pointers must be allocated on doubleword boundaries.

The upper bit of a SPUR type tag indicates whether a pointer references immediate or indirect data. This allows the hardware to check for immediate types by looking at a single bit. In addition to immediate and indirect types, some type tags are reserved for special operations like garbage collection. Table 1 lists all the type codes and their meanings. The following sections describe each data type in detail. Appendix A summarizes the SPUR Lisp data types.

NAME	CODE	CATEGORY	DESCRIPTION
fixnum	0	immediate	$-2^{31} \leq$ integer $\leq 2^{31} - 1$
character	16	immediate	ASCII character with extra bits
nil	32	indirect	Lisp `nil`
cons	33	indirect	list element
symbol	34	indirect	Lisp symbol
i-vector	36	indirect	vector of k-bit integers
string	37	indirect	character string
g-vector	38	indirect	vector of pointers
function	39	indirect	compiled function object
array	40	indirect	array header
gc-forward	41	special	marks forwarding pointers
undefined	43	special	marks unbound symbols
unused	44	special	impossible tag
cclosure	45	indirect	compiled closure object
short-float	56	indirect	32-bit IEEE short float
long-float	57	indirect	64-bit IEEE long float
extended-float	58	indirect	80-bit IEEE extended float
bignum	60	indirect	arbitrary sized integer
complex	61	indirect	complex number
ratio	62	indirect	rational number

Table 1: SPUR Lisp Data Type Codes

2.3 Immediate Data Types

There are two immediate data types in SPUR Lisp: fixnum, and character. Fixnums are 32-bit two's-complement integers. Characters are 32-bit objects that contain 8-bit ASCII characters with additional font and bits information as defined by Common Lisp. The format of character immediate data is illustrated in figure 3.

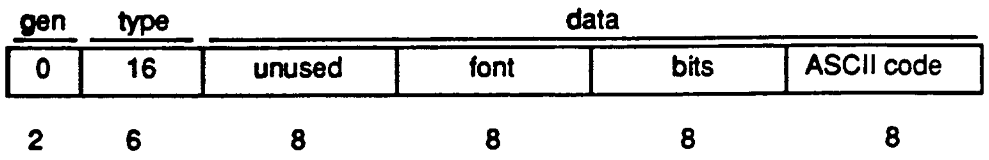


Figure 3: Format of SPUR Character Immediate Data

The generation number of all immediate types is generation 0, the oldest generation. They are given the oldest generation so that when they are stored into other objects, such as arrays, they will not cause an unnecessary generation trap (see section 5).

2.4 Symbol and List Data Types

Symbols are defined in Common Lisp to have 5 components: value, function definition, property list, print name, and package. In SPUR Lisp, symbols are contiguous blocks of 5 pointers corresponding to the 5 values. The layout of a symbol is shown in figure 4. Because each pointer occupies 8 bytes, symbols are 40 bytes long.

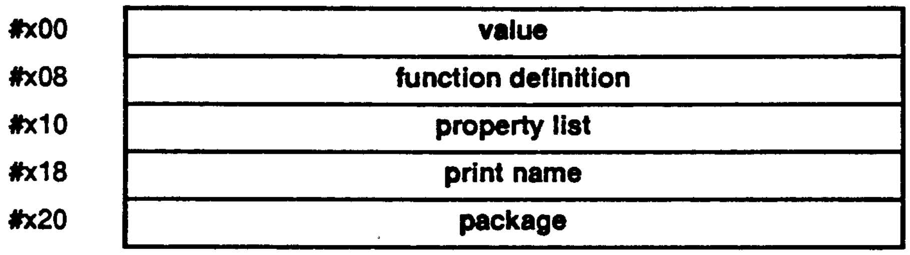


Figure 4: Structure of a Symbol Object

Lists are made from cons objects, which are pairs of pointers called the car and cdr, respectively. A cons occupies 16 bytes.

The special object `nil` is both a symbol and the empty list. In SPUR Lisp, `nil` has all the fields of a symbol. Except for its type code, `nil` really is a symbol. By definition, car of `nil` and cdr of `nil` are both equal to `nil`. By setting the value and function definition cell of the `nil` symbol to `nil`, `nil` can be accessed both as if it were a cons object or a symbol.

2.5 Integer Vectors

Integer vectors, also called i-vectors , store raw bits of data. Objects with the types i-vector, string, and bignum have the same internal structure and are distinguished by their type tags. Abstractly, i-vectors are vectors of untagged k-bit integers, where k is determined by a field in the header of the vector. This arrangement allows i-vectors to be used to represent vectors of bits, characters, and instructions.

The *access-type* of an i-vector determines how the bits in the vector are interpreted. The bit-width, k, of the integers in the vector is related to the access-type according to the formula:

$$k = 2^{access-type}.$$

For example, access-type 0 indicates the i-vector is a bit vector. Access-type 3 indicates a vector of bytes (e.g., a string), and access-type 5 indicates a vector of 32-bit quantities (e.g., instructions). SPUR Lisp supports access-type values of 0 through 5.

I-vectors are implemented as vectors of untagged 32-bit quantities with a header. Elements of i-vectors are numbered 0 through $n - 1$, where n is the number of elements in the vector. The vector elements are packed into the 32-bit quantities from right to left, low-index to high-index. I-vectors are aligned to doubleword boundaries. The padding needed to align an i-vector to a doubleword boundary is left unused and its contents are not defined. This space is considered part of the i-vector and is moved and copied as such.

The header of an i-vector contains 4 fields in 64 bits as shown in figure 5. The 24-bit length field contains the size of the entire i-vector, including the header and padding, in 32-bit words. The access type field has already been explained. The subtype field is unused in every i-vector structure except bignums and caller sets. The second 32-bit word of the header contains the number of elements in the vector. An element is a k-bit quantity, as defined by the i-vector access-type. Figure 6 illustrates an i-vector used to represent a bit vector.

2.6 Bignums and Strings

The i-vector data structure is used to represent many data types. Strings and bignums have the same internal format as i-vectors, but have different type tags.

Bignums are arbitrary size integers that arise when an arithmetic operation creates an integer larger than the largest machine integer. Bignums are represented as i-vectors with 8-bit elements (access-type 3). A bignum is stored as a vector of bytes, with the low order byte having index 0. The sequence of bytes represents a two's complement integer. The subtype field of bignums is used to redundantly encode the sign of the bignum, with a subtype of 0 indicating a non-negative integer, and a subtype of 1 indicating a negative integer.[3]

[3]Currently, SPUR Lisp performs arithmetic operations on bignums a byte at a time. If bignums were manipulated 32 bits at a time, as the hardware allows, bignum arithmetic would be many times faster than it currently is. [CM87]

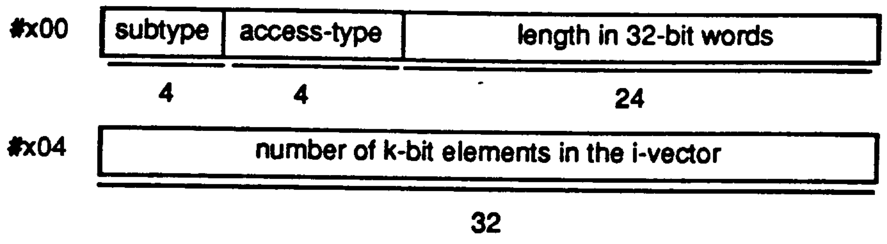


Figure 5: Structure of an I-Vector Header

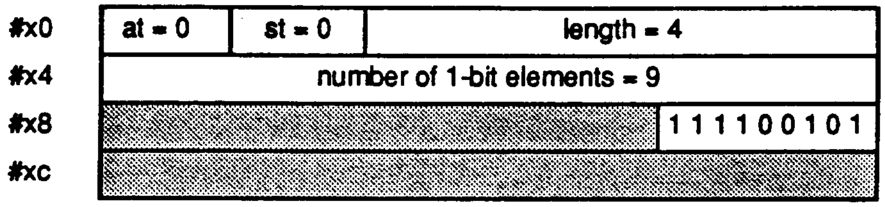


Figure 6: I-Vector Representing the Bit Vector #*101001111

A string is stored as a sequence of 8-bit ASCII characters in an i-vector. In SPUR Lisp, strings are NULL (0) terminated to follow the conventions of the UNIX and Sprite operating system library calls. The NULL byte appears immediately after the last character in the string and is inaccessible from Lisp programs.

2.7 General Vectors

General vectors, or g-vectors , are vectors of SPUR Lisp pointers. They are used to implement aggregate types including arrays, structures, and hashtables. A general vector consists of a 64-bit header attached to a vector of 64-bit pointers as shown in figure 7. The fields in the header are similar to the fields in the i-vector header. The header mark field is used to distinguish the g-vector header from elements of the g-vector for purposes of scanning through the g-vector. The two bits of the header mark field are always both zeros (see section 5.4.2). The subtype field of g-vectors is used to distinguish structures from other g-vectors. A 1 in the subtype field indicates that the g-vector represents a structure; all other g-vectors contain a 0 in the subtype field.

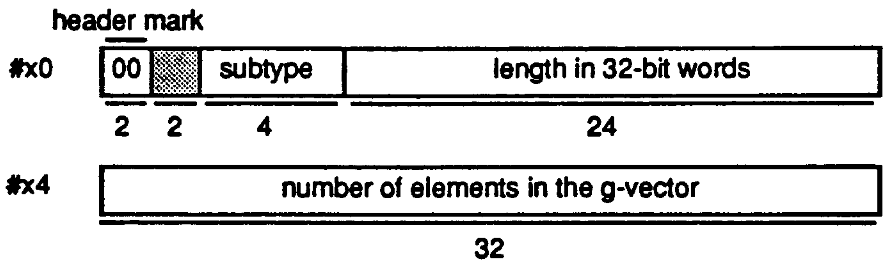


Figure 7: Structure of a G-Vector Header

2.8 Array Header Objects

An array header has the same form as a general vector and is used to implement multidimensional, resizable, and displaced arrays. SPUR Lisp arrays are implemented indirectly, using the same representation as Spice Lisp. Array pointers point to an array header object, which in turn points to the array data. The array header object has the structure of a g-vector with the elements of the g-vector describing the properties of the array. Figure 8 illustrates the structure of array header objects.

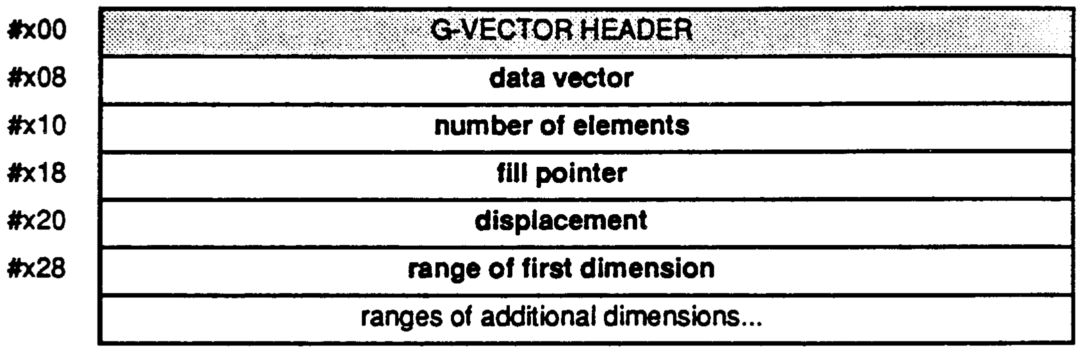


Figure 8: Structure of an Array Header Object

The fields of the array header object are:

data vector A SPUR Lisp pointer to the object containing the data for the array. This object may be an i-vector or g-vector.

number of elements The number of array elements that can be placed in the array (i.e., the number of elements in the object pointed to by *data vector*).

fill pointer The number of elements of the array that are considered to be in use.

displacement A number used to provide displaced arrays as defined by Common Lisp. This number is added to the calculated index into *data vector* before an item is accessed.

range of first dimension This is the index range of the first dimension of the array. Likewise for additional dimensions.

2.9 Numeric Data Types

SPUR Lisp has seven numeric data types: fixnum, bignum, short-float, long-float, extended-float,[4] complex, and ratio. fixnum and bignum types have already been described.

Floating point numbers adhere to the IEEE floating point arithmetic standard [IEE85]. The floating point data types are not immediate objects. Short floats are only 32-bits long and might appear to be good candidates for immediate objects. However, the SPUR processor executes floating point operations on a floating point coprocessor (FPU), and floating point data must be moved to the coprocessor before operations can be executed. Unfortunately, the SPUR CPU cannot transfer data from its registers to the FPU. If short floats were immediate values, the SPUR CPU would have to copy them to memory before sending them to the FPU. Because short floats are allocated in the same area as objects that contain pointers, they must be doubleword aligned and padded to 64 bits in memory. Long floats are also 64-bits long. Since short floats and long floats currently use the same amount of memory, and since the FPU already converts all floating point operands to extended precision before operating on them, we feel strongly that short floats should be eliminated from SPUR Lisp. Extended floats are 80-bits long, and are stored in 128 bits in memory. The bits used to doubleword align floating point numbers have undefined values. Figure 11 illustrates the memory formats of the floating point data types in SPUR Lisp.

The two other numeric data types in SPUR Lisp are complex and ratio. SPUR Lisp implements both of these types as pairs of numbers. For complex numbers, the pair contains the real and imaginary parts of the number, respectively. For rational numbers, the pair contains the numerator and denominator of the number in that order.

2.10 Data Types Related to Functions

Compiled functions in SPUR Lisp are implemented as two objects linked together: the object that contains the code for the function (the *code vector*) and the object that contains the constants for the function (the *constants vector*). A function pointer points to the code vector, which contains a pointer to the constants vector. An alternate design would be to put both the constants and the instructions into the same object. In one configuration, the constants would precede the code for the function. We chose not to do this because the start

[4]The extended-float type is not currently implemented, but we expect it to be easy to add at a later time.

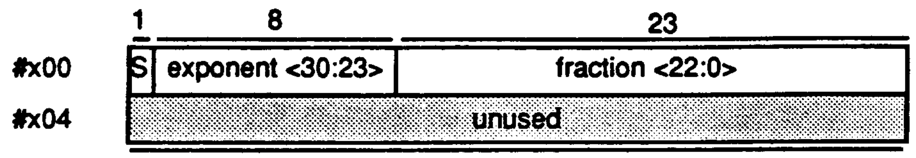


Figure 9: Memory format of a Short Float

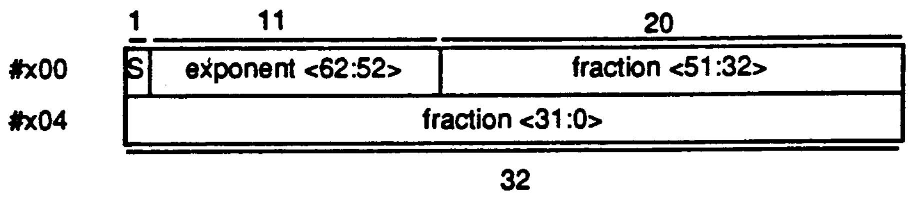


Figure 10: Memory format of a Long Float

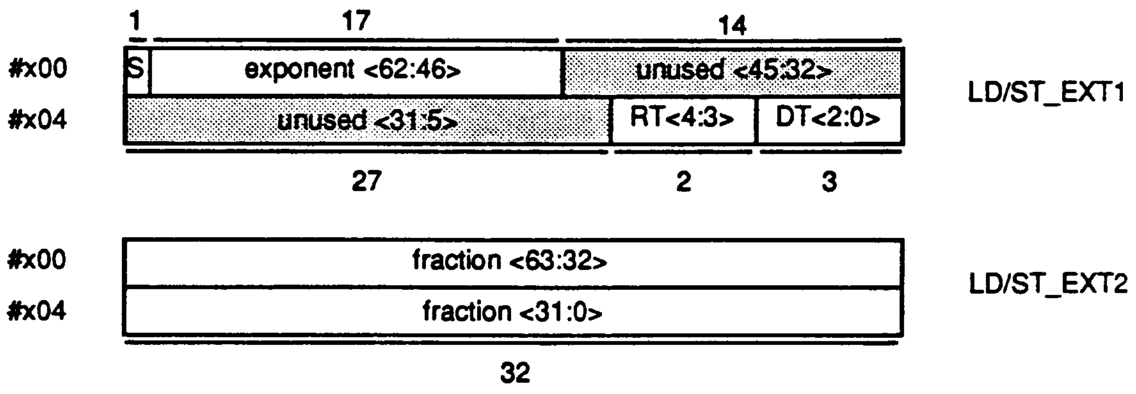
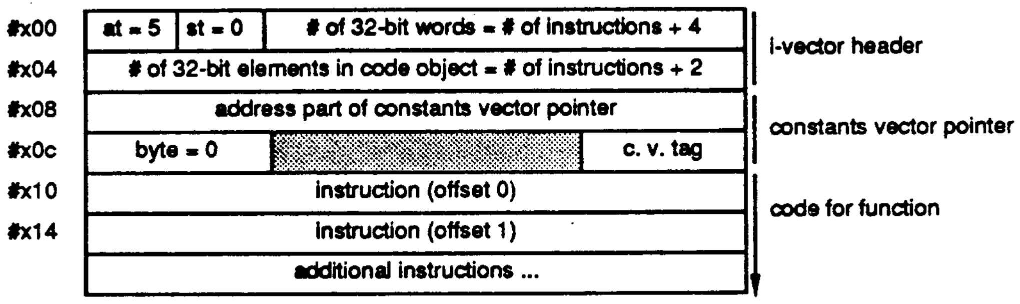


Figure 11: Memory format of an Extended Float

of the code for a function would not be a constant offset from the pointer to the function object. In another configuration, the constants would follow the code for the function. We decided against this alternative because the constants could end up far from the function pointer. SPUR only allows 14 bits of constant displacement in a load instruction. Large functions might make the function constants unreachable using a load with a constant offset. Our design uses existing object formats (i-vector and g-vector) with one minor modification. A single-object compiled function would require an entirely new object format since it would contain both tagged values (the constants) and untagged values (the instructions).

Code vectors are similar in structure to i-vectors, except that they also contain a tagged pointer to the constants vector. The header of a code vector has the same format as the header of other i-vectors with an access-type of 5 (element field width of 32). Other than the first two 32-bit elements of the code object (the pointer to the constants vector), each element in the code object is a SPUR instruction. Figure 12 illustrates the format of a code vector and figure 13 shows how the code vector and constants vector are linked.



#x00
#x04
#x08
#x0c
#x10
#x14
at = 5
st = 0
# of 32-bit words = # of instructions + 4
# of 32-bit elements in code object = # of instructions + 2
address part of constants vector pointer
byte = 0
c. v. tag
instruction (offset 0)
instruction (offset 1)
additional instructions ...
i-vector header
constants vector pointer
code for function

Figure 12: Structure of a Function Code Vector

The constants vector for a function is a g-vector that contains constants used in the code of the function. Figure 14 shows its structure. The first 4 elements of the g-vector are always interpreted in the same way.

argument number encoding A fixnum that encodes the argument discipline of the function. Bit 31 is 1 if the function is a special-form. Bit 30 is 1 of the function takes an `&rest` argument. Bits <29:17> indicate the number of local variables the function uses. Bits <16:8> indicate the minimum number of actual arguments the function can accept. Bits <7:0> indicate the maximum number of actual arguments the function can accept.

function name The symbol that names the function.

argument list The function's formal argument list.

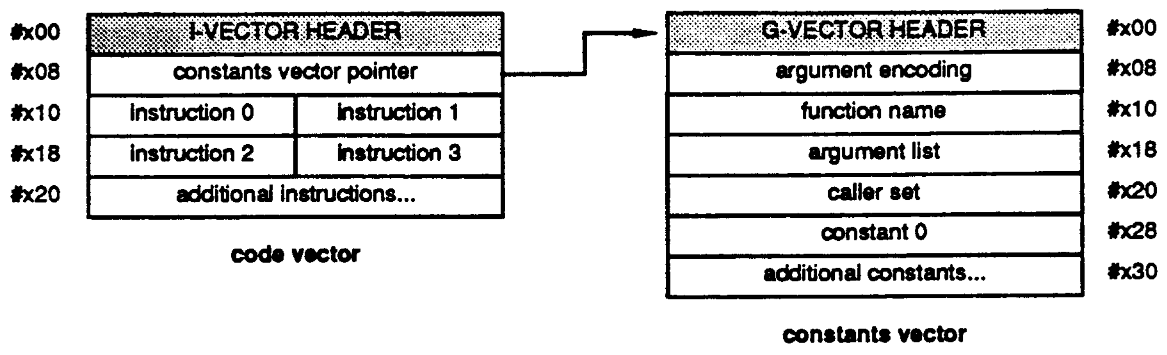


Figure 13: Linkage between Function Code Vector and Constants Vector

caller set A vector of addresses that identifies all the functions that call this function. Section 3.4.2 fully explains the structure and purpose of the caller set.

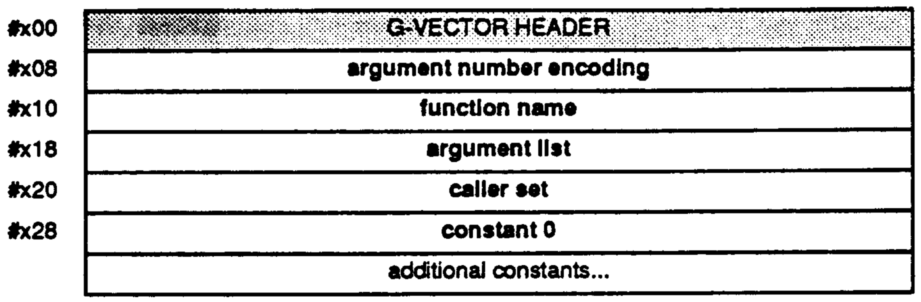


Figure 14: Structure of a Function Constants Vector

After the first four constants, the constants vector contains pointers to constants for the function, such as symbols, numbers, structures, etc.

One important property of function objects is that the beginning of a function object can be identified by scanning back from any instruction in the function. A debugger needs to do this when doing a backtrace. SPUR Lisp functions contain a marker that distinguishes their beginning. This marker is the fourth 32-bit word in the code vector and contains the typegen of the constants vector pointer. The upper byte of this word is always set to zero. Since SPUR contains no instructions with a zero opcode, this word signals the beginning of the instructions in the function object.

Another type related to functions is the cclosure type, used for compiled closures. A compiled closure is a pair of values. The first value is a g-vector containing the environment of the compiled closure. The environment for a compiled closure contains the values of the variables closed over in the environment. The second value in a cclosure is the function object of the compiled closure.

2.11 Special Data Types

There are several data type tags used for special purposes in SPUR Lisp. They are gc-forward, which is used during garbage collection to indicate that an object has been relocated; undefined, which is stored in the value cell of symbols that have not been given a value; and unused, which is used in comparisons requiring a tag value that never matches the tag on an actual object. The unused type was used for the endp comparison condition. endp checked for tag equality and would trap on any type other than cons or nil . We used the endp condition with the tag trap instruction to trap if an object was not a cons or nil. Because we did not want the instruction to trap in any other circumstance, the test compares the tag of the operand with unused, so the test will always fail.

3 System Organization in SPUR Lisp

3.1 Memory Organization of SPUR Lisp

SPUR Lisp operates in the 32-bit virtual address space as provided by the SPUR hardware and the Sprite operating system [OCD*87]. Figure 15 illustrates the memory organization of SPUR Lisp.

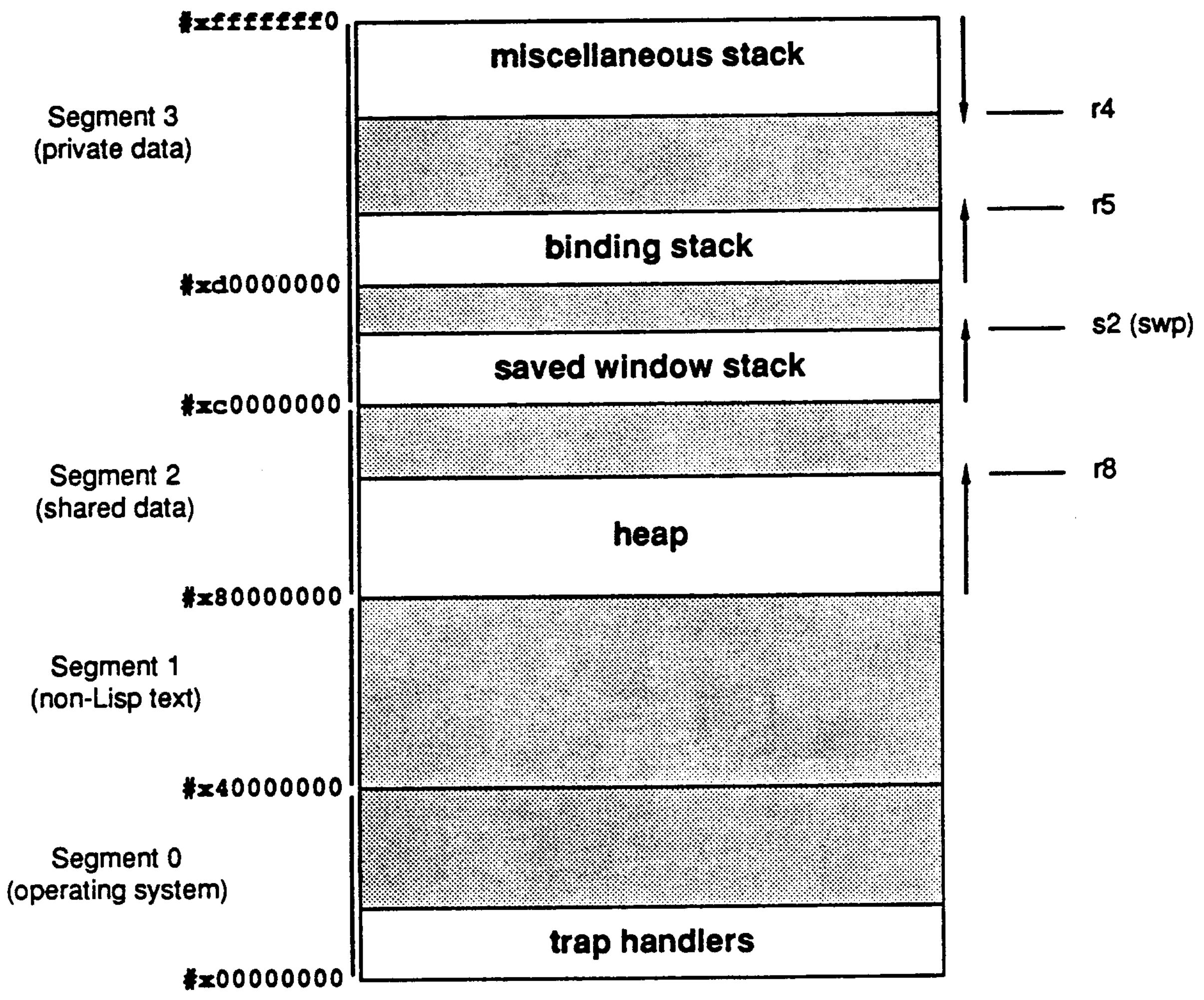


Figure 15: Memory Organization of SPUR Lisp

The SPRITE operating system divides the address space into 4 segments. One of the segments—the kernel segment—is reserved for the operating system kernel and is unusable by SPUR Lisp. Memory for the kernel segment resides in addresses #x00000000—#x3fffffff (segment 0). Segment 1 (#x40000000—#x7fffffff is used for non-Lisp user program text (such as a C program). Segment 2 (#x80000000—#xb7777777) is shared between all processes running Lisp and will contain the heap. Segment 3 (#xc0000000—#xffffffff) contains private data for a process, such as the runtime stack. From segments 2 and 3, uniprocessor SPUR Lisp allocates 4 memory regions: the heap, the saved window stack, the miscellaneous stack, and the binding stack. SPUR Lisp uses non-writable pages to delimit these regions so that memory accesses outside the allocated regions cause processor traps. The trap handler for these traps can invoke the garbage collector (as in the heap), grow a stack (as in the miscellaneous stack), or raise an error if that is preferred.

All SPUR Lisp objects are allocated in the heap. Because multiprocessor SPUR Lisp will share the heap between processes, we allocate the heap in segment 2 starting at #x80000000 and growing upward to a boundary determined by the memory allocation system.[5] Generally, the size of the heap is tens of megabytes. A global register (r8) points to the first free doubleword in the heap, which is subdivided for effective storage reclamation. Section 5 describes the structure of the heap in detail. Objects of all types are allocated from the same area of memory and not segregated by type as they are in some Lisp implementations [Moo85,WF84]. For this reason, all objects in the heap are doubleword aligned.

The second memory region in SPUR Lisp is the saved window stack . SPUR contains overlapping on-chip register windows similar to those in RISC [PS81] and SOAR [UBF*84]. SPUR contains eight register windows. Each register window contains six incoming argument registers, ten registers for local variables, and six outgoing argument registers which overlap with the incoming arguments of the next window (figure 16).

In addition to the 22 registers local to a particular function call, SPUR contains 10 global registers. In all, 32 registers are available to an executing function. The SPUR processor contains eight on-chip register windows. When the nesting level of function calls exceeds seven, a register window is copied to memory.[6] The saved window stack is a region of memory used to store overflowed register windows. The saved window stack is private to a process and stored in segment 3 beginning at #xc0000000 and grows toward larger addresses. A special register (s2) in the SPUR hardware points to the next free space to which a register window will be copied.

The miscellaneous stack is also private to a process and contains pointers to various items like arguments and local variables that will not fit in the on-chip registers, catch frames, etc. The miscellaneous stack contains only pointers and grows toward smaller addresses starting from #xfffffff0. The miscellaneous stack pointer is a global register (r4) that points to the first free location in the stack. The miscellaneous stack is used for the following purposes:

- It holds arguments to functions that will not fit in the six arguments registers provided by the hardware.

[5]In this discussion a region growing *upward* means that it grows to larger addresses.

[6]Only seven windows can be used because the trap handler that copies register windows to memory must also have a window to operate in.

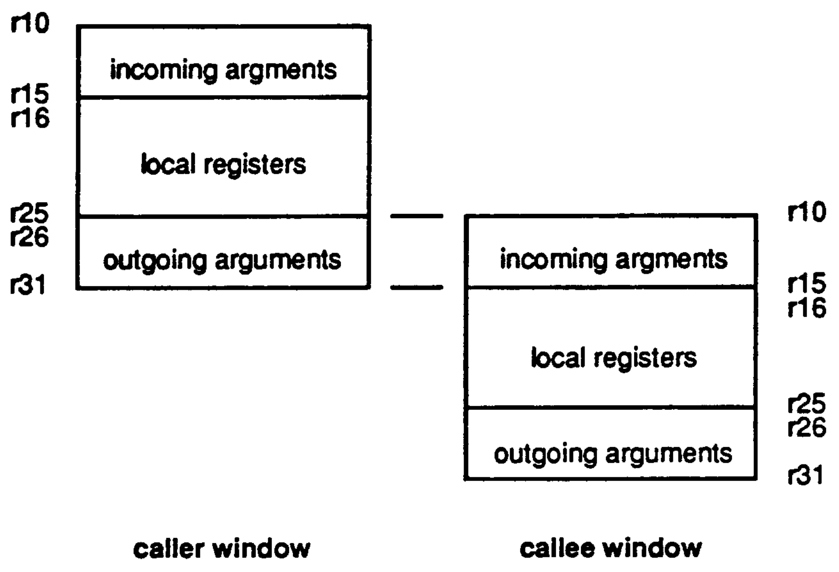


Figure 16: Overlapping Register Window Design on the SPUR Processor

- Returned multiple values that will not fit in the argument registers are stored on this stack.
- It holds compiler temporaries that do not fit into the ten local registers.
- Catch frames, described in section 4, are also placed in the miscellaneous stack.

The final memory region is used for the bindings of special variables. The binding stack is a stack of pairs of pointers that grows toward higher addresses starting at #xd0000000. Each pair consists of a symbol and the value associated with that symbol. The binding stack pointer is a global register (r5) that points to its first free element. SPUR implements special variables with shallow binding. The value cell of a symbol contains the current binding of a special variable. The binding stack contains saved bindings of rebound special variables. Multiprocessor SPUR Lisp may use deep binding to implement dynamic binding. In that case, the binding stack and binding stack pointer may be unnecessary.

3.2 SPUR Lisp Register Assignment

The SPUR hardware provides ten global general registers and six special registers. Tables 2 and 3 summarize how the SPUR Lisp system uses these registers.[7]

[7]Some global registers are unusual in that they point to objects that are not valid SPUR Lisp objects (as defined in section 2). We chose to give these registers fixnum tags so they appear to be integers for purposes of addition and garbage collection relocation.

NAME	DESCRIPTION
r0	fixnum zero. Fixed by the hardware.
r1	Lisp nil .
r2	Reserved for compiler temporary values.
r3	System constants vector. A g-vector containing additional system constants and variables.
r4	Miscellaneous stack pointer. Address of the next free cell on the miscellaneous stack (from #xfffffff0 down). fixnum tag.
r5	Binding stack pointer. Address of the next free cell on the binding stack (from #xd0000000 up). fixnum tag.
r6	Catch frame pointer. Address of the current catch frame (initially nil). fixnum tag when non-nil.
r7	Reserved for compiler temporary values.
r8	Heap pointer. Address of the next free cell on the heap (from #x80000000 up). Because most objects allocated in the heap are cons objects, the heap pointer has the cons type tag.
r9	Reserved for trap handling by the operating system.

Table 2: SPUR Lisp Global Register Assignments

Registers 2 and 7 are reserved by the compiler so that temporary values can be spilled to memory and retrieved from memory without using any of a function's local registers. Temporaries and local variables that do not fit in the registers are stored in memory or "spilled". SPUR is a load/store architecture, which means that all operands to instructions must be loaded into registers. For SPUR to operate on spilled temporaries, they must be placed in registers. Two registers are needed because some instructions require two operands, and if they are both spilled temporaries, then they must both be loaded into registers at the same time.

Register 9 is used by the window trap handlers. The window overflow trap handler needs to save the registers in the previous window. To do this, the trap handler changes the current window during its execution. Register 9 is used to save a local value that will remain constant while the window stack pointer changes. Register 9 is not entirely necessary, but the window trap handlers would execute one instruction slower if a local register were used instead. If a better use for register 9 becomes clear, we will use it for that purpose.

3.3 Additional System Constants

As is specified in table 2, register 3 points to a general vector containing additional system variables that did not fit in the global registers. The *system constants vector* contains a variety of variables defined when the system is created. The name "system constants" was

NAME	DESCRIPTION
s0 (upsw)	32-bit user program status word. See the SPUR hardware description for details [Tay85].
s1 (cwp)	Current window pointer. 3-bit register that indicates which register window is in use.
s2 (swp)	Saved window pointer. 32-bit address of the next free area to which overflowed register windows will be copied (from #xc0000000 up). The hardware compares bits <9:7> of this register against s1 (the current window pointer) to determine when a window overflow or underflow will occur.
s3 (pc)	32-bit address of the currently executing instruction.
s4 (fpu pc)	FPU program counter. 32-bit address of last FPU instruction issued.

Table 3: SPUR Special Register Assignments

poorly chosen, for as can be seen, many of the "constants" are actually variables. Table 4 describes each of the constants and variables.

3.4 System Structures Associated with Functions

When we discuss function calls, we will use the term *caller* to refer to the function making the call, and *callee* to refer to the function being called. Many Lisp implementations call functions indirectly through the function definition cell of a symbol [Moo85,WFG85]. Insteading of placing the address of a callee directly in a `call` instruction, the address of the callee is computed each time a call is made. The address is retrieved from the function definition cell of the symbol that names the callee. Each call requires one or two additional loads to determine the actual address of the callee.

This implementation has several advantages. Because all calls are indirect, when a new definition for a function is provided, only one value must be changed (the function definition cell of the symbol naming the callee) and all callers will invoke the new version of the callee. Similarly, keeping track of unresolved references is easy with indirect function calls. Since all references to functions go through the definition cell of a symbol, an undefined value indicates an undefined function. The main disadvantage of indirect function calls is the overhead to find the address of the function on every function call. Since function calls are frequent in Lisp and SPUR calls are fast, several cycles per function call is a significant overhead.

SPUR Lisp implements function calls directly. Each call instruction contains the actual address of the function being called. This implementation eliminates the overhead associated with indirection, but requires additional information to be maintained for correct function definition and redefinition. We have looked at the costs and performance benefits

NAME	INDEX	DESCRIPTION
unresolved_references	0	The unresolved reference list is a data structure maintained by the loader (section 3.4.1).
allocation variables	1–18	Indices into various allocation spaces maintained by the memory management software (section 5).
package_names	19	The symbol `lisp:*package-names*`, whose value is the root pointer to all objects in the SPUR Lisp system.
lisp_true	22	The symbol `lisp::t` returned from many predicates.
lisp_nil	23	A redundant copy of `nil`, the value in register 1. This copy is used to initialize register 1 at system startup time.
find_symbol_address	25	The symbol `lisp:find-symbol`. The function `find-symbol` returns the symbol associated with a particular print name. `find-symbol` is used to locate symbols at system creation time.
interpreter_address	26	The address of the function `%sp-internal-apply` that implements the SPUR Lisp interpreter. fixnum tag.
bit_table	27–58	This is a table of positive powers of two. Used for setting bits in a bit vector (e.g., Common Lisp `sbit`).
current_space	59	A fixnum indicating which space memory is currently being allocated in. Used for generation garbage collection.
deep_binding_marker	60	A cons object, `'(0 . 0)`, used to mark the beginning of special bindings for a function in the deep binding implementation of SPUR Lisp.

Table 4: Contents of the System Constants Vector

of direct function calls, and the results are reported in [ZH87]. Here, we present the details of our implementation.

3.4.1 The Unresolved Reference List

A Lisp function, `f1`, can contain a call to another function, `f2`, that has not been defined. When `f2` is defined, the unresolved reference to `f2` in `f1` must be resolved.

Constant 0 in the system constants vector contains a pointer to the *unresolved reference list*. The unresolved reference list specifies all functions that have been referred to and not yet defined, and for each of these, lists the locations of references made to these functions. After a function is defined (using `load` or `defun`), the unresolved reference list is checked to see if there are any unresolved references to the function. If there are unresolved references, these are resolved by modifying the `call` instructions to call the correct address, and the unresolved reference list is updated. Before the reference is resolved, the `call` instruction contains a call to an error routine that reports the unresolved reference. Figure 17 summarizes the structure of the unresolved reference list.

```
((callee-symbol1
   ((caller-function1 (offset1 offset2 ... offsetn))
    (caller-function2 (offset1 offset2 ... offsetn))
    ...))
 (callee-symbol2
    ((another-caller-function1 (offset1 ... offsetn))
     (another-caller-function2 (offset1 ... offsetn))
    ...))
 ...)
```

Figure 17: Structure of the Unresolved Reference List

The unresolved reference list is a list of lists. The `car` of each sublist is a symbol, called the *callee symbol*, and the `cdr` of each sublist is another list of lists, called the *caller list* for the particular callee. The callee symbol names a function that has not been defined yet. The caller list is a list containing information about all the callers of the callee symbol. When the callee symbol is defined, calls in all the functions in the caller list are updated to reflect the actual address of the callee function. Each sublist of the caller list contains information about one function that calls the callee function. The `car` of each sublist is the caller function object and the `cdr` of each sublist is a list of offsets into the caller function where references to the callee exist. The offsets are instruction offsets from the first instruction, which is considered to be at offset 0.

3.4.2 Caller Sets

When a function is redefined, all calls to the old instance of the function must be updated. This implies that each function must maintain information about all its callers. We store this information in a data structure called the *caller set*, which is pointed to from the constants vector of every function. Abstractly, the caller set records the addresses of all functions that call a particular function. Unlike the unresolved reference data structure, the exact location of every call to a function is not recorded. Only the caller function address is recorded; to update calls in a particular function, all the instructions in the caller are scanned. The addresses of individual calls are not stored because the caller sets would then contain information about every call instruction in the Lisp system and be quite large. Our implementation trades off the space required to store locations of the individual instructions with the time required to scan a function object. Since defining functions is a relatively infrequent operation, we consider this a good trade off.

The caller sets are straightforward to maintain. When a function is defined, its instructions are scanned to record the functions it calls. If it calls a function that is already defined, then the function being defined is added to the caller set of the function it calls. If a call is to an undefined function, that call is recorded in the unresolved reference list. Later, when the reference is resolved, the calling function is added to the caller set of the function it calls.

When a function is redefined, references to the old definition are updated. The caller set of the old definition identifies all functions that contain obsolete calls. The instructions of each of these caller functions are scanned, and calls to the old function are updated. In SPUR Lisp, function objects are garbage collected like other objects. When all references, such as pointers or call instructions, to a particular object are deleted, the storage used by the object can be reclaimed.

Caller sets in SPUR Lisp are implemented as i-vectors of 32-bit function addresses. We set the subtype field of the caller set to 2 so we can easily measure the space allocated to caller sets. The first 32-bit slot of the caller set i-vector is used to store the current number of callers in the caller set. The rest of the elements contain 32-bit addresses of function code vectors. Caller sets are initially allocated with five slots for caller addresses. When the initial size of the caller set is exceeded, a new caller set i-vector is allocated, and the entries from the old i-vector are copied to it. The size of the new caller set i-vector is determined as a function of the size of the old caller set i-vector. Our current policy is to double the i-vector size until it contains several hundred callers, and then to increase it linearly by several hundred callers at a time. Measurements indicate that most functions have less than five callers. However, some functions, like `error`, have hundreds of callers.

4 SPUR Lisp Function Calling Conventions

4.1 Argument Register Assignment

Each register window in the SPUR architecture contains 6 registers for incoming arguments, 10 registers for local variables, and 6 registers for outgoing arguments that overlap with the incoming arguments of the function being called. Table 5 illustrates how SPUR Lisp assigns the incoming (r10–r16) and outgoing (r26–r31) argument registers.

INCOMING	OUTGOING	DESCRIPTION
r10	r26	Return address. Set by the hardware.
r11	r27	Number of actual arguments.
r12	r28	First actual argument.
r13	r29	Second actual argument.
r14	r30	Third actual argument.
r15	r31	Fourth actual argument if the number of actual arguments is exactly four. If there are more actual arguments, this register points to the fourth argument (located on the miscellaneous stack) and has a `fixnum` tag.

Table 5: SPUR Lisp Argument Register Assignments

SPUR has a limited number of argument registers. To handle functions with more than four actual arguments, we use the last argument register to point to the remainder of the actual arguments, which are stored in memory on the miscellaneous stack. Arguments are stacked consecutively on the miscellaneous stack in the same direction that it grows; later arguments are placed at smaller stack addresses. Figure 18 illustrates the argument registers and miscellaneous stack for a function call with five arguments.

4.2 Function Calling Sequence

When calling a function with fewer than four arguments, the calling function sets the number of arguments, moves the arguments into the outgoing argument registers, saves the miscellaneous stack pointer, and executes the call. After the function returns, the miscellaneous stack pointer is restored. For example, the call `(foo 1 2 3)` would result in the following sequence:[8]

[8]The assembly language used in this document is a simplified version of the SPUR assembly language (SAS) documented elsewhere [Hil87]. The sequences presented here will not work in the actual assembler. Furthermore, the sequences are provided to illustrate a point and may not be the optimal sequence or the actual sequence used in many cases.

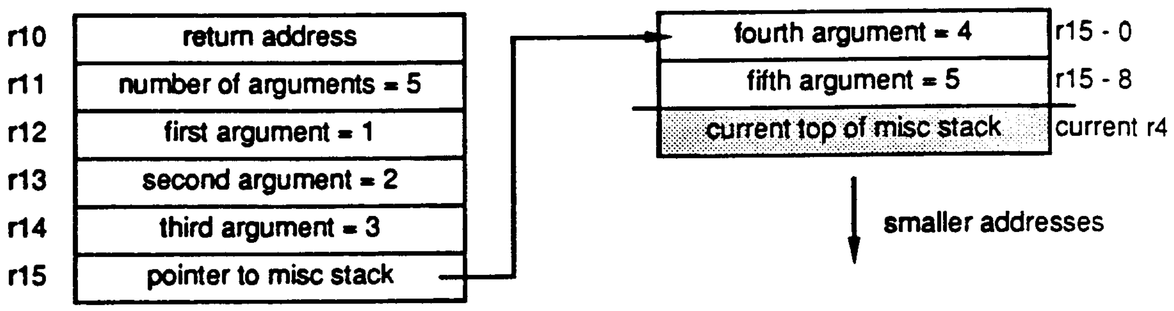


Figure 18: Argument Registers and Miscellaneous Stack for the call (f 1 2 3 4 5)

```
move    r27, 3          ; number of arguments
move    r28, 1          ; arg 1
move    r29, 2          ; arg 2
move    r30, 3          ; arg 3
move    rTMP, r4        ; save misc stack pointer
call    foo+16          ; (see below)
nop                     ; call is a two-cycle instruction
move    r4, rTMP        ; restore misc stack pointer
```

The offset in the call is 16 bytes from the function because we need to jump over the 8 byte function object header and the 8 byte constant vector pointer to get to the first instruction of the function. A slightly more complicated example involves a function with six arguments. The call (foo 1 2 3 4 5 6) would result in the following instruction sequence:

```
move    rTMP, r4        ; save misc stack pointer
move    r27, 6          ; number of arguments
move    r28, 1          ; arg 1
move    r29, 2          ; arg 2
move    r30, 3          ; arg 3
move    r31, r4         ; set up additional args
store   4, r31(0)       ; arg 4
store   5, r31(-8)      ; arg 5
store   6, r31(-16)     ; arg 6
add     r4, r4, 24      ; bump misc stack pointer
call    foo+16
nop
move    r4, rTMP        ; restore misc stack pointer
```

We follow the convention that a function's caller saves the miscellaneous stack pointer before a call and restores it after a call because a function that returns multiple values uses the miscellaneous stack for values that do not fit in the registers. A caller cannot know whether or not the callee will return multiple values. Functions that ignore multiple values restore the miscellaneous stack pointer when the call returns. Functions that use the multiple values restore the saved miscellaneous stack pointer after using the values on the stack.

Recall that SPUR Lisp uses direct function calls (section 3.4.2). Placing the address of the callee in the call instruction works well for conventional function calls. However, with `funcall` and `apply`, the address of the callee is determined at runtime. The call `(funcall X)` would result in the following instructions:

```
        load    rX, X                       ; place X in register rX
        nop                                 ; load is two cycle instr
        move    r27, 0                      ; number of arguments
        tag_cmp neq_tag, rX, symbol, L1     ; symbol?
        load    rX, rX(symbol-value-cell)   ; load value
L1:     tag_cmp neq_tag, rX, cclosure, L2   ; compiled closure?
        load    r7, rX(0)                   ; save closure environment
        nop
        load    rX, rX(8)                   ; load closure function
        nop
L2:     tag_cmp_trap neq_tag, rX, function, ERROR ; function?
        call_reg r10, rX, 16                ; call_reg is not provided by
                                            ; the hardware (see below)
```

The SPUR hardware does not provide a call instruction that will transfer control to an address in a register (e.g., `call_reg`). Instead, the hardware provides a jump to an address provided in a register (`jump_reg`). `jump_reg` can be used to get the effect of `call_reg` if the register window pointers are properly manipulated. Below is a code sequence that achieves the effect of `call_reg` using `jump_reg`.

```
;;; original code sequence with call_reg
        move     r11, 0                 ; zero arguments
        load     rFUNC, FUNC_ADDRESS    ; load the function address
        nop
        call_reg rFUNC
        nop                             ; call_reg is two-cycle

;;; equivalent code sequence using jump_reg
        move     r11, 0
        load     rFUNC, FUNC_ADDRESS    ; rFUNC is a global register
        nop
        call     label1                 ; change cwp
        nop
        jump     done                   ; 'return' returns here
        nop
label1: jump_reg rFUNC                  ; jump to correct address
```

```
        nop                             ; jump_reg is two-cycle
done:
```

A call instruction to a nearby address is used to allocate another register window stack frame and then the jump_reg modifies the program counter. Also note that the register containing the jump_reg address must be global since it is used in two different register frames, and the shared in/out registers are needed for passing arguments to the function being called.

4.3 Function Entry Sequence

Common Lisp provides several different parameter mechanisms, including &rest, &optional, and &keyword arguments and SPUR Lisp implements all of these. The exact entry sequence to a function depends on the function's formal parameters. If a function uses any constants, which are stored in the constants vector, then we load the constants vector pointer into a local register (r16) with the first three instructions of the function:

```
        rd_spec r17, s3         ; put the pc (s3) into r17
        load    r16, r17(-8)    ; load constants vector into r16
        nop
```

Functions use the miscellaneous stack for temporary values and compiler locals that do not fit in the local registers. The *frame pointer* points to the part of the miscellaneous stack that contains these values. After setting up the constants vector, the function creates a frame pointer in register 17 by saving the current value of the miscellaneous stack pointer with the following instruction:

```
        move    r17, r4         ; save frame pointer in r17
```

All functions check the number of actual arguments against the number of formal arguments. Functions without any &rest or &optional arguments test for an exact number of actual arguments. Functions with an &rest argument test for a minimum number of actual arguments. Functions with &optional arguments test that the number of actual arguments falls within the minimum and maximum number of actuals allowed.

Functions with &optional arguments also use the number of actuals to branch to code that initializes omitted values. Functions with &rest arguments create a list of the arguments and change the number of actuals accordingly. &keyword arguments are parsed by a runtime primitive that scans the actual arguments looking for keyword–value pairs. We tried parsing &keyword arguments with in-line code, but this approach expanded the size of functions with many keywords—in particular structure constructors.

Parsing of argument lists is done in two ways. If a function has an &rest argument in addition to &keyword arguments, the actuals are made into a list and passed to the Spice Lisp keyword parsing routines. In the case in which a function has &keyword arguments but no &rest arguments, parsing of the actual arguments is done by hand-coded assembly routines that scan the arguments in registers and memory. This approach is faster than the previous one because it avoids construction of the &rest argument.

4.4 Function Return Sequence

Function returns occur in three stages. First, the return value(s) are moved to the incoming argument registers. Register 11 indicates the number of values returned. Register 12 contains the first return value. Additional registers store additional values if multiple values are returned. Multiple values that require more than 4 registers are discussed in detail in section 4.5.

After the return value is set, special variable bindings are undone. If the number of bindings is known, then that number of unbind operations are generated. If the number of bindings is unknown, then the binding stack is unwound to the point saved on function entry.

The last stage of a function return is to execute the return instruction and return to the address of the calling instruction (stored in r10) + 8, which skips over the calling instruction and the instruction after it. Like **call**, **return** is a two-cycle instruction. For example, consider a function that returns the value 24.

```
move    r11, 1          ; return 1 value
move    r12, 24         ; set return value
return  r10, 8          ; return to saved pc + 8
nop                     ; two-cycle instruction
```

4.5 Multiple Values

Multiple values are created and reside, for the most part, in the outgoing argument registers of the function that creates them. They are stored in the same format as actual arguments. To return a multiple value, the function must copy the 5 outgoing arguments to the incoming arguments. For example, to return the value returned by the call (foo), we execute the following code:

```
move    r11, 0          ; number of args to foo
move    rTMP, r4        ; save misc stack pointer
call    foo+16
nop
move    r11, r27        ; copy back number of values
move    r12, r28        ; copy back all values from
move    r13, r29        ;   outgoing to incoming arguments
move    r14, r30
move    r15, r31        ; do NOT restore the misc stack (r4)
return  r10, 8
```

```
        nop
```

There are advantages to having identical formats for multiple return values an multi-argument function calls. `multiple-value-list` takes the multiple value that is returned and passes it as an argument list to `list`. For example, `(multiple-value-list (foo))` produces:

```
        move    r27, 0          ; number of args to foo
        move    rTMP, r4        ; save misc stack pointer
        call    foo+16
        nop
        call    list+16         ; make returned values into a list
        nop
        move    r4, rTMP        ; restore misc stack only after
                                ;   calling list
```

The next example illustrates how recovering values from multiple values is done. The form `multiple-value-setq` assigns variables to the values returned. If there are more variables than values, the remaining variables are set to `nil` . The following code sequence is produced for the form: `(multiple-value-setq (A B) (foo))`.

```
        move    r27, 0                  ; number of arguments to foo
        move    rTMP, r4                ; save misc stack pointer
        call    foo+16
        nop
        move    rA, nil
        cmp     lt, r27, 1, val2        ; 1 value?
        nop
        move    rA, r28                 ; use 1st returned value
val2:   move    rB, nil
        cmp     lt, r27, 2, done        ; 2 values?
        nop
        move    rB, r29
done:   move    r4, rTMP                ; restore misc stack pointer
```

There are three Common Lisp constructs in which multiple values must be saved and restored: `multiple-value-call`, `multiple-value-prog1`, and `unwind-protect`. SPUR Lisp provides the primitive functions `rt-mv-spill` and `rt-mv-unspill` to save and restore multiple values.[9] A multiple value is saved by calling the `rt-mv-spill` runtime routine. `rt-mv-spill` takes a multiple value and saves it on the miscellaneous stack, returning a pointer to the spilled multiple value. `rt-mv-unspill` takes any number of pointers to spilled multiple values and copies them back into the registers as a single multiple value.

[9]A list of the names and purposes of all the SPUR Lisp primitive functions is provided in section 6.

rt-mv-unspill also resets the miscellaneous stack pointer to point just after the new multiple value. For example the form, (multiple-value-call #'foo (bar) (bar-none)) results in the following code:

```
move    r27, 0            ; number of args to bar
move    rTMP, r4          ; save misc stack pointer
call    bar+16
nop
call    rt-mv-spill+16    ; save 1st multiple value
nop
move    rSAVE, r28        ; save 1st multiple value pointer
move    r27, 0            ; number of args to bar-none
call    bar-none+16
nop
call    rt-mv-spill+16    ; save 2nd multiple value
nop
move    r27, 2            ; number of args to rt-mv-unspill
move    r29, r28          ; 2nd arg is return value of 2nd
                          ;   call to rt-mv-spill
move    r28, rSAVE        ; first arg is saved mv pointer
call    rt-mv-unspill+16 ; unspill 2 multiple values
nop
call    foo+16            ; call foo
```

4.6 Calling Interpreted Functions

An interpreted function is given a dummy compiled function body that loads the interpreted form and calls a runtime primitive that calls the SPUR Lisp function %sp-internal-apply, which implements the interpreter. The dummy function body loads the form to interpret into r18 and then calls the runtime primitive rt-invoke-interpreter , which changes the current window pointer, recovers the form to interpret, and sets up a call to the function %sp-internal-apply. The first argument to %sp-internal-apply is the form to interpret and the remaining arguments are the arguments passed to the dummy compiled function body. To recover these arguments, rt-invoke-interpreter executes mostly in the window frame of the dummy body that calls it. The dummy body does not even have to perform a return because rt-invoke-interpreter does the return for it. Since the dummy compiled function body is duplicated for every interpreted function, we attempted to make this dummy body as small as possible.

The dummy function is created when the interpreted function is defined. Since both compiled and interpreted function objects have the same structure, interpreted functions can be treated exactly the same as compiled functions. The dummy function body consists of the following code:

```
rd_spec r17, s3           ; read the pc
```

```
load    r16, r17(-8)     ; load the constants vector
nop
load_p  r18, r16(40)     ; load constant 0: the form to interpret
nop

;; This is a pretty funny looking call.  rt-invoke-interpreter
;; shuffles the arguments around (including the "argument" in
;; r18) and jumps to the interpreter, which uses this function's
;; register window frame.  Note no return from this function.

call    rt-invoke-interpreter+16
nop
```

4.7 Catch and Throw

Catch frames are stored on the miscellaneous stack. The global register r6 points to the first catch frame and each catch frame points to the preceding catch frame. The bottom of the stack of catch frames is indicated by nil , which is the initial value of r6. A catch frame contains seven values as illustrated in figure 19.

#x00	**previous catch frame**
#x08	**tag of catch**
#x10	**saved pc**
#x18	**saved binding stack pointer (r5)**
#x20	**saved miscellaneous stack pointer (r4)**
#x28	**saved window stack pointer (swp)**
#x30	**saved current window pointer (cwp)**

Figure 19: Structure of SPUR Lisp Catch Frame

The contents of a catch frame are:

previous catch frame The address of the previous catch frame on the miscellaneous stack (with fixnum tag) or nil, which indicates no more catch frames.

tag of catch Compared using eq with the tag from the throw.

saved pc The address of the instruction to execute after the catch occurs. fixnum tag.

saved r5 The binding stack pointer at the time of the catch. fixnum tag.

saved r4 Value of the miscellaneous stack pointer at the time of the catch. fixnum tag.

saved swp Saved value of the window stack pointer (s2). fixnum tag.

saved cwp Saved current window pointer (s1). fixnum tag.

When `catch` is invoked, a new catch frame is allocated, the proper values are stored in it, and r6 is updated. `throw` invokes a runtime primitive with its value and a tag. Invoking this routine is tricky when the value being thrown is a multiple value. Instead of shifting the return values in the registers to make the tag the first value, we make the throw tag the last element of the augmented thrown value. Even so, much in-line code is needed to put the tag in the correct place for all sizes of multiple values.

When throw is invoked, two special symbols are used to implement `unwind-protect`, which catches every throw, executes some code, and propagates the throw upward. The unbound symbol `lisp::%catch-all-object` is used to indicate a catch frame generated by an `unwind-protect`. Any catch frame with this symbol as a tag will catch all throws. The symbol `lisp::%sp-internal-throw-tag` is used to store the value of the real throw tag when executing an `unwind-protect` generated catch. This symbol is initially unbound, and is rebound by each `unwind-protect` catch frame. SPUR Lisp follows Spice Lisp conventions with respect to the `%catch-all-object` symbol used to indicate an `unwind-protect`. The major difference is that SPUR Lisp does not do a `throw` to clear a catch frame from an `unwind-protect` off the stack, but rather changes the catch frame pointer (r6) and miscellaneous stack pointer (r4) explicitly.

5 Memory Management in SPUR Lisp

5.1 Generation Garbage Collection

SPUR Lisp reclaims unused memory with an algorithm called generation garbage collection, first proposed by Lieberman and Hewitt [LH83]. The major advantage of generation garbage collection is that it allows small pieces of the address space to be collected separately and avoids poor paging performance caused by collecting the entire address space. Generation garbage collection divides the heap into several spaces and places objects in the spaces according to how long the object has persisted in memory. Objects are allocated in the youngest space (called *newspace*), and if they remain in use long enough (e.g., the time between successive garbage collections), they are promoted to a space in which older objects are stored. Objects in newspace are garbage collected most frequently. Since newspace occupies a small fraction of the entire heap, these garbage collections are rapid and non-disruptive. Objects in older spaces are collected less frequently. Generation garbage collection takes advantage of the empirical property that most newly allocated objects become garbage quickly, while objects that have persisted for a period of time are more likely to remain in use.[10]

Generation garbage collection imposes no constraints on where pointers can point in memory. To make generation garbage collection work, all the accessible objects in a particular space need to be identifiable. In particular, all pointers that point from an older space to a younger space must be remembered. This set of pointers is called the *remembered list.* This section will describe the generation garbage collection implemented in SPUR Lisp in detail and focus on some of the problems anticipated in implementing multiprocessor generation garbage collection.

5.2 Hardware Support for Memory Management

As mentioned in section 2.1, SPUR general registers contain two bits in the 8-bit typegen field that are used for memory management. These two bits, the generation number, are used to indicate the space of the object the pointer refers to. SPUR provides a hardware trap on pointer store operations that occurs when the pointer being stored has a newer generation than the pointer being stored through. This trap allows the SPUR Lisp runtime system to keep track of all pointers that point from an older space to a newer one.

To avoid confusion, note the difference between the generation number, which is the number stored in the two generation bits of each SPUR pointer, and the space number, which is the number we refer to throughout this discussion of memory management. For hardware reasons, objects in the newest space have a generation number of 3. In this document, generation number 3 corresponds to space 0. Likewise, generation 2 corresponds with space 1, generation 1 corresponds with space 2, and generation 0 corresponds with

[10]Ungar has shown that generation garbage collection can be used very effectively in Smalltalk because most objects are short lived [Ung84]. Several Lisp implementations use generation garbage collection, and report similar success [Moo84,Sin85].

space 3. Space 0 is also referred to as newspace. Space 3, the oldest space, is sometimes referred to as *static space*, because objects allocated in it are never collected. The fact that generation numbers and space numbers are inverted is an unfortunate result of the SPUR hardware design.

5.3 Organization of Spaces in Memory

The organization of the SPUR Lisp heap is the subject of ongoing research and the design is currently very fluid. This discussion describes the initial implementation, which is very simple and serves well as a control from which to measure the performance advantages of more complex implementations. The SPUR heap is divided into four spaces. Initially, objects are allocated in newspace. After surviving a single garbage collection, objects in newspace are copied to space 1. Objects collected in space 1 are copied to space 2. Space 3 is reserved for allocation of objects that are unlikely to become garbage. Space 2 is where all long lived objects are stored. Since remembered lists are only kept for pointers to younger generations, when space 2 needs collection, spaces 1 and 0 must also be copied. Cross space pointer information needs be kept only for pointers into spaces 1 and 0. Thus there two remembered lists: rem 0, which contains the addresses of pointers into space 0 from spaces 1–3, and rem 1, which contains addresses of pointers into space 1 from spaces 2 and 3. The remembered lists are implemented as stacks of consecutive 32-bit addresses. Each 32-bit address is the address of a pointer that points from an older space to the space of the remembered list. These stacks of 32-bit addresses are stored as a separate part of the heap. Figure 20 illustrates the layout of the heap in SPUR Lisp.

Constants 1–18 of the system constants vector are allocation variables. Table 6 shows how these allocation variables correspond to addresses into spaces and remembered lists.

5.4 Memory Consistency Issues

In its initial configuration, SPUR Lisp memory management is very simple. Objects of all types are allocated from the same region of memory and are not segregated by type as in many Lisp systems. The SPUR Lisp heap cannot be scanned linearly, as it can in other Lisp implementations [Moo85,Sha87]. Being able to scan the heap is essential to incremental garbage collection, as proposed by Baker [Bak78]. Thus, the SPUR Lisp memory organization does not permit Baker-style incremental garbage collection. With generation garbage collection, however, SPUR Lisp avoids non-disruptive behavior, which is a primary advantage of incremental garbage collection. [11]

Correct garbage collection requires that all objects in use can be identified at any time. Such identification is impossible at certain critical times because operations such as object allocation temporarily place memory in an inconsistent state. Two kinds of situations

[11] The other major advantage of incremental garbage collection, that of being suited for real-time applications, is only available for Lisp systems with limited array sizes and no virtual memory. No existing Lisp system we are aware of provides real-time response from its incremental garbage collection.

NAME	INDEX	DESCRIPTION
rem_0_base	1	Address of the base of remembered list for space 0. Constant with fixnum tag.
rem_0_top	2	Current top the the remembered list for space 0. Variable with fixnum tag.
rem_0_max	3	Address of the maximum allowed extent of the remembered list for space 0. Coincides with rem_1_base. Constant with fixnum tag.
rem_1_base,top,max	4,5,6	Addresses as per rem_0_base, top, and max.
space_0_base, max	7,9	Addresses as per rem_0_base and max.
space_0_top	8	*NOT* current top of space 0, since this is pointed to by r8, the heap pointer. This slot is unused except to store the value of r8 when a memory image of SPUR Lisp is saved.
space_1_base,top,max	10,11,12	Addresses as per rem_0_base, top, and max.
space_2_base,top,max	13,14,15	Addresses as per rem_0_base, top, and max.
space_3_base,top,max	16,17,18	Addresses as per rem_0_base, top, and max.

Table 6: Allocation Variables in the System Constants Vector

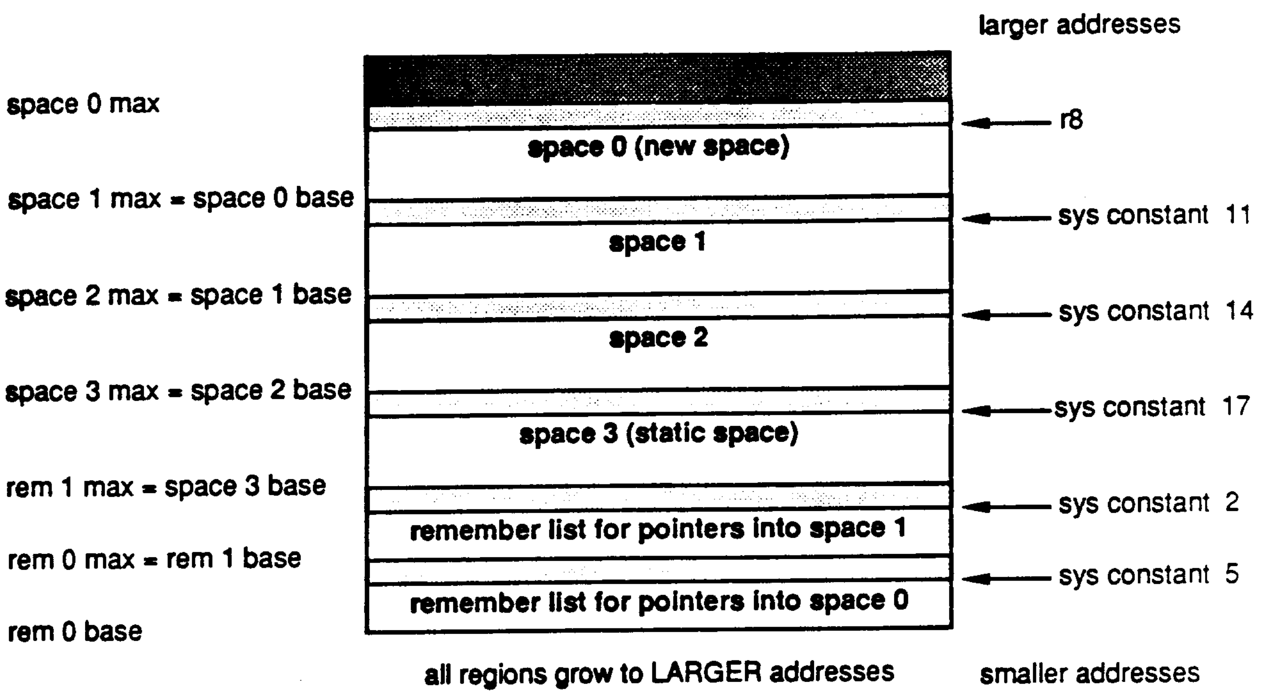


Figure 20: Division of Heap into Spaces and Remembered Lists

cause inconsistent state in SPUR Lisp. Some primitive system routines (section 6) cause temporary inconsistent state. These routines are coded in assembly language for speed. Instruction sequences in these routines are carefully chosen by the implementor. Inconsistent state also occurs during execution of instruction sequences generated by the SPUR Lisp compiler. These sequences are more likely to cause garbage collection errors because they are not chosen by the implementor.

For example, the SPUR architecture is pipelined and contains instructions such as `call`, `return`, and `cmp` that execute an extra instruction before changing the pc. The SPUR Lisp compiler produces code naively, not attempting to use this extra instruction. After the initial code generation another pass could attempt to reorder the instructions to take advantage of these extra slots. Such a code reorganizer will only produce correct code for SPUR Lisp if it understands that the following code sequences are immutable.

5.4.1 Allocation Sequences

The philosophy of memory management in SPUR Lisp matches the simplicity of SPUR architecture design. Certain essential operations in SPUR Lisp are very fast. The cons operation takes 4 SPUR instructions and is open-coded.[12] Assume the car value to store is in rCAR, and the cdr value to store is in rCDR, then a cons operation looks like:

```
store   rCDR, r8(8)      ; r8 points to the heap
store   rCAR, r8(0)
move    rCONS, r8        ; rCONS contains the result
add     r8, r8, 16       ; bump r8
```

This sequence is possible because the page at the top of the heap is a non-writable page. If the first store causes an operating system bad page trap, the garbage collector is invoked, and r8 is reset to the reclaimed portion of the heap. Then the store instruction is retried and works the second time. The order of instructions is important; if the car were stored first, and then the store of the cdr failed, r8 would be relocated, but the car would already have been stored into the incorrect place.

Another immutable instruction sequence is the open-coded operation list, which takes advantage of the non-writable page at the top of the heap to avoid explicitly checking if enough storage is available. The first instruction in a list operation writes to the last cons object needed for the list. By forcing a garbage collection at the outset, the code for list is written assuming that enough memory is available.

5.4.2 Stores into General Vectors

A more complex situation arises during stores into vectors. The store instruction in the SPUR architecture cannot use two registers (base + displacement) to form the store address. Instead, the displacement has to be a 14-bit constant. This means that to store into a vector, a register must contain the address of an item inside the vector. This address is not a valid pointer to any Lisp object since it points *inside* the vector, not to its header. For example, suppose we want to execute the form (setq (svref A X) V). The instruction sequence would be:

```
load    rA, A                    ; rA points to a g-vector
nop
load    rX, X                    ; rX contains a fixnum index
nop
load    rV, V                    ; V is the value to store
nop
lsh     rX, rX, 3                ; 8 bytes/pointer element
add_nt  rINTERNAL, rA, rX        ; create pointer to the element
store   rV, rINTERNAL(16)        ; add in header offset
```

Notice that the internal pointer created does not point to a legal SPUR object. The architecture takes the typegen of the destination of add_nt from the typegen of the first operand.

[12]Because the store instructions take two machine cycles, and each cycle is 150ns., a cons operation takes approximately 1 microsecond.

rINTERNAL, thus, has type **g-vector** and the generation number of the g-vector it points inside. If a garbage collection occurred at the **store** instruction, rINTERNAL would look like a pointer to a g-vector. The garbage collector must distinguish between internal g-vector pointers like rINTERNAL, and regular g-vector pointers like rA.

SPUR Lisp uses a clever encoding of the g-vector header to distinguish between an internal g-vector pointer and a pointer to the g-vector header. Internal g-vector pointers always point to memory that contains a legal SPUR pointer, as shown in figure 2. Note that the first 32-bits of the referenced object contain a valid SPUR heap address. Also, recall the format of the g-vector header in figure 7. The first 32-bit word pointed to by a real g-vector pointer contains #b00 in the upper two bits, by definition. If this first word was interpreted as an address, it would be an address in the kernel segment, which does not contain heap objects.

When the garbage collector encounters a pointer into a g-vector, it scans back through the g-vector to the header. It then relocates the g-vector and installs a forwarding pointer in the old object. The garbage collector also updates the internal pointer to point into the new g-vector object. Because the garbage collector can distinguish internal g-vector pointers from real g-vector pointers, internal g-vector pointers do not introduce any inconsistency. They can exist indefinitely and code sequences that produce them can be arbitrarily rescheduled.

5.4.3 Stores into Integer Vectors

Stores into integer vectors have the same constraints as stores into general vectors. The instruction sequence for `(setf (svref A X) I)`, where `A` is an i-vector with 32-bit elements, would be:

```
load     rA, A                    ; rA points to a i-vector
nop
load     rX, X                    ; rX contains a fixnum index
nop
load     rV, V                    ; V is the value to store
nop
lsh      rX, rX, 2                ; 4 bytes/32-bit element
add_nt   rINTERNAL, rX, rA        ; create pointer to the element
store    rV, rINTERNAL(16)        ; add in header offset
```

Without a **store** instruction that uses two registers to form an effective address, an internal pointer into an i-vector must be created. Internal i-vector pointers are handled differently than internal g-vector pointers. Unlike internal g-vector pointers, which are always distinguishable from pointers to g-vector headers, internal i-vector pointers can point to arbitrary bit patterns and cannot be distinguished from pointers to i-vector headers. Because they cannot be distinguished from other pointers, the creation of internal i-vector pointers causes an inconsistent state.

Note the the add_nt instruction gives the internal i-vector pointer the fixnum type. After the store has executed, the fixnum address is no longer needed. Because the fixnum is an immediate, the garbage collector will ignore it. Before the store is executed, the fixnum address is a dangerous inconsistency. If a garbage collection occurs between the add_nt and the store,[13] the garbage collector must recognize that the internal pointer is more than just a fixnum or it might relocate the i-vector without relocating the internal pointer. What is worse, if a code rescheduler places arbitrary code between the add_nt and the store, then the inconsistent state persists for an indefinite period of time. In section 5.5, we solve the inconsistency problem in the absence of rescheduling. In addition, we require that code reschedulers are aware of the two instruction sequence, add_nt followed by store, and that they never schedule instructions between the two.

5.5 Multiprocessor Garbage Collection

Multiprocessing generation garbage collection for SPUR Lisp has not been implemented. We intend to study and implement such an algorithm in the near future. Here we discuss one problem associated with multiprocessor garbage collection: arbitrary garbage collection interrupts. The current model for multiprocessor generation garbage collection is that when one processor runs out of allocation memory, it interrupts all the processors, and they cooperate to collect objects from newspace. This model implies that a processor can be interrupted for garbage collection at any time. Such a model requires special actions to be taken in certain circumstances to ensure that the objects in memory are in a consistent state when the garbage collection occurs. We will discuss both the circumstances that cause an inconsistent state to arise and ways to avoid inconsistency during garbage collection.

There are many SPUR Lisp operations that put memory in an inconsistent state for a short time during their execution. The instruction sequence for setting an element in a i-vector requires the creation of an inconsistent internal pointer (section 5.4.3). The instruction sequence for list allocates the list elements as a unit, and remains in an inconsistent state until the list is entirely linked together. load and defun require operations on the caller sets of functions, which are dangerous because addresses of functions are manipulated as fixnums. If a garbage collection occurs during a caller set operation, a function object might get relocated, but the fixnum address of the function would not. Finally, all the primitive routines that allocate memory for different types of objects require special handling because they require several instructions to set up the headers of objects like i-vectors and g-vectors.

We have two ways to handle garbage collection interrupts during sequences in which the memory state is inconsistent: *software breakpoints* and *software critical sections*. Software breakpoints are implemented by the trap handler for a garbage collection interrupt. When an interrupt occurs, the trap handler examines the instruction sequence starting at the instruction in which the interrupt occurred, and searches for an instruction of the following type: jump, cmp, call, return, or trap—any instruction that alters the flow of control. It

[13]In multiprocessor SPUR Lisp, a global garbage collection can be triggered by any processor at any time. Possible solutions to this problem are discussed in section 5.5.

replaces the first such instruction it finds with a special trap instruction, called the software breakpoint. It then restarts the interrupted instruction, and instructions are executed until the software breakpoint is reached. With this design, all the SPUR Lisp system must do is insure that the memory state is consistent at all instructions that alter the flow of control. This design prevents the problem of inconsistent state between the `add_nt` and `store` instructions on stores into vectors.

The software breakpoint design has the advantage that it is simple to implement. It greatly reduces the places where inconsistent state must be avoided. Certain care must be taken with this design because a second interrupt might occur during the execution of the instructions leading to the software breakpoint after the first interrupt. Interrupt handlers, like the garbage collection interrupt handler, must be written so that they check if a software breakpoint is already in progress, and "do the right thing" in this case. Unfortunately, software breakpoints alone are insufficient to avoid inconsistency during garbage collection interrupts. The code for the open-coded `list` operation loops while the memory state is inconsistent. A mechanism is required that allows inconsistent state across branches and jumps. *Software critical sections* provide such a mechanism.

Software critical sections act much as you would think. If a particular instruction sequence is sensitive, it is placed inside a software critical section, and if a garbage collection interrupt occurs, the critical section is guaranteed to complete before the interrupt is serviced. To mark the entry of a software critical section, we use the low order bit in register 10, the return address. Since the return address must be word aligned, the two lowest bits of the return address are ignored by the hardware. To indicate that a software critical section has been entered, the low order bit of r10 is set. To indicate the end of the software critical section, the low order bit of r10 is cleared. The garbage collection interrupt handler is responsible for identifying software critical sections. If the low order bit of r10 is set when a garbage collection interrupt occurs, the handler scans linearly through the code until it finds the instruction that clears the bit in r10. This instruction must follow the instruction that sets the bit and be in the same function. The handler then replaces the instruction that clears the bit with a software interrupt trap and restarts the instruction as with software interrupts. Code within a software critical section must be very careful not to call other functions or cause traps. With both software breakpoints and software critical sections, any action that alters the flow of control in such a way that the breakpoint is never reached must be avoided. If all possible changes in the flow of control are anticipated, we feel confident that all the code for SPUR Lisp can be written in such a way that a garbage collection interrupt can happen at any time and be serviced correctly.

6 Runtime Operation of SPUR Lisp

This section provides a brief overview of the organization of SPUR Lisp during execution. The primitive non-Lisp coded routines are described and the possible traps that can occur at runtime are listed. As described in appendix D, the SPUR Lisp runtime primitives are currently implemented in both SPUR Lisp assembler (SPLASM) and in C as part of the SPUR simulator. Ultimately, all kernel functions will be written in SPLASM. This listing indicates which primitives are implemented in the SPUR Lisp assembler (asm primitive) and which are implemented in C (C primitive).

6.1 Runtime Primitives

The SPUR Lisp compiler is implemented as a translator from Spice Lisp bytecodes (provided by the Spice Lisp compiler) to a combination of open-coded SPUR instruction sequences and calls to a set of primitive routines. These primitives are coded mostly in SPLASM, but are accessible as SPUR Lisp functions and interned in the `system` package. The primitive routines fall into several categories: memory allocation, vector operations, arithmetic operations, logic operations, string operations, and miscellaneous operations.

6.1.1 Allocation Primitives

`rt-alloc-i-vector` *length access-type* [asm primitive]

`rt-alloc-i-vector` allocates an i-vector of *length* elements, each element of the size specified by *access-type*. This primitive is used to allocate bit vectors and parts of a function's caller set, as well as programmer declared vectors of integers. *length* must be a non-negative fixnum. *access-type* must be a fixnum from 0–5. No initialization is done.

`rt-alloc-string` *length* [asm primitive]

`rt-alloc-string` allocates a string *length* bytes long, and NULL terminates the result before returning it. *length* must be a non-negative fixnum. No initialization is done.

`rt-alloc-bignum` *length* [asm primitive]

`rt-alloc-bignum` allocates a bignum of *length* bytes, and initializes the bytes to 0. *length* must be a non-negative fixnum.

`rt-make-complex` *real-part complex-part* [asm primitive] `rt-make-complex` allocates a complex number and with real and complex parts specified by the arguments. *real-part* and *complex-part* must have non-complex numeric types.

`rt-make-ratio` *numerator denominator* [asm primitive]

`rt-make-ratio` allocates a ratio with the numerator and denominator specified by the arguments. *numerator* and *denominator* must have bignum or fixnum types.

`rt-alloc-g-vector` *length initial-value* [asm primitive]

`rt-alloc-g-vector` allocates a general vector with *length* elements and initializes the elements to the value specified by *initial-value*. *length* must be a non-negative fixnum.

`rt-vector` *arg1 arg2 ... argn* [asm primitive]

`rt-vector` takes any number of arguments up to `call-arguments-limit` as specified by Common Lisp.[14] `rt-vector` allocates a vector as large as the argument list, and copies the arguments into the vector before returning it.

`rt-alloc-function` *code-length constants-length* [asm primitive]

`rt-alloc-function` allocates a compiled function object. *code-length* is a non-negative fixnum that specifies the number of instructions the function will contain. *constants-length* is a non-negative fixnum that specifies how many constants to allocate in the constants vector for the function. `rt-alloc-function` allocates the code vector as an i-vector in static space, and allocates the constants vector by calling `rt-alloc-g-vector`. Elements of the code vector are not initialized. Elements of the constants vector are initialized to `nil`.

`rt-alloc-array` *number-of-dimensions* [asm primitive]

`rt-alloc-array` allocates an array object with the specified number of dimensions. *number-of-dimensions* is a non-negative fixnum less than `array-dimensions-limit`, as specified by Common Lisp.[15] Elements of the array object are initialized to `nil`.

`rt-alloc-symbol` *print-name* [asm primitive]

[14]`call-arguments-limit` in SPUR Lisp is 255.
[15]In SPUR Lisp, `array-dimensions-limit` has the value 2147483647.

`rt-alloc-symbol` allocates a SPUR Lisp symbol with the specified print name. *print-name* should have type **string**. The value and function definition component of the symbol are initialized to **undefined**, and the package and property list components of the symbol are initialized to `nil`. `rt-alloc-symbol` does *not* intern the symbol it allocates.

`rt-list` *arg1 arg2 ... argn* [asm primitive]

Like `rt-vector`, `rt-list` takes an arbitrary number of arguments. The arguments to `rt-list` are made into a list that is returned. Although many `list` operations are open-coded, this function is called frequently to create `&rest` arguments.

`rt-list-star` *arg1 arg2 ... argn* [asm primitive]

`rt-list-star` performs the Common Lisp `list*` operation on its arguments. This function is sometimes open-coded and is rarely used.

6.1.2 Vector and Array Primitives

Many of the vector and array bytecodes in Spice Lisp take a variety of possible argument types. These bytecodes have been translated into primitive routines to avoid the code expansion necessary to open-code the required type dispatch.

`rt-shrink-vector` *vector new-length* [C primitive]

`rt-shrink-vector` takes a vector argument of type **bignum**, **i-vector**, **string**, or **g-vector**. `rt-shrink-vector` makes the size of the array *new-length*, which must be a non-negative fixnum smaller than the original length of the *vector* argument. `rt-shrink-vector` shrinks vectors in place. Shrinking a string causes a new NULL terminator to be placed at the end of the string. `rt-shrink-vector` returns the modified vector.

`rt-typed-vref` *access-type vector index* [C primitive]

`rt-typed-vref` references an element of an i-vector. The *vector* argument must have an i-vector base type: **i-vector**, **string**, **bignum**, or **function**. *access-type* specifies the access-type assumed in the vector reference, and may not correspond to the access-type of the argument *vector*. *access-type* should be a fixnum between 0 and 5. *index* should be a non-negative fixnum that does not cause a reference outside the bounds of the vector. The value returned has type **fixnum**.

`rt-typed-vset` *access-type vector index value* [C primitive] `rt-typed-vset` sets an element of an i-vector. *vector* should have the type: i-vector, bignum, string, or function. *access-type* specifies the access-type assumed in the vector assignment, and may not correspond to the access-type of the vector argument. *access-type* should be a fixnum between 0 and 5. *index* should be a non-negative fixnum that does not cause the assignment to fall outside the bounds of the vector. *value* should be a fixnum of a size appropriate for *access-type*. The *value* argument is returned by this primitive.

`rt-aref1` *vector index* [C primitive]

`rt-aref1` references an element in a vector. *vector* can be any vector type: array, i-vector, function, bignum, g-vector, or string. *index* should be a non-negative fixnum that falls within the bounds of the vector. References to i-vectors, functions, and bignums return fixnums. References to strings return characters; the ASCII code is inserted into a character object with NULL (0) *font* and *bits* fields.

`rt-aset1` *vector index value* [C primitive]

`rt-aset1` assigns an element in a vector. *vector* can be any vector type: array, i-vector, function, bignum, g-vector, or string. *index* should be a non-negative fixnum that falls within the bounds of the vector. For bignums, functions, and i-vectors, *value* should have type fixnum. For strings, value should have type character; the ASCII code is extracted from the character and stored into the string. `rt-aset1` returns *value*.

6.1.3 Arithmetic Primitives

SPUR Lisp provides arithmetic primitives that operate on any numeric type: fixnum, bignum, short-float, long-float, extended-float, complex, and ratio. These routines perform a dispatch on the types of the operands, and return a numeric value of the appropriate type. Not all arithmetic routines have been implemented with all combinations of numeric types, especially the ratio and complex types.

`rt-integer-length` *integer* [C primitive]

`rt-integer-length` returns a fixnum that represents the number of bits in an integer as specified by the `integer-length` function in Common Lisp. *integer* should have type fixnum or bignum.

`rt-short-float` *number* [C primitive]

`rt-short-float` returns a value of type short-float. The parameter *number* is coerced to type short-float as specified by the Common Lisp function `float`. *number* can have any non-complex numeric type.

`rt-long-float` *number* [C primitive]

`rt-long-float` returns a value of type *long-float*. The parameter *number* is coerced to type long-float as specified by the Common Lisp function `float`. *number* can have any non-complex numeric type.

`rt-scale-float` *float integer* [C primitive]

`rt-scale-float` performs the Common Lisp function `scale-float`. *float* must have a floating point type. *integer* must have type fixnum. A floating point number with the same type as *float* is returned.

`rt-decode-float` *float* [C primitive]

`rt-decode-float` performs the Common Lisp function `decode-float`. *float* must have a floating point type. A three value multiple value is returned. The first and third values have type long-float, and the second value has type fixnum.

`rt-truncate` *number divisor* [C primitive]

`rt-truncate` divides the first operand by the second, and returns two values. There are several cases depending on the types of the operands. If the *divisor* has type fixnum, `rt-truncate` returns a fixnum or bignum representing the integer part of *number* divided by *divisor* rounded toward zero and a float or ratio representing the fractional part of *number* divided by *divisor*. If either *number* or *divisor* are floats or ratios, then `rt-truncate` returns a fixnum or bignum quotient and a float or ratio remainder, as determined by the type coercion rules for `+`.

`rt-multiply` x y [C primitive]

`rt-multiply` multiplies its arguments and returns a result with a type as specified in Common Lisp. x and y can have any numeric type.

`rt-divide` x y [C primitive]

`rt-divide` divides its arguments and returns a result with a type as specified in Common Lisp. x and y can have any numeric type.

6.1.4 Logical Primitives

`rt-boole` *operation operand1 operand2* [C primitive]

rt-boole performs a boolean operation on its arguments. The operation is specified by *operation*, which is a fixnum from 0–15. Table 7 summarizes the possible operations and their numeric codes (operation names correspond to those in Common Lisp). *operand1* and

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

Table 7: Numeric Codes for `rt-boole` Primitive

operand2 should both have the same type: either fixnum or bignum. An integer with the same type as *operand1* is returned. `rt-boole` does not currently work for bignum operands.

`rt-ldb` *size position number* [C primitive]

`rt-ldb` performs the Common Lisp function `ldb`. The *size* and *position* parameters should be non-negative fixnums. The *number* should be an integer. A non-negative integer is returned.

`rt-mask-field` *size position number* [C primitive]

`rt-mask-field` performs the Common Lisp `mask-field` function. *size* and *position* should be non-negative fixnums. *number* should be an integer. An integer with the same type as *number* is returned.

`rt-dpb` *bits size position number* [C primitive]

`rt-dpb` performs the Common Lisp `dpb` function. *size* and *position* should be non-negative fixnums. *number* and *bits* should be integers. An integer with the same type as *number* is returned.

`rt-deposit-field` *bits size position number* [C primitive]

`rt-deposit-field` performs the Common Lisp `deposit-field` function. *size* and *position* should be non-negative fixnums. *number* and *bits* should be integers. An integer with the same type as *number* is returned.

`rt-logldb` *size position number* [C primitive]

`rt-logldb` performs the same operation as `rt-ldb`, but all of its operands must be fixnums. This operation is performed by a runtime routine because the SPUR hardware provides minimal support for logical operations. A fixnum is returned.

`rt-logdpb` *bits size position number* [C primitive]

`rt-logdpb` performs the same operation as `rt-dpb`, but all of its operands must be fixnums. This operation is performed by a runtime routine because the SPUR hardware has minimal support for logical operations. A fixnum is returned.

`rt-ash` *number distance* [asm primitive]

`rt-ash` performs the Common Lisp function `ash`. *number* should be an integer. *distance* should be a fixnum. The value returned is the integer obtained by doing an arithmetic shift of *number* to the right or left depending on the sign of *distance*.

`rt-lsh` *number distance* [asm primitive]

`rt-lsh` performs a logical shift in either direction. *number* should be an integer. *distance* should be a fixnum. If *distance* is non-negative, shifting is done to the left. If *distance* is negative, shifting is done to the right. The value returned is the integer obtained by doing a logical shift of *number* to the right or left depending on the sign of *distance*.

6.1.5 String Primitives

`rt-byte-blt` *src src-start dest dest-start dest-end* [asm primitive]

`rt-byte-blt` copies blocks of characters from one string to another. *src* and *dest* should be strings or i-vectors of bytes. *src-start*, *dest-start*, and *dest-end* should be fixnums. *dest-end* - *dest-start* bytes are copied from *src* starting at *src-index* to *dest* starting at *dest-index* up to but not including index *dest-end*. *src* and *dest* may be the same string, and the indices may overlap, in which case the copy occurs as is it were a byte at a time starting from the lowest index. *dest* is returned.

`rt-find-character` *string start end char* [C primitive]

`rt-find-character` returns the index of the first occurrence of *char* in *string* between indices *start* and *end*. *string* should be a string or an i-vector of bytes. *start* and *end* should be fixnums. *char* has type character or fixnum. A fixnum is returned if the character is found. `nil` is returned otherwise.

`rt-find-character-with-attribute` *string start end table mask* [C primitive]

`rt-find-character-with-attribute` finds an element in a string as a function of entries in a table. The 8-bit codes of *string* are scanned between indices `start` and `end` as in `rt-find-character`. *table* and *string* are strings or i-vectors of bytes. *table* should have 256 elements. The codes of elements in *string* are used as indices into *table*. When the referenced element of *table* has a non-zero value after being bitwise anded with *mask*, the current index into *string* is returned. *mask* should have a fixnum value. If no reference succeeds, `nil` is returned.

`rt-sxhash-simple-string` *string length* [C primitive]

`rt-sxhash-simple-string` computes a hash value from the first *length* characters of the *string* argument. *string* should be a string or i-vector of bytes. *length* should be `nil` or a fixnum. If *length* is `nil`, the length of *string* is determined from its header. A non-negative fixnum is returned.

6.1.6 Miscellaneous Primitives

`rt-putf` *list indicator value* [asm primitive]

`rt-putf` manipulates lists with structures like a property list. *list* should have type cons or nil. If *list* is `nil`, a new list is created with *indicator* and *value* as elements. Otherwise, *list* is cddred down, and *indicator* is searched for. If found, the car of the cons after indicator is set to *value*, otherwise, *indicator* and *value* are both consed onto the list, which is returned. If *list* were the property list of symbol `X`, this operation would correspond to the Common Lisp operation `(setf (get 'X` *indicator* `)` *value* `)`.

`rt-throw` *throw-values* ... *throw-tag* [C primitive] `rt-throw` implements the Common Lisp *throw* operation. The arguments to `rt-throw` are the values being thrown, except the last, which is that tag being thrown to. The behavior of `catch` and `throw` are described in section 4.7.

`rt-mv-spill` *value1 value2 ... valuen* [asm primitive]

`rt-mv-spill` takes the arguments passed to it and stores away the group as a saved multiple value on the miscellaneous stack. This function returns the miscellaneous stack pointer with a fixnum tag that identifies the location of the spilled multiple value.

`rt-mv-unspill` *mv-address-1 mv-address-2 ... mv-address-n* [asm primitive]

`rt-mv-unspill` takes an arbitrary number of fixnum addresses of multiple values on the stack (as returned by `rt-mv-spill`) and creates a new multiple value created from the elements of the spilled multiple values. `rt-mv-unspill` returns the multiple value it creates.

`rt-spread` *list start-index* [asm primitive]

`rt-spread` takes a list and creates a multiple value. *list* is a list of values. *start-index* is a non-negative fixnum that specifies which argument index to place the first component of the multiple value. Arguments with indices lower than *start-index* are not modified. The number of return values is set to the actual number of values being returned. `rt-spread` returns a multiple value.

`rt-length` *sequence* [asm primitive]

`rt-length` returns the length of the sequence argument. sequence can have types: nil, i-vector, string, bignum, g-vector, cons, or array. `rt-length` returns a fixnum that is the number of elements in the *sequence* argument.

`spur-get-key` *starting-arg-index keyword* [asm primitive]

`spur-get-key` gets the value of the keyword argument specified as its parameter from the argument list of the function that calls it. `spur-get-key` operates in the window frame of the function calling it and searches the argument list for a keyword/value pair starting with the argument specified by *starting-arg-index*. `spur-get-key` does not perform a return in the normal sense, but instead returns by setting the values of two registers in the output registers of the function that called it. r28 is set to the value of the keyword if the keyword/value pair is found and `nil` otherwise. r29 is set to a non-`nil` value if the keyword is found, and `nil` otherwise. The number of return values is not set upon return.

`spur-keyword-test` *starting-arg-index keyword-list* [asm primitive]

`spur-keyword-test` checks that all keywords in the argument list of a function are valid keywords. Like `spur-get-key`, `spur-keyword-test` operates in the window frame of the function that calls it. *keyword-list* identifies all valid keywords to the function. Arguments to the calling function starting at *starting-arg-index* are checked against the elements of *keyword-list*. `spur-keyword-test` does not return in the usual way. If all keywords are valid, spur-keyword-test returns control to the caller without returning any values. If a keyword is invalid, spur-keyword-test calls the function `error`.

`rt-invoke-interpreter` [asm primitive]

`rt-invoke-interpreter` is called by the compiled function body for interpreted functions discussed in section 4.6. Like `spur-get-key`, `rt-invoke-interpreted` operates in the window frame of the function calling it. r18 in the caller's window frame contains the `lambda` list to interpret. r18 is made the first argument to the calling function, and all other arguments are shift ahead one position. Control is then transferred to `%sp-internal-apply`, which implements the SPUR Lisp interpreter in its full generality. Return is not performed in the normal way. Control of the calling function is simply transferred to `%sp-internal-apply`.

6.2 Error Handling

The SPUR architecture provides a `cmp_trap` instruction for quickly testing for illegal conditions, such as an incorrect number of arguments to a function. A 9-bit immediate field in the `cmp_trap` instruction is used in SPUR Lisp to allow the trap handler to quickly identify the cause of the trap and report this to the user. Tables 8, 9, and 10 enumerate the SPUR Lisp software traps, the trap numbers used to identify the traps, and briefly describes each trap. Note that the symbol <*> in the description refers to a function with the same name as the name of the trap. For example, for the bit-vector-index-trap, <*> refers to the function `bit-vector-index`. Software traps detected by SPUR Lisp will raise an error that will report the kind of trap and location of the trap to the user.

NAME	INDEX	DESCRIPTION
actuals-neq-trap	0	Actuals not equal to number of formals.
actuals-lt-trap	1	Too few actual arguments.
actuals-gt-trap	2	Too many actual arguments.
bind-symbol-trap	4	`bind` applied to non-symbol.
replace-car-cons-trap	5	`replace-car` applied to non-cons.
replace-cdr-cons-trap	6	`replace-cdr` applied to non-cons.
get-value-trap	7	`get-value` applied to non-symbol.
undef-value-trap	8	`get-value` applied to undefined symbol.
set-value-trap	9	`set-value` applied to non-symbol.
get-definition-trap	10	`get-definition` applied to non-symbol.
undef-function-trap	11	`get-definition` applied to undefined symbol.
set-definition-trap	12	`set-definition` applied to non-symbol.
get-plist-trap	13	`get-plist` applied to non-symbol.
set-plist-trap	14	`set-plist` applied to non-symbol.
get-pname-trap	15	`get-pname` applied to non-symbol.
get-package-trap	16	`get-package` applied to non-symbol.
set-package-trap	17	`set-package` applied to non-symbol.
boundp-trap	18	boundp applied to non-symbol.
fboundp-trap	19	`fboundp` applied to non-symbol.
array-index-trap	20	`array-index` applied to a non-array.
array-index-number-trap	21	Too many indices for an array.
array-index-range-trap	22	Array index out of bounds.
char-index-trap	23	`char-index` applied to a non-string.
char-index-range-trap	24	`char-index` out of bounds.
bit-vector-index-trap	25	`<*>` applied to a non-bit-vector.
bit-vector-range-trap	26	`bit-vector-index` out of bounds.
g-vector-ref-trap	27	`g-vector-ref` applied to a non-g-vector.
g-vector-ref-index-trap	28	g-vector index out of bounds on ref.
g-vector-set-trap	29	`<*>` applied to a non-g-vector.

Table 8: SPUR Lisp Traps

7 The SPUR Lisp Implementation

This section presents an overview of SPUR Lisp as it is currently implemented. We outline the important components of the implementation, each described more fully in an appendix. We also discuss the current and future status of the implementation.

7.1 Components of SPUR Lisp

The major components of the SPUR Lisp implementation are: BARB, the SPUR simulator; SLC, the SPUR Lisp compiler; RT, the SPUR Lisp runtime kernel; CODE, our Common Lisp implementation; TH, the SPUR trap handlers; SPLASM, the SPUR Lisp assembler; and ZEUS, the SPUR Lisp creator. The implementation for each component is located in a particular directory, and we shall refer to the directory in which the component resides as one would C-shell variable with the same name. Thus, `$barb` will refer to the directory that the SPUR simulator is implemented in. A C-shell file in the `$barb` directory named `SPUR-NAMES` can be sourced and will create shell variables for all the SPUR Lisp components. Below, we briefly describe each component. Times reported are on a VAX 8800 computer with 32 megabytes of memory.

7.1.1 BARB: the SPUR Simulator

SPUR Lisp is implemented on a simulator for SPUR hardware because the actual hardware was unavailable. BARB, the SPUR simulator, is written in C and runs on the VAX 8800 and Sun 3 computers. BARB is an instruction level simulator, and implements the SPUR instruction set as defined by [Tay85]. BARB executes SPUR instructions at approximately 60,000 SPUR instructions per CPU second. Simulations of more than 100 million SPUR instructions are routine. BARB occupies approximately 40 files containing 15,000 lines of C.

7.1.2 SLC: the SPUR Lisp Compiler

The SPUR Lisp compiler, SLC, compiles SPUR Lisp source code into Lisp object code. The format of the object files, commonly called FASL files, is the same format used by Spice Lisp and is documented in [WFG85]. The SPUR Lisp compiler is a multi-pass compiler that uses the Spice Lisp compiler to translate SPUR Lisp into Spice bytecodes and then translates the bytecodes into SPUR instructions. The SPUR Lisp compiler is written in Common Lisp, and runs in Franz Lisp Opus 43.1 with compatibility macros, VAX Lisp and SPUR Lisp. The SPUR Lisp compiler running in Franz Lisp Opus 43.1 compiles SPUR Lisp files at approximately 30 lines per CPU second when writing large listing files. It is much faster when it does not need to write these files. The SPUR Lisp compiler occupies approximately 20 files containing 23,000 lines of code.

NAME	INDEX	DESCRIPTION
g-vector-set-index-trap	30	g-vector index out of bounds on set.
vector-length-trap	31	vector-length applied to non-vector.
g-vector-length-trap	32	<*> applied to non-vector.
string-length-trap	33	string-length applied to non-string.
bit-vector-length-trap	34	<*> applied to non-bit-vector.
get-vector-subtype-trap	35	<*> applied to non-vector.
set-vector-subtype-trap	36	<*> applied to non-vector.
set-vector-subtype-type-trap	37	Non-fixnum to set-vector-subtype.
get-vector-access-code-trap	38	<*> applied to non-vector.
header-length-trap	39	header-length applied to non-array.
header-ref-trap	40	header-ref applied to non-array.
header-ref-index-trap	41	header-ref index out of range.
header-set-trap	42	header-set applied to non-array.
header-set-index-trap	43	header-set index out of range.
call-non-symbol-trap	44	Call to non-symbol.
assoc-non-list-trap	45	assoc on non-list.
assq-non-list-trap	46	assq on non-list.
last-non-list-trap	47	last on non-list.
nthcdr-non-fixnum-trap	48	nthcdr with non-fixnum index.
nthcdr-non-list-trap	49	nthcdr on non-list.
nthcdr-negative-trap	50	nthcdr with negative index.
member-non-list-trap	51	member on non-list.
memq-non-list-trap	52	memq on non-list.
getf-non-list-trap	53	getf on non-list.
iheader-ref-trap	55	iheader-ref applied to non-array.
iheader-ref-index-trap	56	iheader-ref index out of range.
iheader-set-trap	57	iheader-set applied to non-array.
iheader-set-index-trap	58	iheader-set index out of range.
8bit-system-ref-trap	59	<*> applied to non-vector or integer.
8bit-system-set-trap	60	<*> applied to non-vector or integer.

Table 9: SPUR Lisp Traps (continued)

NAME	INDEX	DESCRIPTION
16bit-system-ref-trap	61	<*> applied to non-vector or integer.
16bit-system-set-trap	62	<*> applied to non-vector or integer.
signed-32bit-system-ref-trap	63	<*> applied to non-vector or integer.
signed-32bit-system-set-trap	64	<*> applied to non-vector or integer.
non-string-argument-trap	65	Non-string to sxhash.
non-fixnum-argument-trap	66	Non-fixnum to kernel routine.
incorrect-access-type-trap	67	rt-alloc-i-vector passed bad access-type.
complex-argument-trap	68	complex arg to rt-make-complex.
non-integer-argument-trap	69	Non-integer to rt-make-ratio.
gc-space-0-trap	70	Garbage collection of space 0 required.
non-list-arg-trap	71	Non-list to rt-putf .
non-array-aref-trap	72	Non-array to rt-aref1.
non-sequence-arg-trap	73	Non-sequence to rt-length.
unimplemented-software-trap	74	Unimplemented primitive operation.
call-kernel	75	Call kernel instruction replacement.
logldb-trap	76	logldb applied to non-fixnum.
logdpb-target-trap	77	logldb destination is non-fixnum.
logdpb-source-trap	78	logldb value is non-fixnum.
ivref-trap	79	ivref applied to non array.
ivref-index-trap	80	ivref-index out of range.
ivset-trap	81	ivset applied to non array.
ivset-index-trap	82	ivset-index out of range.
space-2-too-small-trap	83	Object larger than space 2.
gc-space-1-trap	84	GC of space 1 requested.
full-gc-trap	85	Full GC requested.
make-unbound-trap	86	make-unbound applied to non-symbol.
db-panic-trap	87	Internal inconsistencies in deep binding routines.

Table 10: SPUR Lisp Traps (continued)

7.1.3 RT: the SPUR Lisp Runtime Kernel

The SPUR Lisp runtime kernel is a set of primitive Lisp functions. Originally the runtime kernel was implemented entirely in C and called from Lisp code using special hooks in the simulated `call` instruction. Many of the runtime functions are now implemented in SPUR assembly code assembled into SPUR FASL files. SPLASM, a component of the SPUR Lisp compiler, assembles these files. Ultimately, all kernel functions will be written in SPUR Lisp assembler. Currently, approximately 20 of the 50 kernel functions are coded in 2000 lines of assembler. The remaining 30 routines are coded in approximately 5000 lines of C.

7.1.4 CODE: the SPUR Lisp Common Lisp Implementation

The CODE component of SPUR Lisp provides the many functions available in Common Lisp. The CODE component is copied almost entirely from Spice Lisp and is implemented entirely in Common Lisp. Machine and operating system dependent functions have been modified to be specific to SPUR Lisp. The CODE component occupies 56 files containing 32,000 lines of Lisp. To compile the entire SPUR Lisp `$code` directory using the SPUR Lisp compiler running in Franz Lisp takes approximately 54 CPU minutes.

7.1.5 TH: SPUR Trap Handlers

Trap handlers for SPUR Lisp are written in assembler and loaded into the lowest part of the SPUR memory. Trap handler files have `a.out` file format and are written in SAS, the standard SPUR assembly language. The syntax of SAS is described in another document [Hil87]. Trap handlers handle such traps as window stack over and underflow, tag traps on arithmetic operations, and generation traps on stores. There are approximately 50 traps handled in 2000 lines of SPUR assembler.

7.1.6 SPLASM: the SPUR Lisp Assembler

SPLASM is an assembler with Lisp-like syntax that is used to implement the SPUR Lisp runtime kernel. SPLASM is a separate from the SPUR assembler, SAS, used to assemble the output of the SPUR C compiler, and used to implement the SPUR trap handlers. SPLASM generates SPUR Lisp FASL files that can be loaded with the Lisp loader. We chose to provide an alternative Lisp assembler for several reasons. First, we wanted to access the Lisp runtime kernel from Lisp, and if we had implemented the runtime kernel in SAS, we would have needed to provide an interface between SPUR Lisp and general `a.out` files. This interface is necessary anyway, but we chose to postpone implementing it. Another reason for providing a separate assembler is that much of the functionality of the assembler is already present in SLC, the SPUR Lisp compiler. Using SLC, SPLASM was easy to create. A final reason for implementing SPLASM is that since it is implemented in Lisp, SPLASM can use Lisp macros and variables, and so it is much easier to use than SAS. SPLASM is written in about 1500 lines of Lisp.

7.1.7 ZEUS: the SPUR Lisp Creator

ZEUS is a program that takes the SPUR Lisp system defined by the FASL files in the `$code` directory, and creates an initial SPUR Lisp memory image. ZEUS simulates the loading of the FASL files into the image and defers load-time execution of forms until all the system functions are present. After the loading is complete, an initialization function is executed and the initialized SPUR Lisp memory image is saved. ZEUS is adapted from the `genesis` program that is used to create an initial Spice Lisp image. ZEUS is written in Common Lisp and implemented using Franz Extended Common Lisp for the VAX. The creation of a SPUR Lisp image takes 11 CPU minutes to load the system FASL files and 3 CPU minutes to execute the initialization routines. The image of SPUR Lisp occupies approximately 2.4 megabytes of disk space. If the SPUR Lisp compiler is also included in the memory image, the image occupies 4.7 megabytes of disk space.

7.2 Implementation Status

SPUR Lisp currently runs on the BARB simulator, which implements a superset of the actual SPUR instruction set. Small changes are required to make the SPUR Lisp implementation compatible with actual SPUR hardware. These changes are:

- The `call_reg` must be replaced uniformly with `jump_reg`, using the replacement suggested in section 4.2.
- The `enable_traps` instruction, used in the trap handlers must be replaced by a sequence of instructions to modify the KPSW.
- The various types of `cmp_branch` instructions used in trap handlers must all be converted to `cmp_branch_delayed` instructions.
- All uses of the `eql` compare condition must be replaced with a sequence of tests.
- All uses of the `endp` compare condition must be eliminated.
- Uses of the `tag_cmp_xxx` instructions must be modified to use the `cmp_xxx` instructions with `tag_eq` and `tag_ne` conditions.
- Generation and `cmp_trap` traps, which have their own trap numbers in the simulator, actually have the same trap numbers as fixnum traps and illegal opcode traps in the hardware. The trap handlers must be rewritten to merge the trap numbers.

Several large test applications have been implemented in SPUR Lisp. These applications include an OPS5 interpreter (3600 lines of Lisp); WEAVER, an OPS5 program that performs circuit routing (about 1000 OPS5 rules); RSIM, a Common Lisp program that simulates simple circuits (2000 lines of Lisp); and the SPUR Lisp compiler (13000 lines of Lisp). Additional applications will be moved to SPUR Lisp as the need arises.

There are also some known errors and weaknesses in the existing SPUR Lisp implementation. They are:

- Bignum arithmetic operations are currently inefficient because they are performed 8-bits at a time, instead of 32-bits at a time, as the SPUR ALU will allow.
- Floating point numbers cannot be correctly printed.
- Runtime routines that support generic arithmetic have not been implemented for all combinations of numeric type such as ratios and complex numbers.
- Runtime routines performing boolean operations are not fully implemented for bignum parameters (e.g., `rt-boole`).
- Calls to the function `%make-immediate-type` create type tags without the correct generation number. This has not caused any known garbage collection errors, but it has the potential to cause errors.
- `debug`, `trace`, and other related Common Lisp functions have not been implemented for SPUR Lisp. The versions available with the system are Spice Lisp specific.
- There are still a few unresolved references in the initial SPUR Lisp image. These references should be resolved or eliminated.
- The operating system interface in SPUR Lisp is implemented through the BARB simulator, and calls are made to Unix library routines. SPUR will run the Sprite operating system and the OS interface in SPUR Lisp should reflect the Sprite OS system calls.
- There is no support for foreign function calls in SPUR Lisp. Support must be provided before calls to the native OS can be made.
- Obvious performance bottlenecks are still being discovered and reimplemented.

7.3 The Future of SPUR Lisp

SPUR Lisp will evolve in two directions. The first direction is towards completeness. In time, SPUR hardware will be available and action will be taken to make SPUR Lisp work directly on the hardware. Missing pieces of the implementation will be provided and new applications will be ported to SPUR Lisp. The second direction is towards multiprocessing. Language ideas for multiprocessing in SPUR Lisp have already been discussed, and are presented in [ZHH*86]. Multiprocessor implementation issues are only beginning to be addressed, including compiler detection of parallelism, creation and scheduling of processes, efficient variable binding strategies, and effective multiprocessor storage reclamation. Work in all of these areas will result in additional technical publications.

Other areas of SPUR Lisp development are more speculative. Because other multiprocessors are being commercially developed, porting SPUR Lisp to hardware other than SPUR is possible. SPUR Lisp will be a testbed for new multiprocessing ideas. If SPUR Lisp is available on a variety of multiprocessor architectures, we can try to develop language features and programming styles for parallelism that are effective on a number of different architectures. Currently there are no plans to port SPUR Lisp, but if after SPUR Lisp has been installed on actual SPUR hardware the SPUR Lisp research effort continues, porting it to other architectures may be considered.

Another possible area of SPUR Lisp development is toward a second generation of SPUR hardware. Taylor has carefully studied performance of SPUR Lisp on the SPUR hardware, and the knowledge he has gained would enable architects to build a much improved SPUR workstation. Such a workstation could have custom 40-bit memory, fast trap handling, and some of the features for Lisp that were eliminated from the original SPUR design. Making SPUR Lisp run on new SPUR hardware would probably be quite easy but would require a few changes in the implementation. Having custom 40-bit memory words would mean that all data in memory would have tags, and the Lisp system would have to be redesigned to take full advantage of the tags.

A Summary of SPUR Lisp Data Types

fixnum An immediate 32-bit two's-complement integer.

character An immediate ASCII character with 8 bits of font information and 8 bits of additional code information.

nil A unique object that has its own type tag. Represented as a symbol, with value appropriate as a cons object as well (40 bytes).

cons A two pointer object with car and cdr components (16 bytes).

symbol A five pointer object, with value, function definition, print name, package and property list components (40 bytes).

i-vector A vector of k-bit values, where k ranges from 1 to 32 by powers of two. Fields in the header include the access-type (determines size of elements), subtype, word size of object, and number of elements in the vector. Used to represent bit vectors, strings, bignums, and vectors of instructions in functions. Variable length (8 bytes of header plus 4 bytes per element aligned to a doubleword boundary).

string Implemented as an i-vector of ASCII 8-bit characters with its own tag. NULL (0) terminated for UNIX and Sprite compatibility.

g-vector A vector of pointers. Used to implemented SPUR Lisp data aggregates including vectors, arrays, and structures. Fields in the header include the header mark, subtype, word size of object, and number of elements in the vector. Variable length (8 bytes of header plus 8 bytes per element).

function An object with two parts. The pointer to a function points to a code vector, which is a modified i-vector that contains instructions for the function and a pointer to the constants vector, which is a g-vector that contains the constants used in the function. Variable length (code vector—16 bytes plus 4 bytes per instruction; constants vector—40 bytes plus 8 bytes per constant).

array The array descriptor for a non-simple SPUR Lisp vector or array. Includes fields for displaced arrays, arrays with fill pointers, and multi-dimensional arrays. Implemented as a g-vector. Size depends on dimensionality of the array.

cclosure A pair of pointers used to implement compiled closures (16 bytes). The first pointer points to the environment for the compiled closure, which is implemented as a general vector. The second pointer points to the function object for the compiled closure.

short-float IEEE 32-bit non-immediate floating point type that is stored in 64 bits to keep objects aligned in memory (8 bytes). Soon to be obsolete (section 2.9).

long-float IEEE 64-bit floating point type (8 bytes).

extended-float IEEE 80-bit floating point type (16 bytes).

bignum Arbitrary sized integer represented by a vector of bytes stored in an i-vector. The integer is stored as an n-byte two's-complement number. The sign of the bignum is redundantly encoded in the subtype field of the i-vector.

complex A complex number is implemented as a pair of pointers that point to the real and imaginary part of the number, respectively (16 bytes).

ratio A rational number is represented as a pair of pointers that point to the numerator and denominator, respectively (16 bytes).

B BARB: The SPUR Simulator

B.1 Purpose

BARB is a simulator of a single SPUR processor. BARB emulates the execution of instructions and maintains the current state of the processor and memory. BARB does not correctly emulate the the processor cache, the system bus, the floating point unit, or SPUR virtual memory address translation. BARB was developed to study the performance of the proposed SPUR architecture and to implement the SPUR Lisp system. Although BARB is a general CPU simulator, many additional functions have been added that are specific to SPUR Lisp.

BARB is written in C and runs on VAX 8800 and Sun 3 computers. BARB is an interactive program that prompts the user for input. Users can execute Lisp functions or arbitrary assembler instructions. Interaction with BARB allows users to trace function calls, step through instruction sequences, and print out the contents of memory and registers. Many facilities of the symbolic debugger `dbx` are present in BARB. Currently BARB has little support for executing C programs, primarily because a C compiler for SPUR has not been fully implemented. In the future, we will make the debugging features available to Lisp users also available to C users.

B.2 Organization

The files for BARB are stored in a single directory, hereafter referred to as `$barb`. BARB has several components: the actual instruction simulator, the interactive shell, and the interface to SPUR Lisp. We describe each briefly and note what files implement the component and their specific purposes.

Several files contain global definitions and are used to generate C header files as well as Lisp definition files for the compiler. These files are:

```
SPUR-OPCODES        -- numeric mapping of opcodes
                    -- used to generate opdefs.h, opsym.h, instype.h
SPUR-CMP-CONDS      -- numeric mapping of SPUR compare conditions
                    -- used to generate conddefs.h, condsym.h
SPUR-TAGS           -- numeric assignment of type tags
                    -- used to generate tags.h, tagsym.h
SPUR-TRAPS          -- numeric assignment of SPUR Lisp traps
                    -- used to generate trapdefs.h, utraps.h
SPUR-RUNTIME        -- numeric mapping of runtime kernel calls
                    -- used to generate rtdefs.h, rtsym.h
```

The simulator component contains the code that has specific knowledge of the instruction set of SPUR and the organization of SPUR registers. Trap handler simulation and simulation of floating point operations is also part of this component.

```
std.h        -- contains frequently used C macros and definitions
mem.h        -- structures and macros specific to 40-bit registers
spur.h       -- definitions for the instruction formats
asm.h        -- definitions for fixed trap handler addresses
simdefs.c    -- collected instantiations of the global variables
main.c       -- parses the command line, initializes, and simulates
sim.h        -- definitions of the machine process register formats;
                shared macros and declarations specific to barb
simulate.c   -- implements simulation of individual instructions
sim2.c       -- auxiliary definitions for simulation
regs.h       -- macros and defines for SPUR registers
sim_regs.c   -- functions to use and modify the SPUR registers
traps.h      -- definitions of machine hardware traps
traps.c      -- implements actions for hardware and software traps
float.c      -- simulation of floating point unit operations
```

The SPUR Lisp interface component contains routines that are specific to the execution of SPUR Lisp in the BARB simulator. Currently, the simulator contains a simple C-coded Lisp `read-eval-print` (REP) loop, loads Lisp FASL files, and implements many Lisp runtime kernel calls. The loader and REP loop were originally provided in C before the SPUR Lisp versions of these functions were available. Now these facilities are mostly unnecessary. The majority of the working knowledge of Lisp data structures in the simulator is stored in `sim_rt.c`.

```
rtcall.c        -- interface from SPUR Lisp runtime kernel calls
                   to the C coded implementation of the routines
sfasl.h         -- definitions of SPUR loader FOP codes
sfasl.c         -- implements the Lisp load operation in C
sim_callers.c   -- operations on function caller sets in C
uref.c          -- operations on unresolved references in C
sim_rt.c        -- large collection of simulator functions that require
                   knowledge about SPUR Lisp
```

The interactive facilities of BARB fall into two categories: debugging and tracing tools, and statistics gathering tools. In both cases there is some support for general program execution and additional support for running Lisp programs. In particular, breakpoints can be set and deleted, each instruction can be printed as it is executed, the registers and memory can be examined, and Lisp calls and returns can be monitored. In addition, support exists for attaching address trace output from BARB to an N.2 cache circuit simulator and also to DINERO, a cache execution simulator.

```
active.c    -- central dispatch for BARB shell
itrace.c    -- contains knowledge about how to print each instruction
stats.h     -- definitions for gathering statistics
stats.c     -- functions for gathering statistics
```

```
trheader.h  -- defines address trace output data structure
external.c  -- simulator actions for cache operations;
               interface to N.2 cache simulator
```

B.3 Flags to BARB

BARB accepts many flags on the command line. They are:

```
general flags:

-?, -H, -h   -- print help information
-m size      -- set size of runtime stack in 1K pages
-w size      -- set size of window stack in 1K pages
-a size      -- set size of system segment in 1K pages
-O filename  -- use file as trap handler (default: trap.handler)
-Z           -- does nothing

SPUR Lisp flags:

-N           -- load an uninitialized Lisp core image (see ZEUS)
-L filename  -- load a SPUR Lisp fasl file
-S filename  -- save Lisp core image in file
-U filename  -- use file as Lisp core image (default: icore)
-b size      -- set size of binding stack in 1K pages
-X           -- turn on garbage collection
-G size      -- set size of newspace
-D           -- use deep binding for special variables

SPUR C flags (not fully implemented):

-c           -- read C a.out file format as input (default: spur.data)
-si,so,se fname
             -- set stdin, stdout, stderr files for C (not implemented)

debugging and tracing flags:

-d           -- print trace of every instruction executed
-g           -- gather and print instruction statistics
-i           -- print calls to top-level functions always
-C           -- gather and print cache statistics
-n2          -- send all memory references to n2 simulator
-T filename  -- file where address traces are sent
```

B.4 The BARB Interactive Shell

BARB provides a set of simple interactive commands that allow the user to execute, trace, and debug SPUR Lisp and assembly programs. Currently there are no facilities for interacting with SPUR C programs. It is important to note that commands that are specific to Lisp functions will give erroneous results if used for non-Lisp functions. BARB's interactive commands fall into several categories: execution, breakpoint/tracing, printing, statistics, and miscellaneous operations.

Programs loaded into BARB can be executed in several ways. Lisp functions can be executed from the BARB shell simply by typing the form (e.g., `(+ 1 2)`). Any input beginning with a left parenthesis is assumed to be a call to a Lisp function. The command **run** starts a simple SPUR Lisp `read-eval-print` (REP) loop which accepts symbols or function calls which it evaluates. BARB commands will not work in this REP since all input is interpreted as Lisp forms. To return to the BARB shell, type `q` to the `#>` prompt. Like the reader in the BARB interactive shell, this REP loop will not accept Common Lisp package syntax or bignums. To invoke a more sophisticated REP loop, start the SPUR Lisp top-level by calling the function `(%top-level)`. The SPUR top-level is a full Common Lisp REP loop that accepts all valid Common Lisp syntax. Like the simple REP loop, BARB commands will not work within this REP loop since all input is interpreted as Lisp forms. To finish the SPUR top-level REP and return to the BARB shell, type `:quit` to the `spur>` prompt.

Another way to execute code is with the the command `ex`, which will start executing the instruction at any specified address. Unfortunately, since there is no facility for assigning a value to a register, it is impossible to set up arguments before calling a function using `ex`. `ex` takes an optional argument (in hex) that tells how many instructions to execute before stopping. Another way to stop program execution after an absolute number of instructions have been executed is with the command `si`. Be careful when using `si` that the number of instructions you specify have not already been executed, since BARB executes about 38,000 instructions of initialization before prompting for user input.

Once the program has been stopped, there are several ways to continue execution. The `c` command continues execution until the program finishes, encounters a breakpoint, or executes a number of instructions specified by `si`. Typing a number to the shell tells the program to execute for that many (decimal) instructions. Typing a carriage return executes one instruction. The command `pi` prints out the number of instructions that have executed. For instance, at startup `pi` tells the user 37,230 instructions have already executed.

```
run, r              -- start the program
ex addr [h-count]   -- execute count instrs starting at addr
c                   -- continue execution
[0-9]*, <CR>        -- execute # instructions (<CR> defaults to 1)
si number           -- stop after number instructions have executed
pi                  -- print # instructions executed
```

The second group of commands set breakpoints and trace Lisp functions. `sb` sets a breakpoint at an arbitrary hex address. sbn sets a breakpoint inside a Lisp function whose name is given. db and lb delete and list breakpoints respectively. st turns on tracing of Lisp function calls and returns. If no argument is specified, all function calls are traced. dt and lt delete and list functions traced respectively. v+ is used to increase information printed at each call and return. After one v+, arguments to calls and return values are printed. After two v+ commands, calls to trap handler routines are also printed. v- decreases information printed. ss is used to show the value of a register or memory location after every traced function call. ds and ls delete shown memories and list shown memories respectively.

Just like the where command in dbx, b produces a backtrace of Lisp function calls and the value of the v flag applies to the printed backtrace. For example, when debugging Lisp code, it is common for an error to occur during execution. To see what was being executed at the time of the error, first use v+ to increase the information printed, and then use b to print out a backtrace of calls in progress at the time of the error. Unfortunately, backtracing will not work correctly when the error occurs inside a trap handler because trap handlers manipulate the window pointers in unexpected ways.

To see the execution of the program at an even finer grain use d+. All instructions executed will be printed if the d+ flag is turned on. To turn off instruction tracing, use the d- command. The d+ command is equivalent to using the -d flag on the BARB command line.

```
sb [h-breakpt]      -- set breakpoint(s)
sbn fun-name        -- set breakpoint in function
db [h-breakpt]      -- delete breakpoint(s)
lb                  -- list breakpoints
st [s-funname]      -- set tracing of function(s)
dt [s-funname]      -- delete tracing of function(s)
lt                  -- list functions being traced
v+, v-              -- increase/decrease baktrace and trace output
ss [h-memloc]       -- set showmemory(s) on call/return trace
ds [h-memloc]       -- delete showmemory(s) on function trace
ls                  -- list showmemories
b                   -- backtrace
d+, d-              -- increase/decrease instruction tracing
```

The next group of commands is used to print out values in the memory and registers. p prints general registers (prefixed by 'r'), special registers (prefixed by 's') and floating point registers (prefixed by 't'), as well as arbitrary memory locations, which are just specified by their address. p tries to interpret the values as Lisp data and expects correct type tags. Without an argument, p prints the currently available 32 general registers. To avoid interpreting values as Lisp data use px. px takes arguments just as p does and prints the contents of memory or registers as hex numbers. pp prints all the special and floating point registers. ps prints the registers belonging to procedures on the calling stack, including possible off-chip registers. pw prints all the registers in the on-chip register windows. Both ps and pw interpret values as Lisp pointers. pf prints the number of memory references the program has made.

```
p [h-memloc]        -- print objects (indicate regs by r#, s#, t#)
px [h-memloc]       -- print objects in hex, just as 'p'
ps                  -- print the whole register stack
pw                  -- print all register windows
pp                  -- print special and floating point regs
pf                  -- print # memory references
```

The following commands are used by particular users interested in using BARB for cache performance studies, SPUR address traces, SPUR N.2 cache simulations, and general machine architecture issues.

```
g+, g-              -- increase/decrease statistics gathering
gf                  -- show cache statistics
gr                  -- reset instr set statistics
gs                  -- show instr set statistics
gx                  -- show space statistics
n2+, n2-            -- turn on / turn off n2 simulation
n2*                 -- send remaining line to n2 simulation
T+, T-              -- increase/decrease address tracing
```

These miscellaneous commands below are divided into two categories: those that require SPUR Lisp to be loaded, and those that do not. u, gc, and **inv** all require SPUR Lisp. **help** prints a summary of all BARB shell commands. **dis** disassembles the instruction at an address, but can sometimes cause an error printing out instructions. **quit** exits BARB.

```
h, help, ?          -- prints this message
mx addr data        -- modify memory location
dis [h-memloc]      -- disassemble instruction at an address
f [filename]        -- save Lisp image in filename upon exit
u                   -- show all unresolved references
gc space#           -- force a garbage collection of space number #
inv inv-args        -- check the space memory invariants
quit, q             -- exit xbarb
```

B.5 Using BARB

BARB is being developed on **eros**, a VAX 8800 computer. The source code for BARB is stored in the directory **eros:/eros6/lang/barb**. A file in this directory, **SPUR-NAMES**, is a C-shell script that defines shell variables for all the major components of SPUR Lisp. BARB users should source this file in their `.login` files.

All development work on BARB takes place on **eros**. The most recent version of all files associated with the simulator are stored on this machine. BARB has also been ported to Sun 3 workstatations. The source files for BARB are occasionally copied to a Sun 3 and recompiled, but in general, the Sun 3 version of the the simulator is several weeks behind the VAX version. The existing Sun 3 implementation of SPUR Lisp on BARB does not create and operate on floating point numbers correctly. The VAX implementation of BARB does not represent SPUR floating point numbers in the correct format, but performs correct operations on floating point values.

There are several versions of the BARB simulator available. The most stable version is kept in the file `barb`. `barb` is intended to always contain a working version of the simulator. `xbarb` contains the version of the simulator with the most recent updates. Because `xbarb` is under development, users should be aware that it may stop working at any time due to new bugs. `xbarb` is also used for debugging the Lisp system, and so all statistics gathering code has been disabled in this version. To run the simulator and gather statistics, use `wbarb`. Finally, to produce a call graph with statistics, use the program `qbarb`.

BARB requires two files to execute SPUR Lisp correctly. The first file is an `a.out` file containing the code for handling traps. This trap handler file is by default `trap.handler` in the `$barb` directory, but can be specified by a user with the `-O` option. The other file is used by SPUR Lisp and contains an initial memory image of the SPUR Lisp system. This file is created by ZEUS, the creator component of SPUR Lisp, and described elsewhere. The default name for this file is `icore` in the barb directory. Users may specify another name using the `-U` option.

B.6 Implementation Status

BARB is still undergoing changes because the focus of BARB users has shifted from the SPUR Lisp implementors to the SPUR hardware implementors. BARB will also be used by the SPUR C developers. Here we indicate remaining weaknesses in the BARB implementation and attempt to evaluate the effort required to fix them.

- BARB was implemented on a VAX computer and the floating point numbers used by BARB are VAX double precision floating point. SPUR will use IEEE floating point numbers, and BARB should be reimplemented to use host machine independent IEEE standard floating point numbers. If C-coded IEEE software floating point routines were available, using them in the simulator would require little effort. The availability of these routines is not known, nor is the effort required to implement them ourselves.

- The SPUR Lisp memory image cannot be properly ported to Suns because it contains VAX floating point numbers. When BARB uses host machine independent floating point, SPUR Lisp can be executed on Suns.

- Currently, BARB has lots of specific knowledge about the format of the SPUR Lisp memory image. Ideally, this knowledge should be shifted to a C routine that is used to initialize SPUR Lisp. This C routine could then be simulated and the proper initialization would take place. Ultimately, BARB should be fully functional for executing and debugging C programs, with additional support for executing Lisp. In its current configuration, BARB fully supports Lisp execution, with only minor support for C execution. To provide support for debugging in C, BARB needs to understand the symbol tables provided in the `a.out` files, a minor addition.

- Tracing of specific Lisp functions is not correct in the presence of `catch` and `throw`.

BARB will continue to be developed and maintained. A separate unpublished document, originating from the text of this appendix, will contain the most recent information about the BARB implementation.

C SLC: The SPUR Lisp Compiler

C.1 Purpose

The SPUR Lisp compiler (SLC) compiles SPUR Lisp source code to FASL files. The format of the FASL files is described in [WFG85]. In addition to the FASL files, the SPUR Lisp compiler optionally produces listings of error messages (extension `.err`), Spice Lisp bytecode listings (extension `.slap`), and SPUR Lisp assembly listings (extension `.spur-lap`).

C.2 Organization

The SPUR Lisp Compiler has two major pieces—the Spice Common Lisp compiler (CLC) and the SPUR translator (SPURT). The first piece is the complete Spice Lisp compiler from CMU. It has been slightly modified and a few minor bugs fixed. This compiler produces a stream of unassembled Perq bytecodes, which are translated into SPUR instructions and assembled by the SPUR translator (see Figure 21).

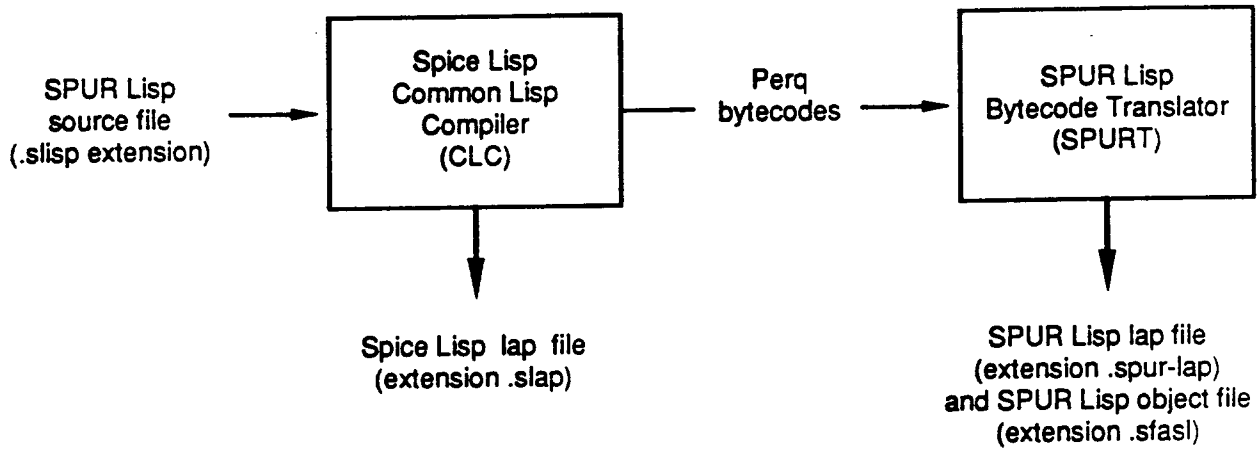


Figure 21: Phases of the SPUR Lisp Compiler. It is build from the Spice Lisp compiler (CLC) and an additional translator (SPURT).

This arrangement permitted us to build on the efforts of the Spice Lisp group and not write a complex compiler from scratch. The obvious disadvantage is that we had to use the Perq bytecodes as an intermediate form in our compiler. These bytecodes were not intended for that purpose, but they suffice.

For details about the Spice Lisp compiler, we refer the reader to the documentation that accompanies the source code. We attempted to minimize changes to the Spice compiler so that future improvements from CMU could be adopted into the SPUR compiler with little effort. The optional peephole optimizer of the Spice compiler must always be used or SPURT will produce bad code from the voluminous, unoptimized bytecodes. The Spice optional register allocator must never be used.

SPURT has four parts (see Figure 22). The first takes a sequence of Perq bytecodes

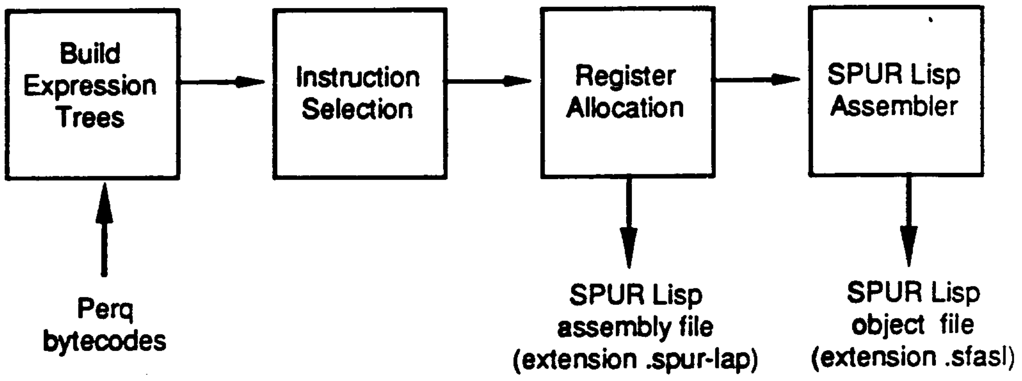


Figure 22: Phases of the SPUR Translator. The first stage builds expression trees from Perq bytecodes. The second produces a sequence of SPUR instructions from these trees. The third allocates registers for these instructions. And the fourth assembles the instructions and writes fasl files.

and builds expression trees from them. The process involves inserting explicit temporaries in place of the Perq's implicit stack locations and collecting the operands to the various bytecodes. The next stage produces SPUR code from these instruction trees. This code uses an infinite set of symbolic registers, necessitating register allocation, which allocates these temporaries to the available registers. The process is described in detail elsewhere [CH84,LH86]. The fourth and final stage assembles the SPUR instructions and writes out a FASL file. These files have the same structure as the Spice Lisp FASL files, except that the function bodies have a different format and instruction set.

The Spice Lisp portion of the compiler resides in these files:

```
aliencompile  -- contains constant folding code
clc           -- main CLC compiler
fndefs        -- information on CL functions
instdefs      -- information on Perq bytecodes
peep          -- peephole optimizer
seqtran       -- optimizations for sequence functions
trans         -- miscellaneous optimizations
typetran      -- optimizations from type specialization
```

The SPUR translator portion of the compiler resides in these files:

```
reg-alloc      -- the register allocator
spur-assem     -- SPUR assembler
spur-cmp-conds - list of SPUR condition tests
spur-inst      -- produces the SPUR instructions
spur-lisp      -- description of SPUR Lisp data structures
spur-opcodes   -- definitions of SPUR opcodes
spur-runtime   -- definitions of SPUR runtime routines
spur-tags      -- definitions of SPUR Lisp type tags
spur-traps     -- definitions of error conditions
spurt-macros   -- SPURT macros
spurt          -- bytecode parser
trans-bc       -- SPUR code generator
```

The SPUR Lisp compiler also relies on the following files to provide a standard set of macros in foreign (i.e., non-SPUR Lisp) environments:

```
defmacro
defstruct
macros
symbol-macros
```

The following files are also used:

```
assembler      -- interface to the assembler for handwritten files
misc           -- odds and ends
print-sfasl    -- print out a fasl file in a readable format
setup-slc      -- command file to load the compiler
```

C.3 Using the SPUR Lisp Compiler

The SPUR Lisp compiler is invoked by calling the function `compile-file`, as defined by Common Lisp. The SPUR Lisp implementation of `compile-file` has several additional `&keyword` parameters. The keyword `:lap-file` is a boolean argument that defaults to `nil`. If `:lap-file` is `nil`, no `.spur-lap` or `.slap` file will be generated by the compilation. `:clean-up` is a boolean keyword that defaults to `t`. If `:clean-up` is `nil`, macros defined in a compilation are not deleted after the compilation completes. `:error-file` specifies the file that error messages are written to. `:errors-to-terminal` is a boolean keyword that specifies whether or not to write compilation error messages to the terminal. `:errors-to-terminal` defaults to `t`, which means that error messages are written to the terminal. `:load` is a boolean keyword that indicates the compiled file should be loaded after being compiled. `:load` defaults to `nil`.

C.4 Implementation Status

SLC runs under Franz Lisp Opus 42 and 43[FSL*85] with the aid of a Common Lisp compatibility package. It also runs in VAX Common Lisp (VAXLISP), Franz Common Lisp (Excl), and, of course, SPUR Common Lisp. There are several additions to the SPUR Lisp compiler that would improve the performance of the compiled code it produces. These are:

- As was mentioned in section 5.4, SLC does a naive job of placing instructions in the delay slots of several two-cycle instructions. A code scheduler could be added to SLC that would improve the use of the delayed instruction slots. Studies show that a rescheduler could relocate useful instructions in about 60% of the unused delay slots, improving overall performance by up to 20%.
- SLC performs none of the standard optimization techniques such as common subexpression elimination, code motion, global interprocedural analysis, or redundant store elimination.
- SLC now ignores many of the programmer specified type declarations provided in Common Lisp. Support for recognition of numeric type declarations (especially floating point types) would significantly improve the performance of numeric applications.
- Some of us think that simple peephole optimization of the generated code would probably improve performance at a small implementation cost.

D RT: The SPUR Lisp Runtime Kernel

D.1 Organization

The SPUR Lisp runtime kernel contains functions coded in SPUR assembly language either for efficiency or because they cannot be written in SPUR Lisp. Section 6 describes each of these primitives in detail. This appendix outlines how these primitives are implemented.

The runtime primitive functions are currently implemented in two ways. About 20 of the 50 functions are implemented in SPUR assembler and called directly from other Lisp functions. Ultimately, all runtime primitives will be implemented in SPUR assembler. Because a SPUR assembler was originally unavailable, all runtime routines were initially implemented in C code as part of the BARB simulator. About 30 routines have not been rewritten in assembler. These routines are invoked through a special simulator mechanism, where calls to negative addresses are interpreted as calls to runtime routines.

The SPUR runtime routines are implemented in SPUR Lisp assembly language, and compiled into SPUR Lisp FASL files. This assembly language has a different syntax than SAS, the assembly language generated by the SPUR C compiler and used to implement the SPUR trap handlers. The syntax of SPUR Lisp assembler (SPLASM) is described in appendix G. The syntax of SAS is described in another document [Hil87]. SPLASM files are suffixed with `.sa` and translated into `.spur-lap` files before being assembled by the same functions that assemble SPUR Lisp compiler generated files. One advantage of SPLASM over SAS is that SPLASM files are interpreted using SPUR Lisp and may contain SPUR Lisp macros. Also, the SPLASM assembler understands SPUR Lisp constant vectors, and users can add their own constants to the constant vector of the function they are defining.

The SPUR assembly files reside in the `$code` directory with the rest of the SPUR Lisp system files (see appendix E). All functions defined in SPUR assembly files are interned in the `system` package. The SPUR Lisp assembly files are listed below.

```
spur-runtime     -- allocation functions for SPUR Lisp objects
spur-runtime2    -- miscellaneous runtime kernel routines
spur-misc        -- other miscellaneous runtime kernel routines
spur-arith       -- arithmetic and logical shift routines
spur-proto       -- prototype function objects for interpreted
                    functions and compiled closures
spur-deepbind    -- functions to implement deep binding of special
                    variables for multiprocessor SPUR Lisp
```

The remainder of the SPUR Lisp runtime kernel is coded in C as part of BARB, the SPUR simulator (see appendix B). Calls to these routines are not performed by simulated instructions. Instead, C code operates on the simulated SPUR memory and hardware to perform the necessary operations. The files for the C-coded portion of the runtime kernel reside in the `rt` subdirectory of the `$barb` directory. The files are organized into three major components: the SPUR Lisp runtime kernel routines, the SPUR Lisp garbage collection routines, and support routines for SPUR Lisp debugging in the BARB simulator. These files are listed below.

```
support for general programming:
    stdlib.h        -- header containing general useful C macros
    stdlib.c        -- useful C routines non-specific to SPUR Lisp

support for the SPUR Lisp runtime kernel:
    typedefs.h      -- definitions of SPUR Lisp types
    spurlib.h       -- macros defining classes of types, general defines
    spurlib.c       -- general routines specific to SPUR Lisp
    spurdefs.c      -- definitions of global variables
    alloc.h         -- defines for SPUR Lisp allocation routines
    alloc.c         -- SPUR runtime kernel allocation (mostly obsolete)
    boole.h         -- defines for boole operation
    arith.c         -- support for generic SPUR Lisp arithmetic
    logic.c         -- support for logical operations
    spurmisc.c      -- miscellaneous runtime kernel routines like throw
    string.c        -- operations on arrays of characters
    vector.c        -- operations on generic vectors

support for memory allocation and reclamation:
    gc.h            -- defines for SPUR Lisp garbage collection
    gc.c            -- generation garbage collection and full gc support
    measure.c       -- code to measure performance of memory reclamation

support for BARB simulator Lisp debugging:
    lexio.h         -- defines for YACC output to a program variable
    lisp.lex        -- simple LEX code for Lisp reader
    lisp.yacc       -- YACC to correctly interpret LEX tokens
    rep.c           -- simple read-eval-print code for BARB simulator
```

D.2 Implementation Status

There are several things that must be done to complete the SPUR Lisp runtime kernel implementation.

- The remaining C-coded runtime routines must be implemented in SPLASM. An alternative to writing these in assembler would be to take the C-coded versions and compile them using the SPUR C compiler that is becoming available.
- Some of the runtime routines are currently implemented as part of the trap handler file and not coded in SPLASM. This happened because the trap handler file was available before the SPLASM assembler was written.

- Complete numeric support must be provided in the runtime kernel routines. Support for all numeric types, and most notably ratio and floating point numbers, is not provided in all the existing kernel routines.

- The implementation of multiple values in the C-coded portion of the runtime routines is now incorrect when more than 4 values are returned from a function. The only runtime routine where this can happen is `rt-throw`, in which any number of values may be thrown.

E CODE: The SPUR Lisp System

SPUR Lisp is built in layers. The most primitive layer provides SPUR Lisp constructions that cannot be implemented in Lisp (such as `list` or `throw`), or that are so frequently executed that hand coded assembly routines are warranted. We have already discussed the contents of the SPUR Lisp kernel in appendix D. Above this layer, the SPUR Lisp system is written in SPUR Lisp. The SPUR Lisp files for this component are stored in a single directory which is referred to as the `$code` directory. The `$code` directory contains 50 files totaling more than 30,000 lines of Lisp.

Most of the SPUR Lisp system has been taken directly from Spice Lisp. Because this system is implemented in Common Lisp, much of the Spice Lisp code works for SPUR Lisp. The main differences are the functions that interface to the operating system,[16] such as `open` and `close`; the functions that manipulate system data structures, such as `load`, which modifies the unresolved reference list; and the functions that manipulate the runtime environment, such as debuggers and tracing routines. The SPUR-specific functions in the `$code` directory are in files whose names are prefixed by "spur-".

At present, some of the Spice-specific routines in the SPUR Lisp system have not been rewritten. In particular, the Common Lisp functions `trace`, `untrace`, `disassemble`, `documentation`, `step`, `time`, `inspect`, `room`, and `ed` have not been reimplemented for SPUR Lisp.

E.1 Organization

This section presents the files in the SPUR Lisp `$code` directory and briefly describes their contents. The following table describes the correspondence between the files in this directory and chapters in Guy Steele's book, *Common Lisp: the Language.*

```
Chapter 6 : Predicates
    pred            -- predicate functions
    subtypep        -- subtypep function

Chapter 8 : Macros
    backq           -- backquote and comma macros
    macromemo       -- macro memoization if *macroexpand-hook* is
                       set to memoize-macro-call
    sharpm          -- the # macro
    defmacro        -- define defmacro
    macros          -- CL macros from various chapters

Chapter 10 : Symbols
    symbol          -- symbol manipulation functions
```

[16]Spice Lisp uses the Accent operating system and SPUR Lisp uses the Sprite operating system.

Chapter 11 : Packages

package	-- main package operations
provide	-- provide, require, etc.

Chapter 12 : Numbers

arith	-- arithmetic, boolean, logic and bit operations
spnum	-- rationals, gcd, lcm, component extraction for integer and floating point numbers (float-radix/sign/digits/precision) (decode-float, scale-float, etc.)
spirrat	-- irrational, transcendental, trigonometric, exponential and logarithmic functions
rand	-- random number functions
lfloat-consts	-- implementation dependent constants for long-floats

Chapter 13 : Characters

seq	-- sequence functions
sort	-- sort and merge

Chapter 15 : Lists

list	-- list functions

Chapter 16 : Hash Tables

hash	-- hashing, hash tables

Chapter 17 : Arrays

array	-- arrays and arrays of bits

Chapter 18 : Strings

string	-- string operations
perqstrops	-- more string operations

Chapter 19 : Structures

defstruct	-- define defstruct

Chapter 20 : Evaluator

eval	-- SPUR Lisp interpreter
nspdefs	-- interpreter access to compiler primitives

Chapter 21 : Streams

defstream	-- definition of a stream
stream	-- machine independent streams

Chapter 22 : Input/Output

reader	-- Lisp reader
print	-- *print-<variable>* variables, print functions
pprint	-- pretty printer
format	-- format function
query	-- y-or-n-p, yes-or-no-p
spur-io	-- stream-level input/output implementation; uses interface defined in sprite.slisp

Chapter 23 : File System Interface

spur-filesys	-- pathname functions
load	-- reads in sfasl files
spur-uref	-- unresolved reference list operations
spur-support	-- caller set operations; interpreted functions and macros; compiled closures

Chapter 24 : Errors

error	-- requires reimplementation for SPUR
errorm	-- define error macros

Chapter 25 : Miscellaneous

misc	-- documentation function, *features*; implementation version, type, etc.
spur-patches	-- time function
trace	-- Spice debugger, not yet modified for SPUR
debug	-- ditto
frames	-- ditto
step	-- ditto

Other files :

constants	-- define Spur-specific system constants
spur-func	-- functions and macros to access fields of a function object
initialization	-- %new-initial-function; most-positive-fixnum, most-negative-fixnum; site/machine/version/system information
sprite	-- OS primitives
keyword	-- keyword parsing macro (part of compiler)
extensions	-- Spice Lisp extensions to CL

E.2 Compiling the SPUR Lisp System

The files in the SPUR Lisp system are interrelated. Macros defined in one file are used in another. To compile the SPUR Lisp system correctly in a foreign Lisp, such as Franz Lisp, the files must be compiled sequentially in a specific order. After the SPUR Lisp system is compiled, the creator component of SPUR Lisp creates an initial SPUR Lisp memory image from all the compiled files. The creation process is described in appendix H. The file `compile-all.l` in the `$code` directory indicates the order of files in the compilation, and notes reasons for the order.

Compilation of the SPUR Lisp system using the SPUR Lisp compiler requires significant resources. While the actual source code occupies just over a megabyte, the files produced in the compilation use more than 20 megabytes of disk space. The space explosion occurs because listings of the intermediate forms (`.slap`, `.spur-lap`) of the SPUR Lisp compiler are saved after compilation and used for finding bugs. If only the SPUR `sfasl` files are saved after recompiling the SPUR Lisp system, the directory is only 3 megabytes. A typical recompilation of the entire `$code` directory requires an hour of CPU time on a VAX 8800 computer.

F TH: SPUR Trap Handlers

F.1 Introduction to SPUR Traps

SPUR traps vector through locations 0x1000 to 0x10f0. The hardware that implements traps is described in [Tay85]. There is a priority ordering which corresponds to the trap vector locations. 0x1000 is highest, 0x10f0 is lowest. The page starting at 0x1000 has read/write access in kernel mode and no access in user mode. It is possible to execute instructions from this page while in user mode because instruction fetches are not restricted by access rights. The code at each trap vector location is a jump followed by a delay slot. The following text lists trap locations and a description of the traps that each location handles.

`1000`	`reset`
`1010`	`error`
`1020`	`window overflow`
`1030`	`window underflow`
`1040`	`fault, interrupt`
`1050`	`fpu exception`
`1060`	`illegal op, kernel mode violation, cxr exception`
`1070`	`fixnum tag trap (add, sub, shift, logical)`
	`fixnum or char tag trap (cmp_br)`
	`generation tag trap (store_40) (in the real hardware)`
`1080`	`integer overflow (add, sub)`
	`cmp_trap (in the real hardware)`
`1090`	`cmp_trap (current simulator)`
`10a0`	`generation trap (current simulator)`
`10b0`	
`10c0`	`eql (simulator only, real SPUR has no eql condition)`
`10d0`	`endp (simulator only, real SPUR has no endp condition)`

When a trap occurs, the cwp is incremented, the pc is saved in r10 of the new window, the next pc is saved in r16 of the new window, and the <all traps> trap enable bit in the kpsw is turned off. The following traps also change the mode from user to kernel— reset, error, fault, interrupt and `cmp_trap`.

All traps except reset and error are disabled when a trap occurs. One effect of this is to mask off page faults (or turn them into errors). Consequently, the trap vector page and certain others containing trap handler code need to be resident in memory. Traps that may correctly cause other traps need to re-enable traps during their execution. The current simulator provides the `enable` instruction for this purpose, but the actual hardware has no `enable` instruction. Because the `return_trap` instruction re-enables traps, it can be used to get the effect of `enable`.

Most trap handlers end with a `return_trap`—`jump_reg` or `return_trap`—`nop` sequence. At the start of the sequence, all traps are disabled. The effect of the `return_trap` is to enable all traps after the cwp has been decremented and after control has passed to the trailing instruction. This order of actions prevents the window overflow trap handler from getting stuck in an infinite loop. It also means that the return_trap instruction does not check for window underflow, as the normal return instruction does. In cases where window underflow is possible, the trap handlers must check for window underflow in software.

F.2 Reset and Error Traps

When a reset or error occurs the mode of the processor changes to kernel. All traps except reset are disabled. The window overflow trap is overridden so only the ten local registers (r16–r25) are known to be empty. Before calling another function, the trap handler must check explicitly for window overflow by comparing the cwp with the swp.

Before enabling traps with the kpsw <all traps> or kpsw <error> bits, you must save the kpsw <previous mode> bit. If a second reset occurs before the previous mode is saved, the previous mode will be lost. In that situation trap handler software cannot prevent the mode from being lost.

If traps are turned on during the execution of the trap handler, then before returning from it, the code must check whether the window underflow handler should be called. Traps should be turned off before calling the window underflow handler so that they will be off when control returns from it.

Before returning from the trap handler, you must have traps off and move the saved <previous mode> bit back into the kpsw <current mode> bit. This action may place the trap handler in user mode, but the handler can still continue because instruction fetches are not limited by access restrictions. The last instructions of the trap handler should be the following:

```
check for window underflow              # traps off
invoke window-underflow if necessary
move           r9, saved_2nd_pc         # any global reg
rd_kpsw
assign the <current mode> bit
wr_kpsw
return_trap    r10
jump_reg       r9
```

F.3 Window Overflow and Underflow Traps

When a window trap occurs, the user/kernel mode does not change, so the window trap handlers do not have to manipulate the kpsw <previous mode> and <current mode> bits. All traps except reset and error are disabled during the execution of window overflow and window underflow trap handlers. The window stack where overflowed registers are stored, pointed to by the swp, is readable and writable in both user and kernel modes.

The window overflow handler starts at 0x1100. The delay slot after the jump is used to load the swp into r9. If the caller's window was x, the current window is $x + 1$. The handler first assigns a youngest generation tag to r9 so that subsequent stores will not take generation traps. The handler then saves r26 of the current window at the address in r9. Next it makes a call in order to move into window $x + 2$, which is the window that needs to be copied to memory. After saving registers 11–25 using r9 + offsets, the handler returns to window $x + 1$, where it checks whether the window stack in memory has overflowed. If it has, then the handler jumps to diag_halt. Otherwise it adds 128 (16 registers $\times$ 8 bytes/register) to the swp and re-executes the instruction that caused the trap. The exit sequence from window overflow is:

```
jump_reg        r10
return_trap     r16
```

`return_trap` turns the <all traps> trap enable back on in time for the next instruction to cause a trap. Since the user/kernel mode did not change when the window trap occurred, the current mode bit is correct when control returns to window x.

The window underflow handler starts at 0x1200. The delay slot after the jump is used to load the swp into r9. If the caller's window was x, the current window is $x + 1$, and the underflow window needs to be copied to window $x - 1$. The handler first checks if the saved window stack has underflowed in memory. If it has, then the trap handler jumps to diag_halt. Otherwise, it decrements the cwp by 2 in order to move to window $x - 1$. The trap handler then subtracts 128 (16 x 8) from both r9 and the swp and loads registers 10–25 using r9 + offsets. The handler returns to window $x + 1$ by incrementing the cwp by 2. The exit sequence is the same as the exit sequence of the window overflow trap.

F.4 Tag and Generation Traps

SPUR will trap when certain operations are performed on registers with the wrong tag. For example, the `cxr` instruction traps when the type of the operand is not cons or nil. Traps also occur when arithmetic operations are performed on registers with types other than fixnum. When one of these traps occur, the Lisp system needs to perform a more complex operation than that provided by the hardware. The trap handler must recover the operands of the instruction that trapped, call a Lisp function (or the FPU) to execute the correct operation, and place the result of the operation back in the destination register of the trapping instruction.

Recovering the operands is a complex task that is implemented with an obscure code sequence in the trap handler. Operand recovery requires decrementing the cwp either explicitly or by using a `return` in order to read the registers in the previous window. While the cwp points to the previous window, traps must be disabled. This is the primary reason why the kpsw <all traps> bit is disabled at the start of every trap. If a hardware reset or error occurs while the cwp points to the previous window, then the return pc for this trap will be lost.

After the operands have been recovered, the trap handler may return control to normal user code. Traps must be enabled before doing so, but the Kpsw <all traps> bit cannot be written while in user mode. Therefore, the trap handler must make a kernel call using the `cmp_trap` instruction. (An `enable` instruction which is executable in user mode would solve the problem).

In many cases, the normal user code will want to write a substitute result into the register that would have been the destination of the instruction that trapped. To do so, it passes the substitute value to the trap handler, which places the value in the destination register.

Generation traps take place during store operations. Like tag traps, generation traps must recover the operands and then return. Because generation traps can also cause page faults, and because the generation trap can occur before the page fault is noticed, and because generation traps disable faults by default, the generation trap handler must immediately reenable faults when it is entered.

F.5 User Traps

User traps are taken when the user program executes a `cmp_trap` instruction. A field in the trapping instruction specifies a trap number that can be used to identify the trap. Trap numbers for SPUR Lisp user traps are given in 6.2. Most user traps occur when an error condition arises in the execution of a program. In this case, control is transferred to an error handling routine, which reports the error to the user.

G SPLASM: The SPUR Lisp Assembler

SPLASM is an assembler that takes SPUR assembly instructions in a Lisp-like syntax and creates Lisp FASL files. SPLASM routines can call and be called directly from any other SPUR Lisp functions. SPLASM uses the SPUR Lisp compiler (appendix C) to assemble files that have a syntax very similar to the syntax of the `.spur-lap` files generated by the compiler. SPLASM is a two pass assembler. The initial pass translates SPLASM source code (extension `.sa`) into the similar but less readable `.spur-lap` syntax. The second pass feeds the `.spur-lap` file into the back end of the SPUR Lisp compiler, which performs register allocation and writes the FASL file. An advantage of using the SPUR Lisp compiler is that symbolic temporary names can be used in the SPLASM files and the compiler's register allocator will perform efficient allocation of the temporaries.

G.1 SPLASM Syntax

Functions are defined in SPLASM using the **`defasm`** form. The syntax of **`defasm`** is specified below:

`defasm` *name arguments* `&optional` *constants* `&rest` *body* [splasm form]

`defasm` defines a new function. The argument list is only provided for documentation. The optional *constants* parameter is a list of the form `(constants` *constant-0* ... *constant-n*`)`. Each constant can be any lisp form. Constants can be referenced as they are in any SPUR Lisp function, from the constant vector pointer, which is stored in r16 at the beginning of each function. SPLASM inserts the following two instruction at the start of every function:

```
rd_spec r17, s3        ; read the pc
load    r16, r17(-8)   ; load the constants vector
```

To place *constant-0* in a register, the following sequence is used:

```
load    rC0, r16(40)   ; constant-0 is 40 bytes off r16
```

If no constants are used in a function, the second phase of the assembler removes the first two instructions from the function.

G.1.1 Instruction Syntax

The body of the **`defasm`** consists of SPUR instructions. The SPUR instruction set is described in [Tay85] and used in SAS. Note that the opcode names in SPLASM are different from the names used in SAS, although there is a one-to-one correspondence between them. Names ending in _40 in SAS are ended with _p (pointer) in SPLASM. Names ending in _32 in SAS end in `_w` (word) in SPLASM. `ld` and `st` in SAS are spelled out fully as `load` and `store`, in SPLASM. Each instruction has one of several possible formats. The formats are:

```
RRI:    (opcode reg-dest reg-source1 reg-or-constant)
RI:     (opcode reg-dest reg-or-constant)
NOP:    (opcode)
RSR:    (opcode reg-dest sp-reg reg-or-constant)
SRR:    (opcode sp-reg reg-dest reg-or-constant)
CMP:    (opcode compare-cond reg-source1 reg-or-constant label)
FCMP:   (opcode compare-cond reg-source1 reg-source2)
TRAP:   (opcode compare-cond reg-source1 reg-or-constant vector)
STORE:  (opcode reg-source2 reg-source1 offset)
JUMP:   (opcode label)
CALL:   (opcode function-name)
JREG:   (opcode reg-target offset)
```

The instructions, their formats, and an example of their typical use are presented below:

```
;;; general instructions

Instruction    Format              Example               Comment
load_p          RRI     (load_p         %r17 %r16 $40)   ; load const0
cxr             RRI     (cxr            %r19 %r20 $8)    ; take cdr of r20
load_w          RRI     (load_w         %r20 %r19 %r12)  ; 3 regs okay
store_p         STORE   (store_p        %r16 %r17 $40)   ; store const0
store_w         STORE   (store_w        %r20 %r21 $0)
test_and_set    RRI     (test_and_set   %r20 %temp $10)
add_nt          RRI     (add_nt         %r15 %temp $10)  ; named temp reg
add             RRI     (add            %r15 %temp %r12)
sub             RRI     (sub            %r15 %temp %r12)
and             RRI     (and            %r15 %temp %r12)
or              RRI     (or             %r15 %temp %r12)
xor             RRI     (xor            %r15 %temp %r12)
extract         RRI     (extract        %r20 %x %y)
sll             RRI     (sll            %dest %src1 $3)
srl             RRI     (srl            %dest %src1 $1)
sra             RRI     (sra            %dest %src1 $1)
insert          RRI     (insert         %r20 %x %y)
rd_special      RSR     (rd_special     %dest %r1 $0)    ; %r1 is s1
wr_special      SRR     (rd_special     %r1 %target $0)  ; %r1 is s1
rd_insert       RRI     (rd_insert      %dest %xxx $0)   ; rs1 is ignored
wr_insert       RRI     (wr_insert      %xxx %xxx $1)    ; rd, rs1 ignored
rd_tag          RRI     (rd_tag         %dest %src $0)
```

```
wr_tag           RRI     (wr_tag          %rX %rX $tag)     ; $tag named const
rd_kpsw          RRI     (rd_kpsw         %dest %xxx $0)    ; rs1 is ignored
wr_kpsw          RRI     (wr_kpsw         %xxx %src $1)     ; rd ignored
cmp_br_delayed   CMP     (cmp_br_delayed test %r0 %15 @label)
cmp_trap         TRAP    (cmp_trap        test %x %y $trapnum)
call             CALL    (call            lisp::vector)     ; call function
jump             JUMP    (jump            @label)           ; jump to label
call_reg         JREG    (call_reg        %x %target $0)    ; dest is ignored
jump_reg         JREG    (jump_reg        %x %target $0)
return           RRI     (return          %r10 %r0 $8)      ; see below

;;; Pseudo-instructions -- provided for purposes of gathering statistics.
;;; These instructions are not provided by the actual hardware.

Instruction     Format              Example                    Comment
load_p_spill     RRI     (load_p_spill    %r2 %r4 $8)       ; load r2 from mem
store_p_spill    STORE   (store_p_spill   %r2 %r4 $8)       ; store r2 to mem
store_p_cons     STORE   (store_p_cons    %cdr %r8 $8)      ; store into cdr
move             RI      (move            %dest %src)
move_imm         RI      (move_imm        %dest $22)
nop              NOP     (nop)

;;; Floating Point instructions

Instruction     Format              Example                    Comment
load_sgl         RRI     (load_sgl        %r3 %r16 $56)     ; load FPU r3
load_dbl         RRI     (load_dbl        %r1 %r16 $56)     ; load FPU r1
load_ext1        RRI     (load_ext1       %r6 %r16 $56)     ; load FPU r6 (hi)
load_ext2        RRI     (load_ext2       %r6 %r16 $64)     ; load FPU r6 (lo)
store_sgl        STORE   (store_sgl       %r3 %r16 $56)     ; store FPU r3
store_dbl        STORE   (store_dlb       %r1 %r16 $56)     ; store FPU r1
store_ext1       STORE   (store_ext1      %r6 %r16 $56)     ; store FPU r6
store_ext2       STORE   (store_ext2      %r6 %r16 $64)     ; store FPU r6
sync             RRI     (sync            %r0 %r0 %r0)      ; ignores operands
fadd             RRI     (fadd            %r5 %r3 %r10)
fsub             RRI     (fsub            %r5 %r3 %r10)
fmul             RRI     (fmul            %r5 %r3 %r10)
fdiv             RRI     (fdiv            %r5 %r3 %r10)
fcmp             FCMP    (fcmp            test %r1 %r3)     ; sets state
cvts             RRI     (cvts            %r5 %r3 $0)       ; ignores rs2
cvtd             RRI     (cvtd            %r5 %r3 $0)       ; ignores rs2
fmov             RRI     (fmov            %r5 %r3 $0)       ; ignores rs2
fabs             RRI     (fabs            %r5 %r3 $0)       ; ignores rs2
fneg             RRI     (fneg            %r5 %r3 $0)       ; ignores rs2
```

Instructions that compare registers, like cmp_trap and cmp_br_delayed, take as an argument the comparison to perform. The possible comparisons are listed in [Tay85], and the only difference in syntax is that eq_38 and neq_38 are replaced by eq_p and neq_p, respectively. Labels, as indicated above, are prefixed with @ (e.g., @label1). Constants are prefixed by $, and use standard Lisp syntax for their representation. For example, $32, $#x20, and $#b10000 are all valid representations for the number 32. Lisp constants may also be used. Thus $thirty-two is also valid if there is a Lisp variable defined in the SPLASM program that gives the global variable thirty-two a value.

G.1.2 Register Specifiers

Register operands can be named in a number of ways. The most direct way is to simply specify the register number (e.g., %r12). All tokens of the form %rN and %rNN, where N is a digit, are interpreted as absolute register references. Many registers also have symbolic names, which are indicated with the syntax %=NAME, where NAME is the symbolic name. The symbolic names for the various registers are presented below.

Symbolic Name	General Register	Comment
%=0	r0	r0 is fixed to be 0
%=nil	r1	Lisp nil
%=in_nargs	r11	number of in arguments
%=in0	r12	first in arg (index 0)
%=in1	r13	second in arg
%=in2	r14	third in arg
%=in3	r15	fourth in arg
%=out_nargs	r27	number of out arguments
%=out0	r28	first out arg (index 0)
%=out1	r29	second out arg
%=out2	r30	third out arg
%=out3	r31	fourth out arg

Symbolic Name	Special Register	Comment
%=upsw	s0	user program status word
%=cwp	s1	current window pointer
%=swp	s2	saved window stack pointer
%=cpu_pc	s3	program counter
%=fpu_pc	s4	pc of last FPU instr

Temporary registers can also be given symbolic names. Any register operand of the form %NAME, where NAME is not of the form %rN, %rNN, or %=NAME, will designate a temporary register that is efficiently allocated by the SLC register allocator, which can even spill temporaries to memory if necessary.

G.1.3 Using Lisp macros in SPLASM Files

Standard Lisp macros can be used in SPLASM files. If an opcode is detected that is not a valid SPUR opcode, SPLASM will look for a SPLASM macro defined with that name. SPLASM macros expand into a list of forms which are inserted as an instruction sequence in the body of the function, replacing the macro instruction. SPLASM macros are exactly like Lisp macros, and can execute an arbitrary Lisp program to determine what instruction sequence they expand into. The sequence of instructions returned is itself expanded if necessary, so that macros can contain other macros. SPLASM macros are defined using the `def-asm-macro` form. For a better understanding of the use of SPLASM macros, look at the file `asm-macros.l` in the `$splasm` directory.

G.2 Organization of SPLASM

Assembling SPLASM `.sa` files into SPUR FASL files is a two-stage process. The first stage translates files is human readable `.sa` format into `.spur-lap` format, understandable by the SLC assembler. The files that perform the `.sa` to `.spur-lap` translation are store in the `splasm` subdirectory of the `$barb` directory. These files define symbolic names for registers, macros used in the SPUR runtime system SPASM files, and symbolic names for tags and traps. They occupy 8 files containing 700 lines of Lisp code and are listed below. Files prefixed with "`def-`" are generated from files in the `$barb` directory.

```
def-sysc.l          -- symbolic names for system constants
def-traps.l         -- symbolic names for SPUR Lisp traps
def-tags.l          -- symbolic names for SPUR Lisp type tags
asm-consts.l        -- symbolic definitions for registers
asm-macros.l        -- SPASM macros used in SPUR Lisp runtime kernel
asm-lib.l           -- reader macros for special syntax
asm-trans.l         -- translation from .sa to .spur-lap files
load-asm.l          -- loads necessary files
```

The second stage of translation, from `.spur-lap` files to FASL files takes place in the back end of the SPUR Lisp compiler, documented in appendix C.

H ZEUS: The SPUR Lisp Creator

ZEUS is a program that takes the FASL files that define SPUR Lisp, and creates a Lisp memory image suitable for execution in the BARB simulator. ZEUS is derived from the Spice Lisp `genesis` program. ZEUS is really just a loader like the program `load.slisp` in `$code` that loads `.sfasl` files into the running SPUR Lisp system. Since ZEUS creates the initial SPUR Lisp image, however, ZEUS must execute initially in a non-SPUR Lisp environment. ZEUS executes in Franz Extended Common Lisp [Fra86].

ZEUS takes as input FASL files that define the Lisp image it will build (e.g. the files in the `$code` directory). ZEUS creates as output two files: a file that contains an initial memory image of the Lisp system created from the FASL files, and a file that lists each function and macro that was loaded and the address it was loaded at.

Since loading FASL files in ZEUS is very much like loading FASL files in SPUR Lisp, ZEUS borrows functions from SPUR Lisp. The source files in ZEUS come from the `$code` directory, the `$slc` directory, and from the `$zeus` directory. The files from the `$code` directory provide functions that manipulate the caller sets and unresolved reference list. They are:

```
spur-func.slisp       -- macros and functions for accessing functions
spur-support.slisp    -- caller set abstraction
spur-uref.slisp       -- unresolved reference list abstraction
```

The files in the `$slc` directory provide definitions of the SPUR opcodes, tags, traps, etc., as well as access functions for all the SPUR Lisp data types. These files are:

```
spur-tags.slisp       -- definitions of type tags
spur-traps.slisp      -- definitions of trap numbers
spur-cmp-conds.slisp  -- definitions of compare conditions
spur-opcodes.slisp    -- definitions of SPUR opcodes
spur-inst.slisp       -- definitions of SPUR instruction formats
spur-lisp.slisp       -- access functions for SPUR Lisp data types
```

The files in the `$zeus` directory redefine the loader functions provided by these other files so that the loading will take place into the memory image that ZEUS creates. The ZEUS specific files are:

```
defs-scv.cl           -- defines offsets into system constants
                         vector; generated from $barb/SPUR-SYSCONSTS
zeus-defs.cl          -- defines macros and functions necessary to
                         manipulate memory image being created
zeus.cl               -- master file: reads in FASL files, creates
                         memory image, writes output files
zeus-imports.cl       -- imports symbols from other package
zeus-types.cl         -- type specific allocation into the ZEUS
```

```
                        memory image
zp.cl                -- print function for objects in the ZEUS image
zeus-ex.cl           -- redefines functions from file in the $code
                        directory
```

ZEUS creates an initial memory image that must then be initialized by running the function `%new-initial-function` defined in the file `initialization.slisp` in the `$code` directory. The image file contains some header information, and then a sequence of triples. The header information identifies the overall size of the heap, the size of newspace, and the location of the system constants vector. From the system constants vector, all the information about the organization of memory can be determined. After the header there are zero or more triples. Each triple contains a four byte base address, a four byte upper address, and then contains the memory contents to load from the base to the upper address. The format of the memory image is shown in figure 23.

ITEM	SIZE (bytes)
`<heap size>`	4
`<newspace size>`	4
`<scv pointer>`	8
`<region 1 base address>`	4
`<region 1 top address>`	4
`<region 1 content>`	top - base
`<region 2 base address>`	4
`<region 2 top address>`	4
`<region 2 content>`	top - base
.	
.	
.	
`<region N base address>`	4
`<region N top address>`	4
`<region N content>`	top - base

Figure 23: Format of the ZEUS Memory Image File. This format is also used when the memory image is dumped from the BARB simulator.

ZEUS creates an initial memory image file (named **athena**) and this file is read into the BARB simulator with the `-N` flag, which tells the simulator that the memory image is uninitialized. The initialization function is then executed, and the new memory image is written to a file (named **acore**) which has the same format as the uninitialized memory image file. The SPUR Lisp image files range from 1.7 megabytes to 6 megabytes, depending how much application code has been loaded into the image.

The other file produced by ZEUS is the loader map, which is a listing of the locations at which all the functions and macros defined in the FASL file were loaded. The loader map file is called `lisp.map` and each line in the file has the following format:

```
function-name   (m or f)    byte-address
```

The function name field always starts in column 1 and is the print name of the symbol that names the function. If the second field is `(m)`, the object is a macro. If the second field is `(f)`, the object is a function. The byte address is a hex address of the first instruction in the function (e.g. #x80337db8).

References

[ABPW85] G. Adams, B. K. Bose, L. Pei, and A. Wang. The design of a floating-point unit. In R. H. Katz, editor, *Proc. of CS292i: Implementation of VLSI Systems*, University of California, Berkeley, CA, September 1985.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[CH84] Fredrick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 222–232, Montreal, Canada, June 1984.

[CM87] William Chang and Luis Miguel. Bignum arithmetic on SPUR: a case study in programming RISC for high performance. In David Patterson and Corinna Lee, editors, *Projects from CS252: Volume II—Systems (Caches and RISCs)*, University of California, Berkeley, CA, May 1987.

[Fra86] *Extended Common Lisp Reference Manual.* Franz Inc., 1986.

[FSL*85] John Foderaro, Keith Sklower, Kevin Layer, et al. *Franz Lisp Reference Manual.* Franz Inc., Berkeley, CA, 1985.

[Hil87] Paul Hilfinger. SPUR architecture user's manual. July 1987. Unpublished.

[HLE*85] Mark Hill, James Larus, Susan Eggers, George Taylor, et al. SPUR: a VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1985.

[IEE85] IEEE. IEEE standard 754 for binary floating-point arithmetic. IEEE, 1985.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 255–263, June 1986.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Moo85] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[OCD*87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welsh. An overview of the Sprite project. *;login*, 12(1), January 1987.

[PS81] David A. Patterson and Carlo H. Sequin. RISC-I: a reduced instruction set VLSI computer. In *Proceedings of the Eighth Symposium on Computer Architecture*, pages 443–457, May 1981.

[Sha87] Robert A. Shaw. *Improving Garbage Collector Performance in Virtual Memory.* Technical Report CSL-TR-87-323, Stanford University, March 1987.

[Sin85] Kenneth H. Sinclair. *A High-Performance Lisp Machine Garbage Collector.* Technical Report, Lisp Machine, Incorporated, October 1985.

[Ste84] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[Tay85] George Taylor. SPUR instruction set architecture. In R. H. Katz, editor, *Proc. of CS292i: Implementation of VLSI Systems*, University of California, Berkeley, CA, September 1985.

[THL*86] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the Thirteenth Symposium on Computer Architecture*, June 1986.

[UBF*84] David Ungar, Ricki Blau, Pete Foley, Dain Samples, and David A. Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the Eleventh Symposium on Computer Architecture*, June 1984.

[Ung84] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Envioronments Conference*, pages 157–167, April 1984.

[WF84] Skef Wholey and Scott E. Fahlman. The design of an instruction set for Common Lisp. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984.

[WFG85] Skef Wholey, Scott Fahlman, and Joseph Ginder. *Revised Internal Design of Spice Lisp.* Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1985.

[ZH87] Benjamin Zorn and Paul Hilfinger. Direct function calls in SPUR Lisp. July 1987. To be published as a tech report.

[ZHH*86] Benjamin Zorn, Paul Hilfinger, Kinson Ho, James Larus, and Luigi Semenzato. Features for multiprocessing in SPUR Lisp. October 1986. To be published as a tech report.

Index