SYMBOLICS, INC.

This document is COMPANY CONFIDENTIAL. It contains extremely sensitive and proprietary information. Reasonable and appropriate care should be taken to protect this information. Copies of this document are individually numbered with the recipient's name recorded. You are responsible for your copy of this document. As newer versions of this document are created, older versions will be collected for centralized destruction. Under no circumstances is this document to be copied. If you need to have someone added to the distribution list or if more copies of this document are needed, contact Linda Birch (Birch or x7605).

Document Number 3

Recipient Cassels

I-Machine Architecture Specification

Revision 2

Company Confidential

Table of Contents

List of Figures

List of Tables

1. Lisp-Machine Data Types

****
This file is confidential. Don't show it to anybody, don't hand it out to people, don't give it to customers, don't hardcopy and leave it lying around, don't talk about it on airplanes, don't use it as sales material, don't give it as background to TSSEs, don't show it off as an example of our (erodable) technical lead, and don't let our competition, potential competition, or even friends learn all about it. Yes, this means you. This notice is to be replaced by the real notice when someone defines what the real notice is.
****

The purpose of this chapter is to categorize and define all the objects that occur in I-machine memory, both visible and invisible. The categorization of a storage object is done according to its data type as specified by its type code. The definitions are presented in order by Lisp-object type.

The essence of I-machine architecture is its support of the execution of the Lisp language *at the hardware level*. This dictates the salient features of individual architectural components. In particular, I-machine data representations reflect the fact that, in a Lisp machine, every datum is a Lisp object. Every word in memory therefore contains either a Lisp object reference or part of the stored representation of a Lisp object. (The only exceptions are forwarding pointers and special markers. "Invisible" to ordinary Lisp code, these are used primarily for system memory management, including garbage collection.)

I-machine architecture is fully type coded: every word in memory has a data-type field. The function of the data-type encoding, to be described in this chapter, is to allow I-machine hardware to discriminate between the types of data it is operating on in order to handle each appropriately. More information in how I-machine instructions use different types of data is contained in another chapter. See the section "Macroinstruction Set."

The chapter first introduces the I-machine's basic storage unit. It then lists the different ways that a Lisp object can be stored in memory and describes the components of these representations. Note the interrelation between object references and stored representations of objects: while a stored representation is the target of an object reference, it can also *contain* object references as part of its structure. This relationship reflects the nature of the Lisp language.

As part of its introduction to stored representations, the chapter discusses those stored objects that are *not* object references, including those that are invisible. This includes forwarding pointers, which are used when list or structure objects are moved. These are discussed here, despite the fact that the structures they are used in have not yet been defined. The general overview of data types encountered in I-machine memory makes forward references to some structures necessary. The reader can make use of the cross references supplied to help clarify these sections.

After the introduction, the body of the chapter describes and defines the structure of each of the Lisp objects that the I-machine architecture accommodates with a specifically assigned data type. The concluding section summarizes the data-type information.

1.1 Introduction to Lisp-Machine Objects

1.1.1 Memory Words

1.1.1.1 Length and Format

Words are the basic unit of storage on the I machine. Every item in memory, including *object references* and *object representations*, is made up of one or more words. Whenever we refer to an address, it is the address of some word. More information on addresses is available elsewhere. See the section "Memory Layout and Addressing."

A word contains 40 bits, which are assigned to the following fields:

Position	*Length*	*Field Name*
<39:38>	2 bits	Cdr Code
<37:32>	6 bits	Data Type
<31:0>	32 bits	Address or Immediate Data

```
+---+--------+--------------------------------+
|CDR|  TYPE  |      ADDRESS/DATA              |
+---+--------+--------------------------------+
39  37       31                               0
```

1.1.1.2 Fields

The *data-type field* indicates what kind of information is stored in a word. Each Lisp object referenced by its own assigned data type is explained in detail in the data-type section. See the section "Data-type Definitions." The functions of data types that do not serve as Lisp object references are described in an introductory section. See the section "Components of Stored Representations."

The *address or immediate data field* is interpreted according to the data type of the word. This field contains either the address of the stored representation of an used in have not yet been defined. The general overview of data types encountered in I-machine memory makes forward references to some structures necessary. The reader can make use of the cross references supplied to help clarify these sections.

After the introduction, the body of the chapter describes and defines the structure of each of the Lisp objects that the I-machine architecture accommodates with a specifically assigned data type. The concluding section summarizes the data-type information.

1.1 Introduction to Lisp-Machine Objects

1.1.1 Memory Words

1.1.1.1 Length and Format

Words are the basic unit of storage on the I machine. Every item in memory, including *object references* and *object representations*, is made up of one or more words. Whenever we refer to an address, it is the address of some word. More information on addresses is available elsewhere. See the section "Memory Layout and Addressing."

A word contains 40 bits, which are assigned to the following fields:

Position	*Length*	*Field Name*
<39:38>	2 bits	Cdr Code
<37:32>	6 bits	Data Type
<31:0>	32 bits	Address or Immediate Data

```
+---+-------+-------------------------------+
|CDR| TYPE  |      ADDRESS/DATA             |
+---+-------+-------------------------------+
39  37      31                              0
```

Words in actual physical memory may be more than 40 bits wide to allow for parity or ECC schemes. The architecture does not require the existence of parity or ECC nor does it specify any specific ECC algorithm. Such information and its implications are part of the documentation of each implementation and of the system units that support the implementation.

1.1.1.2 Fields

The *data-type field* indicates what kind of information is stored in a word. Each Lisp object referenced by its own assigned data type is explained in detail in the

object, or the actual representation of an object. This is explained in the sections covering the individual data types.

The *cdr-code field* is used for various purposes. For header data types, the cdr-code field is used as an extension of the data-type field. For stored representations of lists, the contents of this field indicate how the data that constitute the list are stored. Other uses of the cdr-code field are for instruction sequencing. Use of the cdr code is explained in the sections on lists, headers, and compiled functions.

1.1.2 Classes of Stored Object Representations

Figure 1 illustrates the ways in which objects are represented.

The storage structures for Lisp objects are introduced here so that the reader will be able to see how the various data types function within them.

There are three fundamentally different ways that Lisp objects are stored in memory. An object can be stored

- as a list,
- as immediate data,
- or as a structure.

A *list* object is an object built out of one or more conses. Refer to the *Reference Guide to Symbolics-Lisp* for the definition of a cons. The representation consists of a block of memory words strung together by means of the cdr codes. Often the block consists of only one or two words, so it is important to avoid the overhead of having an extra header word: this is why list representation and structure representation are different. The following types of objects have list representations:

conses,
lists,
big ratios,
double-precision floating-point numbers,
complex numbers,
dynamic closures,
lexical closures, and
generic functions

Note that there is a difference between the concept of a list as a type of structure and the concept of the data type **dtp-list**. All the above data types use list structure, including cdr coding (described later). Only the object references to lists and conses use the data type **dtp-list**. (There is no dtp-cons.)

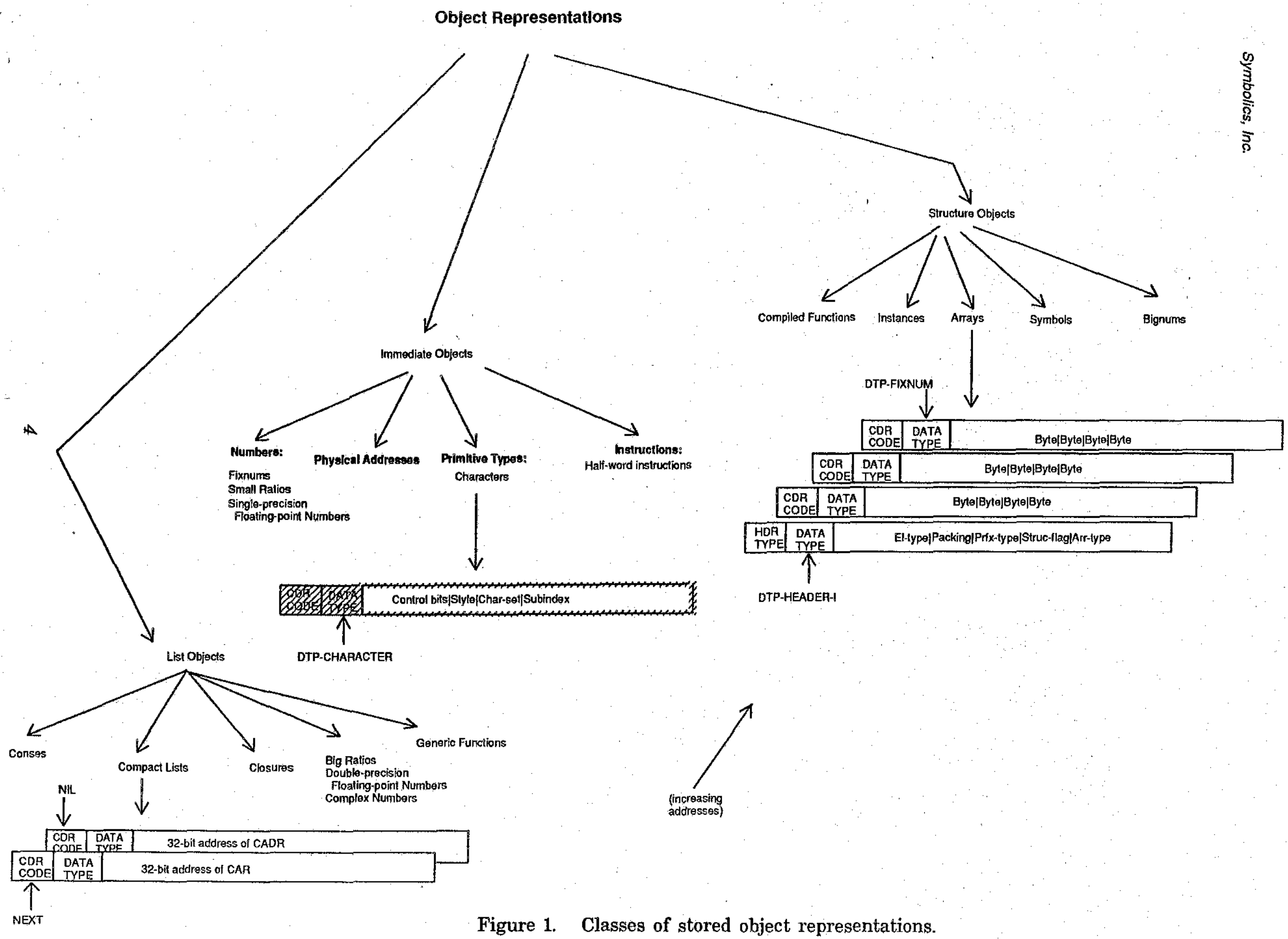


Figure 1. Classes of stored object representations.

An *immediate* object does not require any additional memory words for its representation. Instead the entire object representation is contained right in the object reference. To be an immediate object, an object type must not be subject to side-effects, must have a small representation, and must have a need for very efficient allocation of new objects of that type. The following types of objects have immediate representations:

small integers (fixnums),
single-precision floating-point numbers,
small ratios,
characters,
packed instructions, and
physical addresses

A *structure* object is represented as a block of memory words. The first word contains a header with a special data type code. Usually all words after the first contain object references. The header contains enough information to determine the size of the object's representation in memory. Further, it contains enough information about the type of the object so that a legal object reference designating this object can be constructed. Structure representation is designed to work for large objects. Some attention is also paid to minimizing overhead for small objects, but there is always at least one word of overhead. The objects represented as structures are:

symbols,
instances,
bignums,
arrays, and
compiled functions.

The stored representation of a list or structure object is contained in some number of consecutive words of memory. Each memory word within the structure may contain

an *object reference*,
a *header*,
a *forwarding pointer*, or
a *special marker*.

The data-type code identifies the word type. For example, an array is represented as a header word, which contains such information as the length of the array, and, following the header, memory words that contain the elements of the array. An object reference to an array without a leader contains the address of the first memory word in the stored representation of the array.

1.1.3 Components of Stored Representations

The components of the stored representations to be found in Lisp machine memory are either object references, headers, forwarding (invisible) pointers, or special markers.

1.1.3.1 Object References

Object references are the mechanism by which one refers to an object. The object reference is the fundamental form of data in this and any Lisp system. Object references are similar in function to the "pointers" of other languages. As noted before, an object reference can both point to the representation of a Lisp object and be a component part of such a representation.

There are three types of object references:

object references *by address*
immediate object references, and
pointers.

Figure 2 illustrates the three types of object references.

Object references by address are implemented by a memory word whose address field contains the virtual address of the stored representation of the object. Such memory words are categorized as pointer data. Examples of this type of object reference are symbols, lists, and arrays.

Immediate object references are implemented by memory words that directly contain the entire representation of the object. These are implemented by memory words that contain the object in the 32-bit immediate data field. Examples of this type of object reference are small integers (fixnums) and single-precision floating-point numbers.

Pointers are implemented in the same way as object references by address. The difference between these two types is that pointers contain the virtual addresses of locations that do not contain objects: they point instead to locations *within* objects -- for example, to the value cell of a symbol. Pointers are also categorized as pointer data.

1.1.3.2 Headers

A header is the first word in the stored representation of structure objects. The header marks the boundary between the stored representations of two objects and contains information about the object that it heads. This information is either immediate data, when the header type is **dtp-header-i,** or it is the address of some descriptive data, when the header type is **dtp-header-p.** The header-i format contains object-specific immediate data in bits <31:0>. The header-p format contains the address of an object-specific item in bits <31:0>. Object references usually use the address of an object's header as the address of the object. (The only exceptions are the object reference to a compiled function and the object

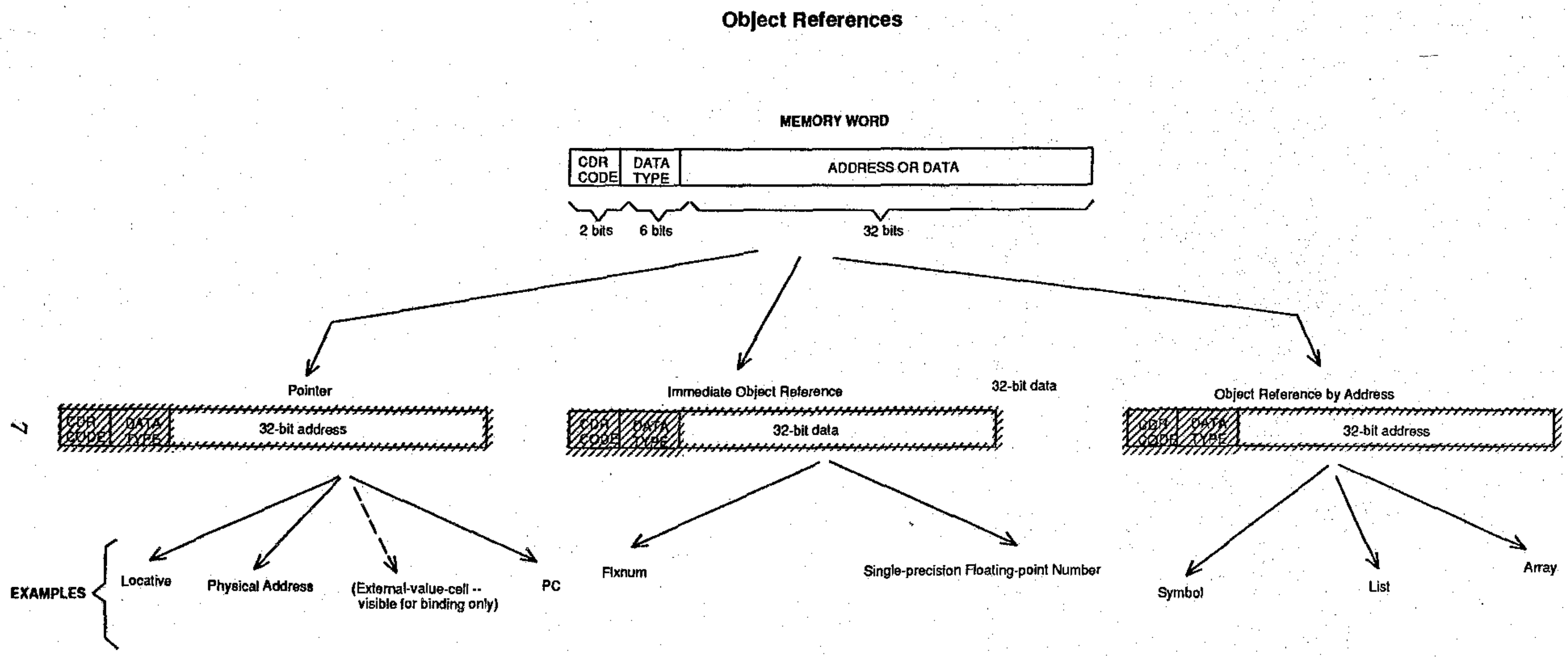


Figure 2. Three types of object references.

Symbolics, Inc.

reference to an array with a leader, in which case the reference points to a specified location inside the structure.

The cdr-code field of a header word is used as the header-type field: it distinguishes what kind of object the structure represents. The four header types for each type of header format are shown in Table 1.

Table 1. Header Types

DTP-HEADER-P

Header Type	*Symbolic Name*	*Object Type*
0	**%header-type-symbol**	Symbol
1	**%header-type-instance**	Instance
2	**%header-type-leader**	Array leader
3		*Reserved*

DTP-HEADER-I

Header Type	*Symbolic Name*	*Object Type*
0	**%header-type-compiled-function**	Compiled Function
1	**%header-type-array**	Array
2	**%header-type-number**	Number
3		*Reserved*

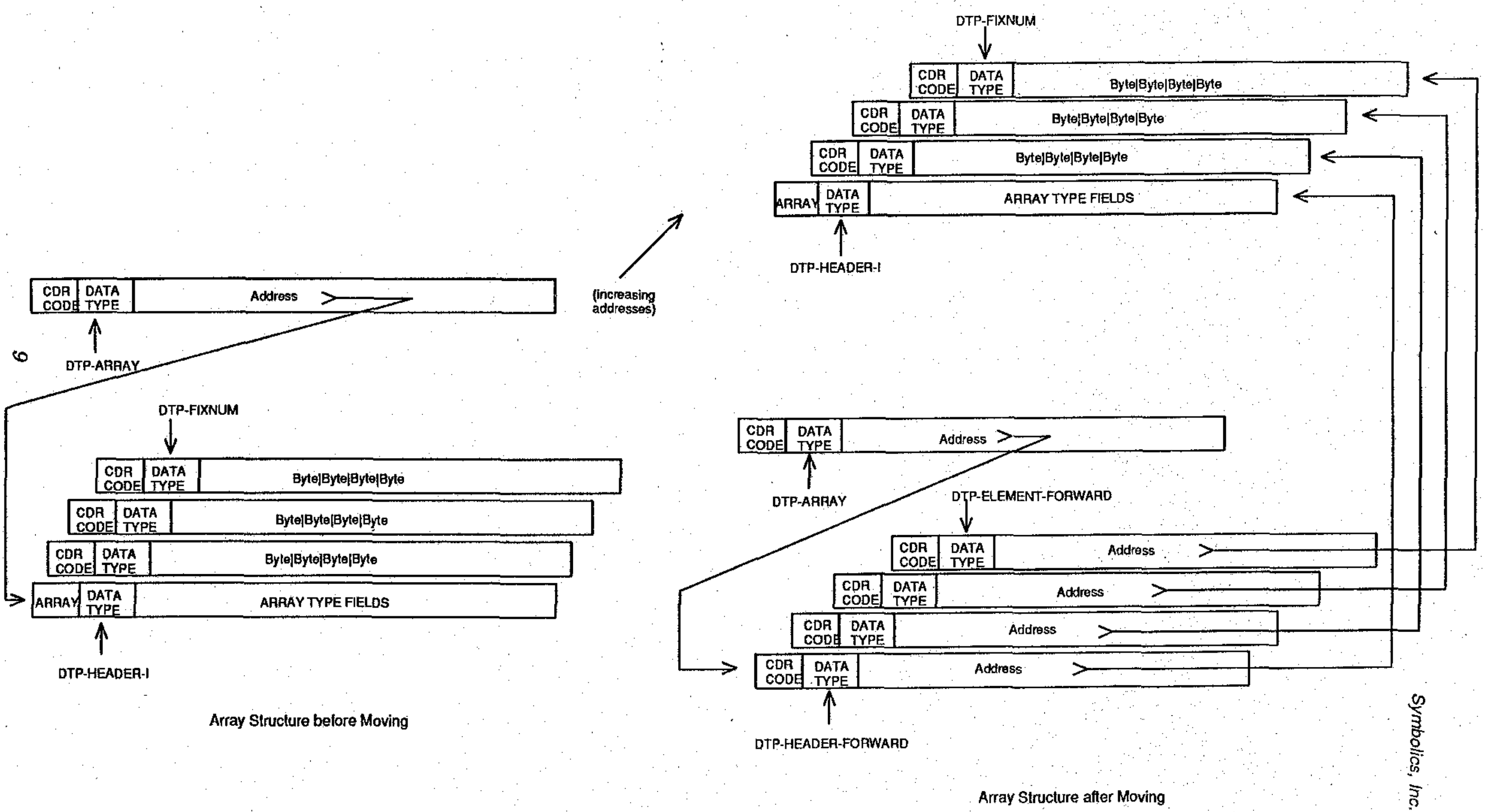
It is possible to change the memory location of an object represented by a structure. In this case, the object's header is moved to a new location and the object's old location is filled with a word of data type **dtp-header-forward,** an invisible pointer that contains the address of the new location of the reference. The object references in the locations of the old structure are all replaced with pointers of the type **dtp-element-forward**, which contain the addresses of the new locations of the objects. This arrangement allows all existing references to the object to continue to work. Refer to Figure 3. Forwarding pointers are described more fully in the next section. See the section "Forwarding (Invisible) Pointers."

1.1.3.3 Forwarding (Invisible) Pointers

A forwarding pointer specifies that a reference to the location containing it should be redirected to another memory location, just as in postal forwarding. These are also called invisible pointers. They are used for a number of internal bookkeeping purposes by the storage management software, including the implementation of extendable arrays.

The data types of the forwarding pointers are:

Use of Forwarding Pointers



Symbolics, Inc.

Figure 3. Use of forwarding pointers to move an array.

dtp-external-value-cell-pointer
dtp-one-q-forward
dtp-header-forward
dtp-element-forward

An external-value-cell pointer is used to link a symbol's value cell to a closure or instance value cell. It is not invisible to binding and unbinding. See the section "Binding Stack."

A one-q-forward pointer forwards only the cell that contains it, that is, it indicates that the cell is contained at the address specified in the address field of the **dtp-one-q-forward** word and that the cdr-code of the required data is the cdr code of the **dtp-one-q-forward** word. This pointer is used to link a symbol value or function cell to a wired cell or a compiled-function's function cell, as well as for many other applications.

A header-forward pointer is used when a whole structure is forwarded. This word marks where the header used to be, and contains the address of where the header is now. When an array with a leader is forwarded, **dtp-header-forward** pointers replace both the prefix header and the leader header. The other words of the structure are forwarded with **dtp-element-forward** pointers. The address field of an element-forward pointer contains the new address of the word that used to be there. The cdr code of the required data is stored with the relocated data -- the cdr code of the header-forward pointer is ignored. Every word of the structure except the headers contains an element-forward pointer.

A header-forward pointer is also used in connection with list representation. List representation is explained fully in another section. See the section "Representation of Lists." When a one-word cons must be expanded to a two-word cons by **rplacd,** a new two-word cons is allocated and the old one-word cons is replaced by a header-forward pointer containing the address of the new cons. (The cdr code of the header-forward pointer is required to be **cdr-nil** for garbage-collection purposes. It is ignored by **cdr** and **rplacd** operations.) The cdr code in the location containing the forwarding pointer is ignored. This is one difference between a header-forward pointer and a one-q-forward pointer: the cdr code in the location containing a one-q-forward pointer is used rather than ignored. See Figure 4. This figure illustrates how a cons whose car contains a reference to a fixnum and whose cdr is **nil** is changed when an **rplacd** instruction changes its cdr to another fixnum.

1.1.3.4 Special Markers

A special marker indicates that the memory location containing it does not currently contain an object reference. An attempt to use the contents of that location signals an error. The address field of a special marker is used by error-handling software to determine what kind of error should be reported. (The hardware does not use the special-marker address field.)

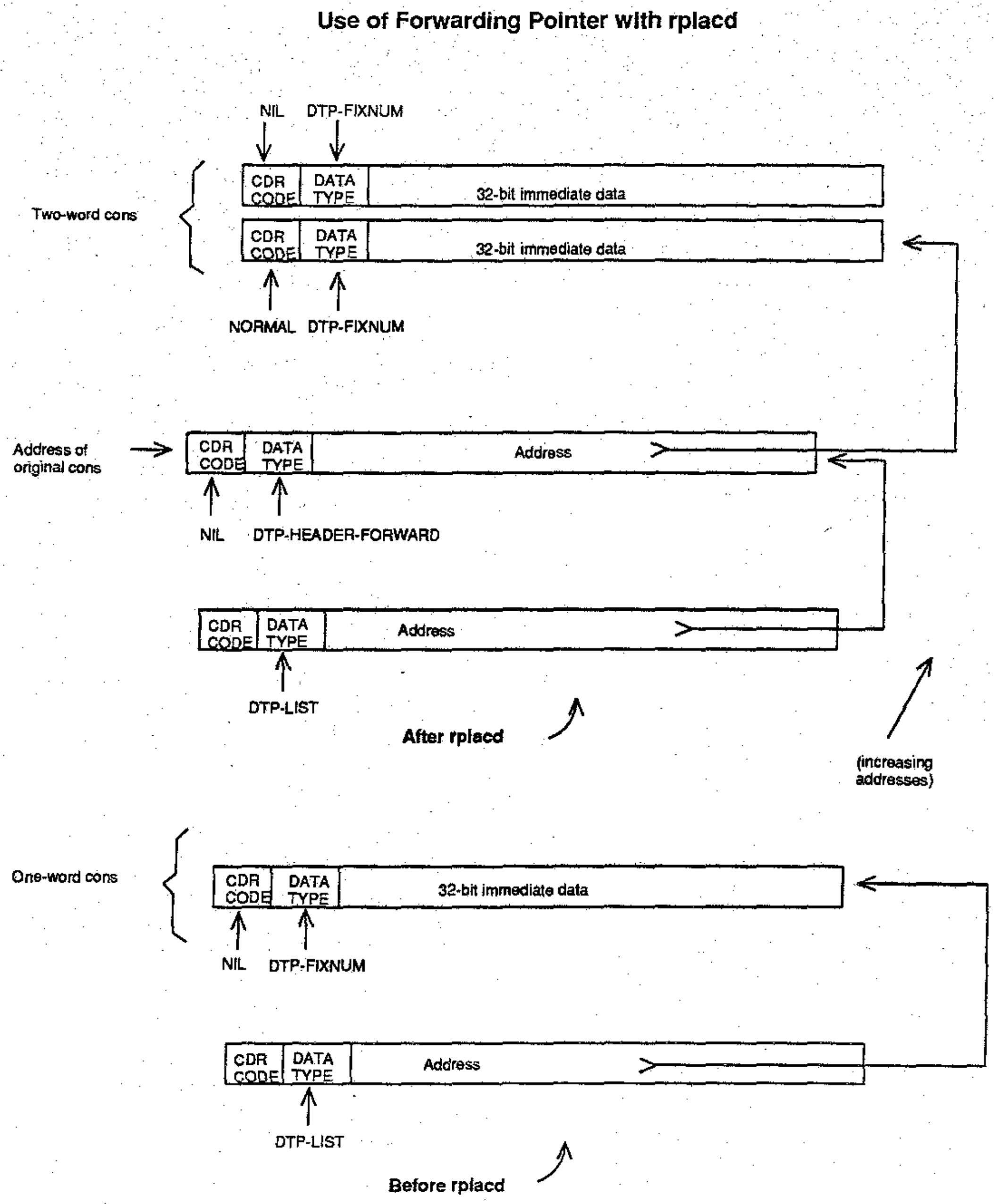


Figure 4. Use of forwarding pointers to expand a cons.

The data types of the special markers are:

dtp-null
dtp-monitor-forward
dtp-gc-forward

A null special marker is placed in the value cell or function cell of a symbol or in the instance-variable value cell in an instance, in those cases when no value has been assigned. The address field of the null marker contains the address of the name of the variable. This makes it possible for an error handler to report the name of the offending variable when an attempt to use the value of an unbound variable is detected.

A null special marker is also used to initialize a freshly-created virtual memory page in case it is accidentally accessed before an object is created in it. The address field contains the word's own address.

The encoding of the null-special-marker data type is zero. Memory that is initialized to all bits zero thus contains all null words, which will cause a trap if referenced.

The monitor special marker is intended for use with a debugging feature that will allow modifications of a particular storage location to be intercepted. See the section "Exception Handling."

A marker of type **dtp-gc-forward** is used by the garbage collector and may only appear in oldspace. When an object is evacuated from oldspace, each word of the object's former representation contains a **dtp-gc-forward** that points to the new location of that word. It is categorized here as a special marker, rather than as a pointer, since it is visible only to the garbage-collecting system, never to Lisp code.

1.1.4 Operand-Reference Classification

Immediate data	**dtp-fixnum, dtp-small-ratio, dtp-single-float, dtp-character, dtp-physical-address, dtp-packed-instruction, dtp-spare-immediate-1** (22 type codes)
Pointer data	**dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex, dtp-spare-number, dtp-instance, dtp-list-instance, dtp-array-instance, dtp-string-instance, dtp-nil, dtp-list, dtp-array, dtp-string, dtp-symbol, dtp-locative, dtp-lexical-closure, dtp-dynamic-closure, dtp-compiled-function, dtp-generic-function, dtp-spare-pointer-1, dtp-spare-pointer-2, dtp-spare-pointer-3, dtp-spare-pointer-4, dtp-even-pc, dtp-odd-pc, dtp-call-compiled-even, dtp-call-compiled-odd,**

	dtp-call-indirect, dtp-call-generic, dtp-call-compiled-even-prefetch, dtp-call-compiled-odd-prefetch, dtp-call-indirect-prefetch, dtp-call-generic-prefetch (33 type codes)
Null	**dtp-null** (1 type code)
Immediate Header	**dtp-header-i** (1 type code)
Pointer Header	**dtp-header-p** (1 type code)
HFWD	**dtp-header-forward** (1 type code)
EFWD	**dtp-element-forward** (1 type code)
1FWD	**dtp-one-q-forward** (1 type code)
EVCP	**dtp-external-value-cell-pointer** (1 type code)
GC	**dtp-gc-forward** (1 type code)
Monitor	**dtp-monitor-forward** (1 type code)
Data	The union of immediate data and pointer data (55 type codes)
Header	The union of immediate header and pointer header (2 type codes)
Immediate	The union of immediate data and immediate header (23 type codes)
Pointer	The union of pointer data, null, pointer header, HFWD, EFWD, 1FWD, EVCP, and monitor (40 type codes)
Numeric	**dtp-fixnum, dtp-small-ratio, dtp-single-float, dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex, dtp-spare-number** (8 type codes)
Instance	**dtp-instance, dtp-list-instance, dtp-array-instance, dtp-string-instance**

1.2 Data-Type Descriptions

This section defines how each type of object is represented in storage and explains how the stored representations make use of type-coded objects.

1.2.1 Representations of Symbols

The object reference to a symbol is a word of data type **dtp-symbol** or **dtp-nil**. The address field of this word contains the address of a header of type

dtp-header-p. The header is followed by four words. The header's header-type field equals **%header-type-symbol** and the address field of the header contains the address of the symbol's name, a string. The five words that constitute a symbol object, in order, are:

0	SYMBOL-NAME-CELL	address of the symbol's name
1	SYMBOL-VALUE-CELL	the value, or an unbound marker
2	SYMBOL-FUNCTION-CELL	the definition, or an unbound marker
3	SYMBOL-PROPERTY-CELL	the property list
4	SYMBOL-PACKAGE-CELL	the home package, or NIL

See Figure 5.

The special symbols **nil** and **t** reside in fixed memory locations: (vma=pma 1011000) and (vma=pma 1011005), respectively. See the section "Wired Addresses." The fixed address and separate data type for **nil** speed up operations such as predicate functions.

1.2.2 Representations of Instances and Related Data Types

The data types described in this section are used by the flavor system, which deals with flavors, instances, instance variables, generic functions, and message passing. A flavor describes the behavior of a family of similar instances. An instance is an object whose behavior is described by a flavor. An instance variable is a variable that has a separate value associated with each instance. A generic function is a function whose implementation dispatches on the flavor of its first argument and selects a method that gets called as the body of the generic function. Generic functions are described in the section on function data types. See the section "Representation of Functions and Closures." In message passing, an instance is called as a function; the function's first argument, known as the message name, is a symbol that is dispatched upon to select a method that gets called.

See the Lisp documentation for more information about flavors, instances, instance variables, and messages.

1.2.2.1 Flavor Instances

The object reference to an instance is a word of data type **dtp-instance** whose address field points to the instance structure. The stored representation of an instance consist of a header with type **dtp-header-p,** whose header-type field equals **%header-type-instance**. The words following the header of the instance are the value cells of the instance variables. They contain either object references or an unbound marker. The cdr codes are not used. The address field of the header contains the address of the hash-mask field of a flavor-description structure. This description structure is called a *flavor.*

A flavor contains information shared by all instances of that flavor. The architecturally defined fields of a flavor are:

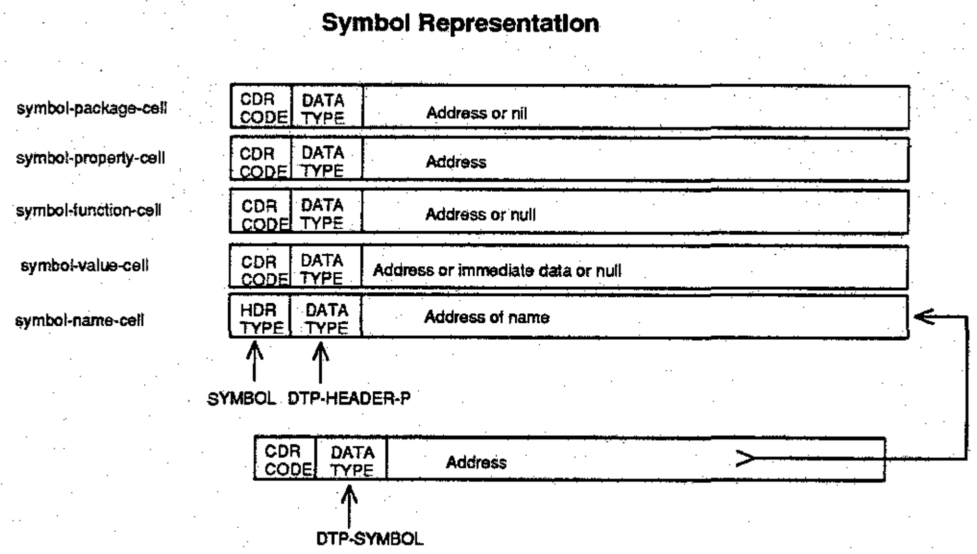


Figure 5. Structure of a symbol object.

- the array header, part of the packaging of the structure (It must be a short-prefix array format, but is not checked.)
- the named-structure symbol, part of the packaging of the structure
- the size of an instance, used by the garbage collector and by the instance referencing instructions (**%instance-ref** and the like)
- the hash mask, used by the hardware for method lookup
- the handler hash table address, used by the hardware for method lookup
- the name of the flavor, used by the **type-of** function
- additional fields known only to the flavor system

A handler table is a hash table that maps from a generic function or a message to the function to be invoked and a parameter to that function. Typically, the function is a method and the parameter is a mapping table used by that method to access instance variables. The mapping table is a simple, short-prefix ART-Q array. For speed, the format of handler tables is architecturally defined and known by hardware. Handler hash tables are packaged inside arrays, but this is software dependent, not hardware or architecture dependent.

A handler table consists of a sequence of three-word elements. The address of the first word of the first element is in the flavor. Each element consists of:

the key
This is a generic function (**dtp-generic-function**), a message name (**dtp-symbol**), or **nil**, which is a default that matches everything (**dtp-nil**).

the method
This is a program-counter value (**dtp-even-pc** or **dtp-odd-pc**) addressing the instruction at which the compiled function corresponding to the method is to be entered.

the parameter
This is a parameter that gets passed from the function or message to the method as an extra argument. If the parameter in the handler table is **nil**, the generic function or message is used as the parameter.

Method entries are normally of type **dtp-even-pc** or **dtp-odd-pc**. An interpreted method invokes a special entry point to the Lisp interpreter; this is implemented by storing the interpreter (a **dtp-even-pc** or **dtp-odd-pc**) as the method function and storing the actual method as the parameter.

Each unused three-word slot in the handler hash table, plus a fence slot at the end of the table, is filled with **nil**, a default method function, and **nil**. The default method function takes care of rehashing after a garbage collection, default handling, and error signalling.

Figure 6 illustrates the structure of an instance object, a flavor, and a handler table. Refer to the chapter on function calling to see how instances, methods, and generic functions are applied. See the section "Handler Table."

1.2.2.2 List Instances

The object reference to a list instance is a word of data type **dtp-list-instance** whose address field points to an instance structure. The instance structure for a list instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable list-manipulation instructions to operate in a generic manner on objects of the list-instance data type. See the section "Flavor Instances."

1.2.2.3 Array Instances

The object reference to an array instance is a word of data type **dtp-array-instance** whose address field points to an instance structure. The instance structure for an array instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable array-manipulation instructions to operate in a generic manner on objects of the array-instance data type. See the section "Flavor Instances."

1.2.2.4 String Instances

The object reference to a string instance is a word of data type **dtp-string-instance** whose address field points to an instance structure. The instance structure for a string instance is the same as that for an ordinary instance. Trap handlers written in Lisp enable string-manipulation instructions to operate in a generic manner on objects of the string-instance data type. See the section "Flavor Instances."

1.2.3 Representation of Characters

The object reference to a character is an immediate object of data type **dtp-character**, which contains the following fields in its data field:

Instance Representation

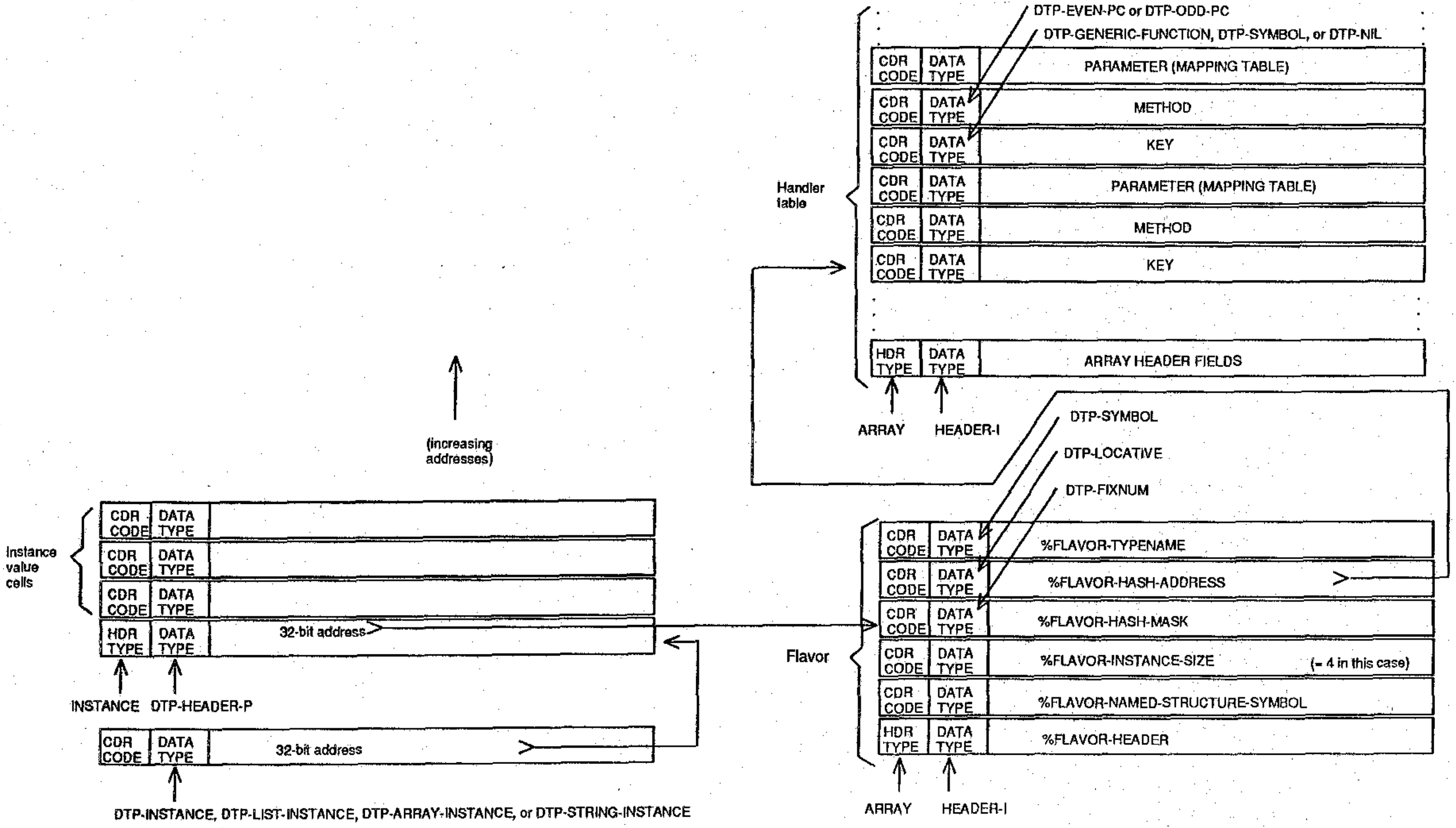


Figure 6. The structure of an instance.

Position	*Symbolic Name*	*Description*
<31:28> (4 bits)	%%CHAR-BITS	Control, Meta, Super, Hyper bits
<27:16> (12 bits)	%%CHAR-STYLE	Italic, large, bold, ...
<15:8> (8 bits)	%%CHAR-CHAR-SET	Character set
<7:0> (8 bits)	%%CHAR-SUBINDEX	Index within this character set

```
+--+------+----+-----------+---------+--------+
|CC| TYPE |BITS|   STYLE   |CHAR-SET |SUBINDEX|
+--+------+----+-----------+---------+--------+
39 37      31   27          15        7       0
```

Note that the fields in a character object are *not* used by the hardware; character format is invisible to it. The fields may change in future software.

1.2.4 Representations of Numbers

1.2.4.1 Fixnum Representation

A fixnum is represented by an immediate object whose data field contains a 32-bit, two's-complement integer. Its data type is **dtp-fixnum.**

1.2.4.2 Bignum Representation

The object reference to a bignum is a word of data type **dtp-bignum**, whose address field points to a bignum structure. The header word of the structure contains data type **dtp-header-i,** with the header-type field equal to **%header-type-number**, and **%header-subtype-bignum**. (Note that fifteen values of the 4-bit header subtype field are available for expansion.) See Figure 7. The following fields in the header word are specific to bignums:

Position	*Symbolic Name*	*Description*
<31:28>	%%HEADER-SUBTYPE-FIELD	0 for a bignum
<27>	%%BIGNUM-SIGN	0 for a positive number, 1 for a negative number
<26:0>	%%BIGNUM-LENGTH	the number of fixnums that follow

Note that the hardware does not make use of these header-word fields. Following the header is a sequence of fixnums that make up the bignum. The least-significant part of the bignum is stored in the first fixnum. The fixnums are two's complement and use all 32 bits for each digit. The bignum sign bit is the value of all the most significant bits not explicitly stored in the bignum. Therefore, -1_32. would occupy 2 words: the header with sign 1 and length 1, and a fixnum of 0. (The notation -1_32 stands for a two's complement -1 that has been multiplied by 2^32, that is, shifted left 32 places.)

```
+--------------------------------------------+
|NM|HEADER-I|BIGN|100000000000000000000000001|
+--------------------------------------------+
39                                          0

+--------------------------------------------+
|CC| FIXNUM |00000000000000000000000000000000|
+--------------------------------------------+
39                                          0
```

1_31. would also occupy 2 words: the header with sign 0 and length 1, and a fixnum that happens to be -1_31.

```
+--------------------------------------------+
|NM|HEADER-I|BIGN|000000000000000000000000001|
+--------------------------------------------+
39                                          0

+--------------------------------------------+
|CC| FIXNUM |10000000000000000000000000000000|
+--------------------------------------------+
39                                          0
```

1.2.4.3 Small-Ratio Representation

A small ratio is represented by an immediate object of data type **dtp-small-ratio**. The data field is divided into two subfields as follows:

Position	*Description*
<31:16>	form a two's-complement numerator. 0 is an illegal value.
<15:0>	is an unsigned denominator. 0 and 1 are illegal values.

```
+--+------+-----------------+-----------------+
|CC|SM-RAT|  NUMERATOR      |   DENOMINATOR   |
+--+------+-----------------+-----------------+
39 37     31                15               0
```

The illegal values are so because of either division by zero, or because the number is an integer and should be represented as such. The ratio is reduced to lowest terms. Note that the hardware does not make use of the fields of the small ratio.

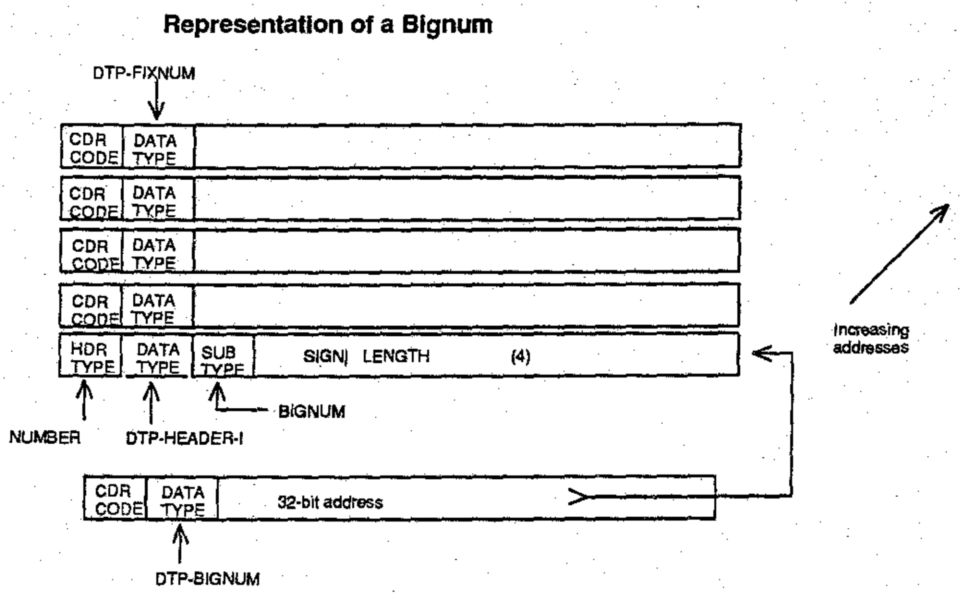


Figure 7. Structure of an object of type **dtp-bignum**

1.2.4.4 Big-Ratio Representation

The object reference to a big ratio is a word of data type **dtp-big-ratio**, whose address field points to a cons pair. The car of the cons contains the numerator of the ratio, and the cdr contains the denominator. As with small ratios, a numerator of 0, or a denominator of 0, 1, or a negative number, is illegal. The ratio is reduced to lowest terms. See Figure 8.

1.2.4.5 Single-Precision Floating-Point Representation

A single-precision floating-point number is represented as an immediate object of data type **dtp-single-float** whose data field contains a 32-bit IEEE single basic floating-point number. The following fields are defined:

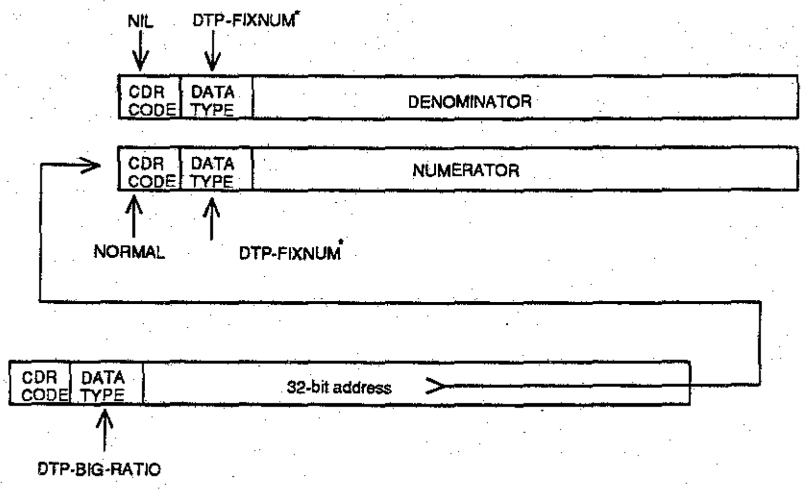
Position	*Symbolic Name*	*Description*
<31>	%%SINGLE-SIGN	0 for positive numbers, 1 for negative numbers
<30:23>	%%SINGLE-EXPONENT	excess-127 exponent
<22:0>	%%SINGLE-FRACTION	positive fraction, with hidden 1 on the left

```
+--+------+-+---------+----------------------+
|CC|SNG-FL|S| EXPONENT|     FRACTION         |
+--+------+-+---------+----------------------+
39 37     31          22                    0
```

1.2.4.6 Double-Precision Floating-Point Representation

The object reference to a double-precision floating-point number is a word of data type **dtp-double-float**. The address field of the double-float word contains the address of a cons pair. See Figure 9. The data fields in the words of the cons pair hold two fixnums, containing the sign, exponent, and fraction as packed fields. The most-significant word is stored first, violating normal byte-order conventions. The second fixnum contains the low 32 bits of the fraction. The first fixnum contains the following fields:

Representation of a Big-ratio



* The numerator or denominator could also be a bignum.

Figure 8. Representation of a big ratio.

Position	*Symbolic Name*	*Description*
<31>	%%DOUBLE-SIGN	0 for a positive number, 1 for a negative number
<30:20>	%%DOUBLE-EXPONENT	excess-1023. exponent
<19:0>	%%DOUBLE-FRACTION-HIGH	top 20 bits of fraction (excluding the hidden bit)

```
+--+------+-+----------+-------------------+
|CC| FXNM |S| EXPONENT |    FRACTION-HIGH  |
+--+------+-+----------+-------------------+
39 37     31           19                  0
```

The second fixnum contains one field:

Position	*Symbolic Name*	*Description*
<31:0>	%%FRACTION-LOW	bottom 32 bits of fraction

```
+--+------+--------------------------------+
|CC| FXNM |        FRACTION-LOW            |
+--+------+--------------------------------+
39 37     31                               0
```

This conforms to the IEEE standard 64-bit representation. In non-generic code double-precision floating-point numbers are often represented as a pair of fixnums. Avoiding the normal in-memory object representation saves consing overhead.

1.2.4.7 Complex-Number Representation

The object reference to a complex number is a word of data type **dtp-complex**, whose address points to a cons pair. The car of the cons contains the real part of the number, and the cdr contains the imaginary part. See Figure 10.

1.2.4.8 The Spare-Number Type

An object reference using **dtp-spare-number** can be employed by software to implement additional numeric data types. Functions that require numeric data types as arguments will behave properly (usually trapping out to user-defined handlers) with **dtp-spare-number** operands.

1.2.5 Representations of Lists

The object reference to a list is a word of data type **dtp-list**, whose address field contains the address of a word that contains the car of a cons. The storage representation of a list is usually a linked collection of conses. Refer to the *Reference Guide to Symbolics Lisp* for a complete description of conses and lists. In

Representation of Double-precision Floating-point Number

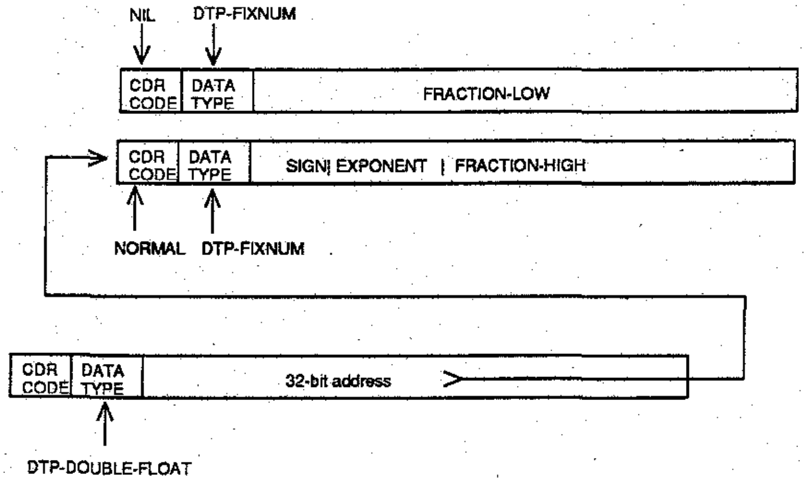


Figure 9. Representation of a double-precision floating-point number.

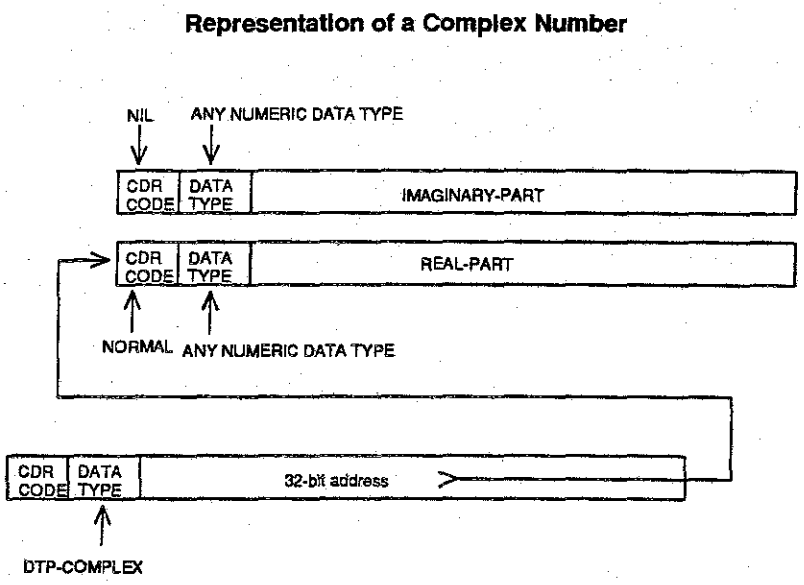


Figure 10. Representation of a complex number.

compact form, however, a list can be stored in a sequence of adjacent memory words. See Figure 11.

The cdr-code tag of a memory word that constitutes an element of a list specifies how to get the cdr of its associated cons according to whether the list is stored in normal linked-list form or in compact form. The cdr-code tag works as follows:

Code	*Symbolic Name*	*Description*
0	**cdr-next**	Increment the address to get a reference to the cdr, itself a cons. This is used for compact lists.
1	**cdr-nil**	The cdr is **nil.** This is used for both kinds of list.
2	**cdr-normal**	Fetch the next memory word; it contains a reference to the cdr. This is used for normal lists.
3		*(illegal)*

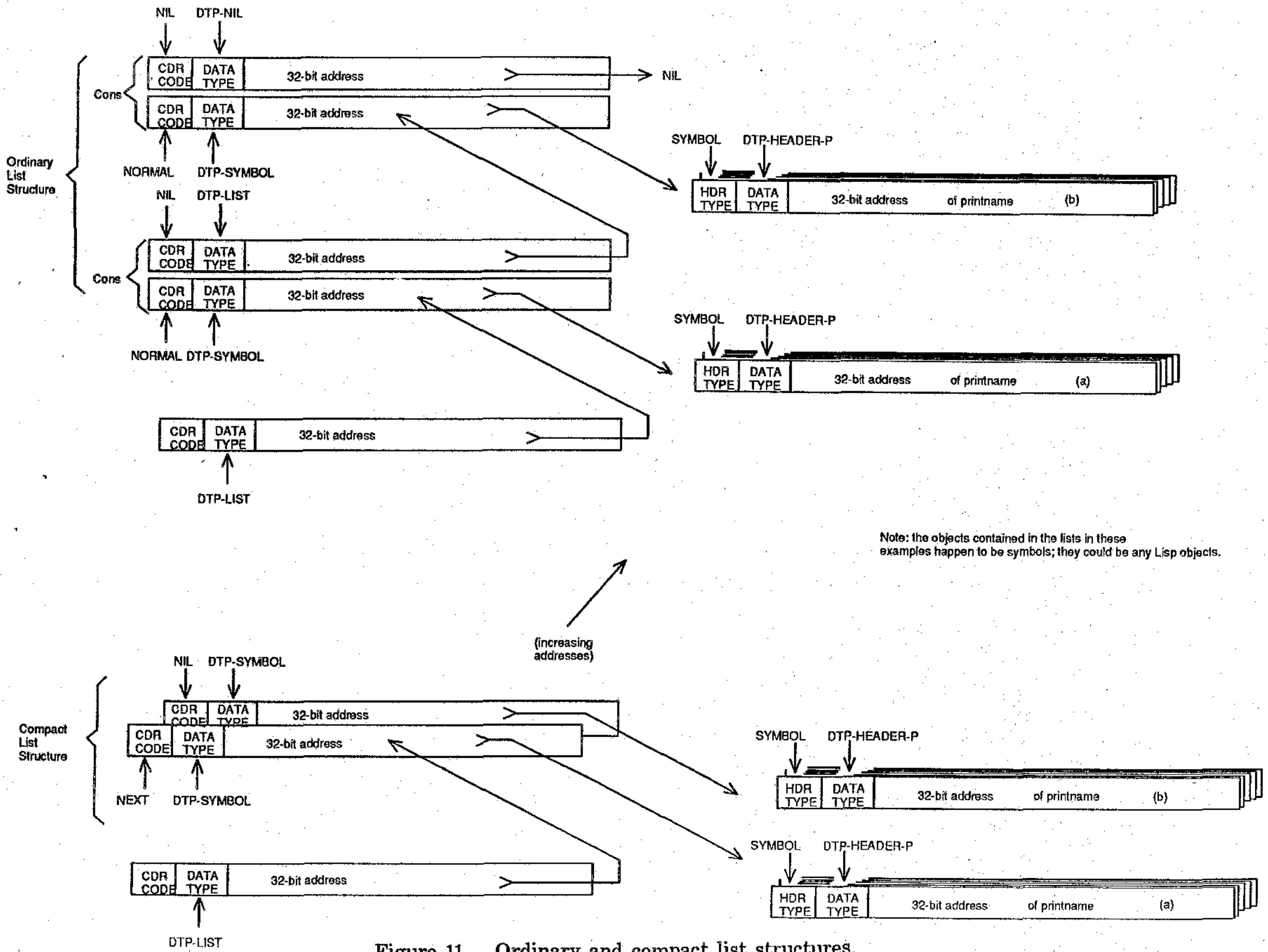
A typical, that is, not compact, two-word cons has **cdr-normal** in the cdr-code tag of its first word and **cdr-nil** in that of its second. The **car** and **cdr** operations ignore the cdr code in the second word, but it is helpful to the garbage collector.

In general, a compact list representation consists of a contiguous block of one or more memory words. The cdr code of the last word is always **cdr-nil.** The cdr code of the second-to-last word may be **cdr-normal** or **cdr-next.** The cdr code of each of the remaining words is **cdr-next.** Note that when a cons consists of exactly two words, the **cdr-normal** form is used in its representation, and the cdr code of the second word is always **cdr-nil.** In a two-element list consisting of two words, the cdr code of the first word is **cdr-next.**

Note that a **dtp-list** pointer can point into the middle of a list representation. This happens any time **cdr-next** is used; for instance, if a list of four elements is fully cdr-coded -- that is, it is stored in compact form -- its representation consists of four words. The contents of each word is an element of the list. The cdr codes of the first three words are **cdr-next**; the cdr code of the last word is **cdr-nil.** An object reference to the cddr of this list has data type **dtp-list** and the address of the third word. The garbage collector protects the entire block of storage if any word in it is referenced. See Figure 12.

The **rplacd** operation interacts with cdr coding. An illustration of this was presented in an earlier section. See the section "Forwarding (Invisible) Pointers." **rplacd** of a cons represented with **cdr-normal** simply stores into the second word. But **rplacd** of a cons represented with **cdr-next** or **cdr-nil** must change the representation so that the cdr is represented explicitly before it can be changed.

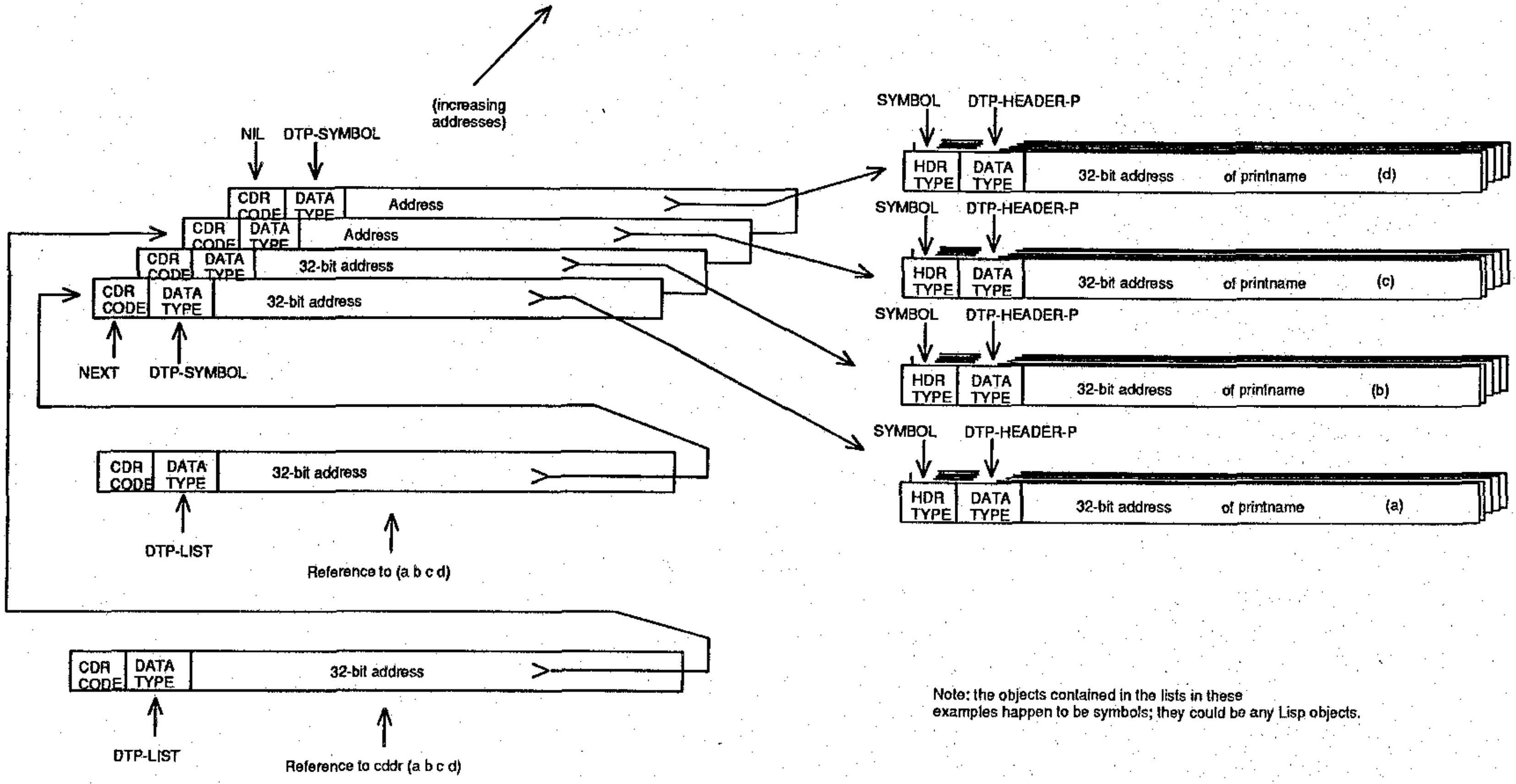
List Representations of the List (a b)



Note: the objects contained in the lists in these examples happen to be symbols; they could be any Lisp objects.

Figure 11. Ordinary and compact list structures.

List Representations of the Lists (a b c d) and cddr (a b c d)



Note: the objects contained in the lists in these examples happen to be symbols; they could be any Lisp objects.

Figure 12. An object reference to the **cddr** of a list.

There is one exception; if the cdr is being changed to **nil,** the **cdr-nil** cdr code is used to represent it. Use of **rplacd** can split an object representation into two independent object representations, one of which might then be garbage-collected.

dtp-header-forward is used to implement list forwarding. If the data-type tag (of the car) is **dtp-header-forward,** the cdr code is ignored (except by the garbage collector, which expects it to be **cdr-nil**). The address in the forwarding pointer points to a pair of words that contain the car and cdr.

1.2.6 Representations of Arrays and Strings

The object reference to an array or string is a word with data type **dtp-array** or **dtp-string.** The representation of arrays described here does not apply to object references with data type **dtp-array-instance** or **dtp-string-instance.**

Whether an array is referred to by **dtp-array** or **dtp-string** has no effect on its stored representation: the data type of the object reference simply serves to make the **stringp** predicate faster.

An array is a structure consisting of a prefix followed by optional data. (Data does not follow the prefix of an array structure if, for example, the array is displaced.) A prefix is defined to be a word whose data type is **dtp-header-i** and whose header type is **%header-type-array**, followed by zero or more additional words. The prefix defines the type and shape of the array. This is similar to the 3600. The detailed format of the prefix is different from the 3600, and simpler. The data is a sequence of object references or of fixnums containing packed bytes.

The byte fields in a prefix header's 32-bit immediate field are:

Position		*Symbolic Name*	*Description*
<31:26>	6	ARRAY-TYPE-FIELD	Combination of fields below
<31:30>	2	ARRAY-ELEMENT-TYPE	Element type, one of: fixnum, character, boolean, object-reference.
<29:27>	3	ARRAY-BYTE-PACKING	Byte packing. Base 2 logarithm (0 to 5) of the number of elements per word 6 or 7 in this field is undefined.
<26>	1	ARRAY-LIST-BIT	1 in ART-Q-LIST arrays, 0 otherwise
<25>	1	ARRAY-NAMED-STRUCTURE-BIT	1 in named-structures, 0 otherwise
<24>	1	ARRAY-SPARE-1	(spare for software use)
<23>	1	ARRAY-LONG-PREFIX-BIT	1 if prefix is multiple words
<22:15>	8	ARRAY-LEADER-LENGTH-FIELD	Number of elements in the leader
<14:0>	15	ARRAY-	Use of these bits depends on the prefix type, as described below

in the defini

```
+--+------+--+---+-+-+-+-+------+----------------+
|AR|HDR-I |TY|BPB|L|S|-|P|L-LEN|                |
+--+------+--+---+-+-+-+-+------+----------------+
39 38    31 30  27      23     15                0
```

Bits <31:27> correspond to the same bits of the control word of an array register. Array registers are discussed in the following section. See the section "I-Machine Array Registers.". Bits <26:24> are not used by hardware. Bits <31:27,23> enable various special pieces of hardware (or microcode dispatches). Bits <22:0> are used by hardware under microcode control. Bits <31:26> are sometimes grouped together as ARRAY-TYPE-FIELD.

Some arrays include *packed data* in their stored representation. For example, character strings store each character in a single 8-bit byte. This is more efficient than general arrays, which require an entire word for each element. Accessing the *n*th character of a string fetches the *n*/4th word of the string, extracts the mod(*n*,4)th byte of that word, and constructs an object reference to the character whose code is equal to the contents of the byte. Machine instructions in compiled functions are stored in a similar packed form. For uniformity, the stored representation of an object containing packed data remains a sequence of object references. Each word in an array of element-type fixnum, boolean, or character is an immediate object reference, data type **dtp-fixnum**, whose thirty-two bits are broken down into packed fields as required, such as four 8-bit bytes in the case of some character strings.

An array can optionally be preceded by a leader, a sequence of object references that implements the array-leader feature. If there is a leader, the leader is preceded by a header of its own, tagged **dtp-header-p** and **%header-type-leader**; the address field of this header contains the address of the array's main header -- that is, the address of the header of the array prefix. Note that if an array has a leader, the address field of an object reference designating that array contains the address of the main header, the one after the leader, not the address of the header at the beginning of the array's storage, before the leader. Refer to the diagram, Figure 13.

The address of leader element **i** of an array whose address is **a**, regardless of whether the prefix is long or short, is given by **(- a i 1)**.

The two array formats (**%array-prefix-short** and **%array-prefix-long**) are provided to optimize speed and space for simple, small arrays, which are the most common. Wherever possible fields have been made identical in both formats to simplify the implementation.

Description of the two prefix types:

%array-prefix-short:

Position	*Bits*	*Symbolic Name*	*Description*
<14:0>	15	ARRAY-SHORT-LENGTH-FIELD	Length of the array.

```
  +--+-----+--+---+-+-+-+-+-----+-----------+
  |AR|HDR-I|TY|BPB|L|S|-|0|L-LEN| AR-LENGTH |
  +--+-----+--+---+-+-+-+-+-----+-----------+
39 38     31 30  27    23      14           0
```

The prefix is one word. The array is one-dimensional and not displaced, but may have a leader. Most common arrays including defstructs, editor lines and most arrays with fill-pointers use this type. (You can find out about fill pointers by using the Document Examiner, or refer to the *Reference Guide to Symbolics Lisp.*) See Figure 13.

The address of data element **i** of a short-prefix array whose address is **a** and whose ARRAY-BYTE-PACKING field is **b** is given by **(+ a (ash i (- b)) 1)**. When **b** is greater than zero, packed array elements are stored right-to-left within words, thus the right shift to right-justify data element **i** is **(ash (logand i (1- (ash 1 b))) (- 5 b))**.

Arrays with Prefix Type %array-prefix-short

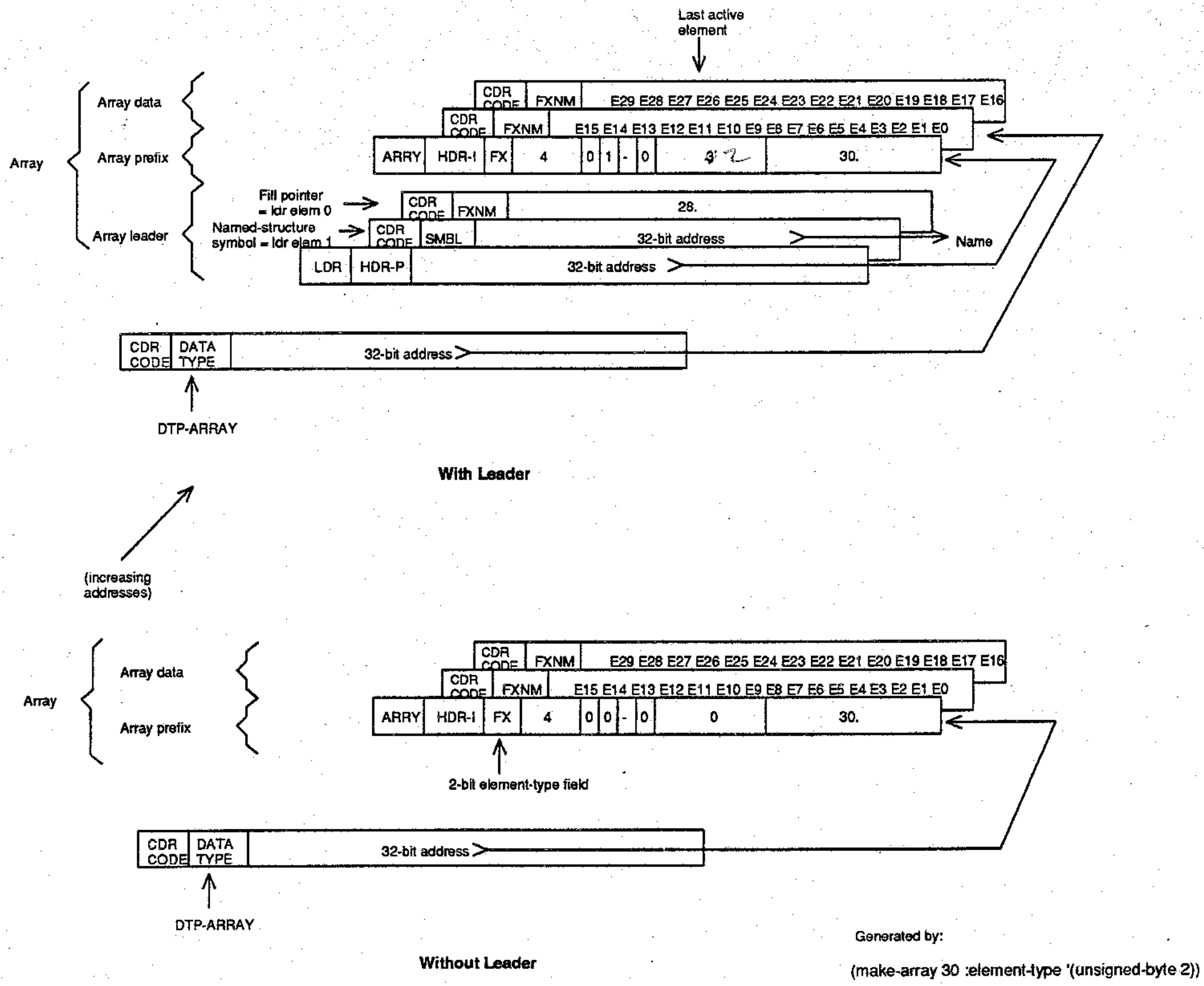


Figure 13. Short-prefix arrays with and without leaders.

%array-prefix-long:

Position	*Bits*	*Symbolic Name*	*Description*
<14>	1	ARRAY-DISPLACED-BIT	0 for normal array, 1 for displaced array.
~~<13:3>~~	~~12~~	~~ARRAY-LONG-SPARE~~	~~Spare.~~
<2:0>	3	ARRAY-LONG-DIMENSIONS-FIELD	Number of dimensions.

```
+--+-----+--+---+-+-+-+-+-----+-+-----+----+
|AR|HDR-I|TY|BPB|L|S|-|1|L-LEN|D|SPARE|DIMS|
+--+-----+--+---+-+-+-+-+-----+-+-----+----+
39 38    31 30  27      23     1413    2    0
```

The long prefix format is used for displaced arrays (including indirect arrays), arrays that are too large to fit in the short-prefix format, and multidimensional (including zero-dimensional) arrays. The first word of the prefix contains the number of dimensions in place of the length of the data. The total length of the prefix is **(+ 4 (* d 2))** where **d** is the number of dimensions.

The second word of the prefix is the length of the array. For conformally displaced arrays, this is the maximum legal linear subscript, not the number of elements (which may be smaller).

The third word of the prefix is the index offset. This word is always present, even for non-indirect arrays. Zero should be stored here in non-displaced arrays, since the this word is always added to the subscript. Always having an index offset keeps the format uniform and allows the feature that displaced arrays of packed elements can be non-word-aligned.

The fourth word of the prefix is the address of the data. This is a locative to the first word after the prefix for normal arrays, except for normal arrays with no elements, in which case it is a locative to the array itself to avoid pointing to garbage. For displaced arrays, this is a locative or a fixnum. For indirect arrays, this is an array.

The remaining words of the prefix consist of two words for each dimension. The first word is the length of that dimension and the second word is the value to multiply that subscript by. Note that this is different from the 3600. See Figure 14.

A one-dimensional array with a subscript multiplier not equal to 1 cannot be encached in an array register. Currently the software considers such arrays illegal and will never create one.

%array-prefix-long:

Position	*Bits*	*Symbolic Name*	*Description*
<14>	1	ARRAY-DISPLACED-BIT	0 for normal array, 1 for displaced array.
<13>	1	ARRAY-DISCONTIGUOUS-BIT	0 for normal array, 1 for conformal array.
<12:3>	12	ARRAY-LONG-SPARE	Spare.
<2:0>	3	ARRAY-LONG-DIMENSIONS-FIELD	Number of dimensions.

```
 +--+-----+--+---+-+-+-+-+-----+-+-----+----+
 |AR|HDR-I|TY|BPB|L|S|-|1|L-LEN|D|SPARE|DIMS|
 +--+-----+--+---+-+-+-+-+-----+-+-----+----+
39 38    31 30  27     23     1413    2     0
```

The long prefix format is used for displaced arrays (including indirect arrays), arrays that are too large to fit in the short-prefix format, and multidimensional (including zero-dimensional) arrays. The first word of the prefix contains the number of dimensions in place of the length of the data. The total length of the prefix is **(+ 4 (* d 2))** where **d** is the number of dimensions.

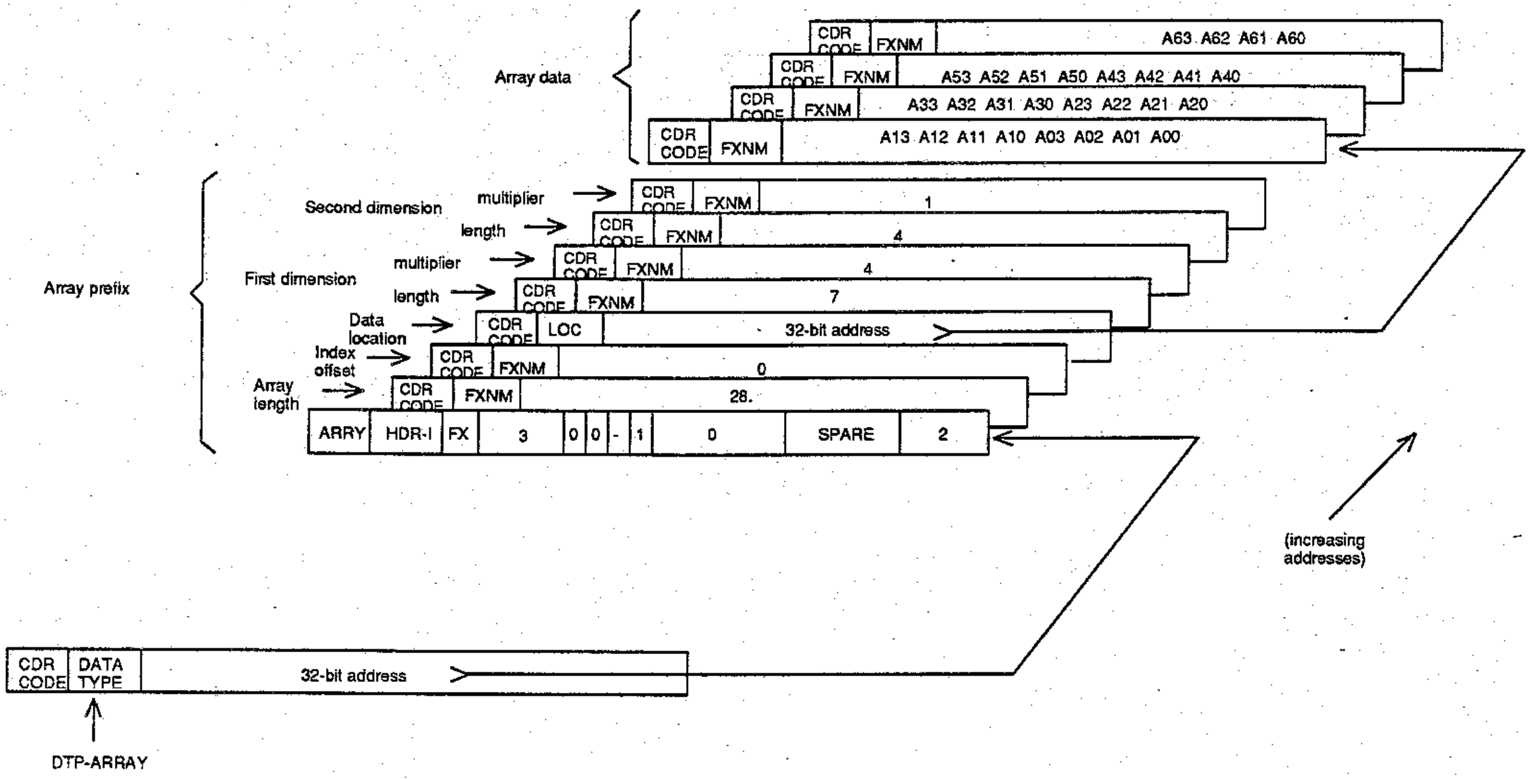
The second word of the prefix is the length of the array. For conformally displaced arrays, this is the maximum legal linear subscript, not the number of elements (which may be smaller).

The third word of the prefix is the index offset. This word is always present, even for non-indirect arrays. Zero should be stored here in non-displaced arrays, since the this word is always added to the subscript. Always having an index offset keeps the format uniform and allows the feature that displaced arrays of packed elements can be non-word-aligned.

The fourth word of the prefix is the address of the data. This is a locative to the first word after the prefix for normal arrays, except for normal arrays with no elements, in which case it is a locative to the array itself to avoid pointing to garbage. For displaced arrays, this is a locative or a fixnum. For indirect arrays, this is an array.

The remaining words of the prefix consist of two words for each dimension. The first word is the length of that dimension and the second word is the value to multiply that subscript by. Note that this is different from the 3600. See Figure 14.

A one-dimensional array with a subscript multiplier not equal to 1 cannot be encached in an array register. Currently the software considers such arrays illegal and will never create one.



Two-Dimensional Array

Figure 14. A two-dimensional array.

The way you tell a displaced/indirect array from a normal array is by checking the array-displaced bit of the array header (assuming the array has its long prefix bit set). Indirect arrays can be can detected by the data type tag of the fourth word. Figure 15 shows a simple displaced array, while the figure in Figure 16 shows a one-dimensional array indirected to another two-dimensional array. The following code generates two such arrays:

```
(setq a (make-array '(7 4) :element-type '(unsigned-byte 4))
      b (make-array 4 :displaced-to a
                      :dispaced-index-offset 10.
                      :element-type '(unsigned-byte 4)))
```

Software defines the precise algorithm to be used when accessing an indirect array.

1.2.7 I-Machine Array Registers

An array register is four words on the stack that contain a decoded form of an array, permitting faster access because no reference to the prefix is required. I-machine array registers are essentially the same as those on the L-machine, with the addition of an index-offset feature to allow non-word-aligned array registers with reasonable speed (on the L-machine they are very slow).

The four array-register words on the stack are, in order:

Array Object reference

Control word a fixnum containing the following packed fields:

Position	*Bits*	*Symbolic Name*	*Description*
<31:30>	2	%%ELEMENT-TYPE	One of: fixnum, character, boolean, or object-reference
<29:27>	3	%%BYTE-PACKING	Base 2 logarithm (0 to 5) of the number of elements per word
<26:22>	5	%%BYTE-OFFSET	Offset from word boundary in units of array elements
<21:0>	22	%%EVENT-COUNT	Used for validity checking

Base address The address of the first element in the array

Array length The number of elements in the array

The %%EVENT-COUNT field is a copy of the internal processor register array-event-count. This copy is set when the array register is created, and updated by Lisp code whenever an exception is taken because the %%EVENT-COUNT field

The way you tell a displaced/indirect array from a normal array is by checking the array-displaced bit of the array header (assuming the array has its long prefix bit set). Indirect arrays can be can detected by the data type tag of the fourth word. Figure 15 shows a simple displaced array, while the figure in Figure 16 shows a one-dimensional array indirected to another two-dimensional array. The following code generates two such arrays:

```
(setq a (make-array '(7 4) :element-type '(unsigned-byte 4))
      b (make-array 4 :displaced-to a
                      :dispaced-index-offset 10.
                      :element-type '(unsigned-byte 4)))
```

Software defines the precise algorithm to be used when accessing an indirect array.

Conformal arrays are detected ty testing ARRAY-DISCONTIGUOUS-BIT. Software may be able to do certain optimizations with this knowledge. ARRAY-DISCONTIGUOUS-BIT and ARRAY-DISPLACED-BIT are not used by hardware.

1.2.7 I-Machine Array Registers

An array register is four words on the stack that contain a decoded form of an array, permitting faster access because no reference to the prefix is required. I-machine array registers are essentially the same as those on the L-machine, with the addition of an index-offset feature to allow non-word-aligned array registers with reasonable speed (on the L-machine they are very slow).

The four array-register words on the stack are, in order:

Array Object reference

Control word a fixnum containing the following packed fields:

Position	*Bits*	*Symbolic Name*	*Description*
<31:30>	2	%%ELEMENT-TYPE	One of: fixnum, character, boolean, or object-reference
<29:27>	3	%%BYTE-PACKING	Base 2 logarithm (0 to 5) of the number of elements per word
<26:22>	5	%%BYTE-OFFSET	Offset from word boundary in units of array elements
<21:0>	22	%%EVENT-COUNT	Used for validity checking

Base address The address of the first element in the array

Array length The number of elements in the array

Displaced Array with Prefix
Type %array-prefix-long

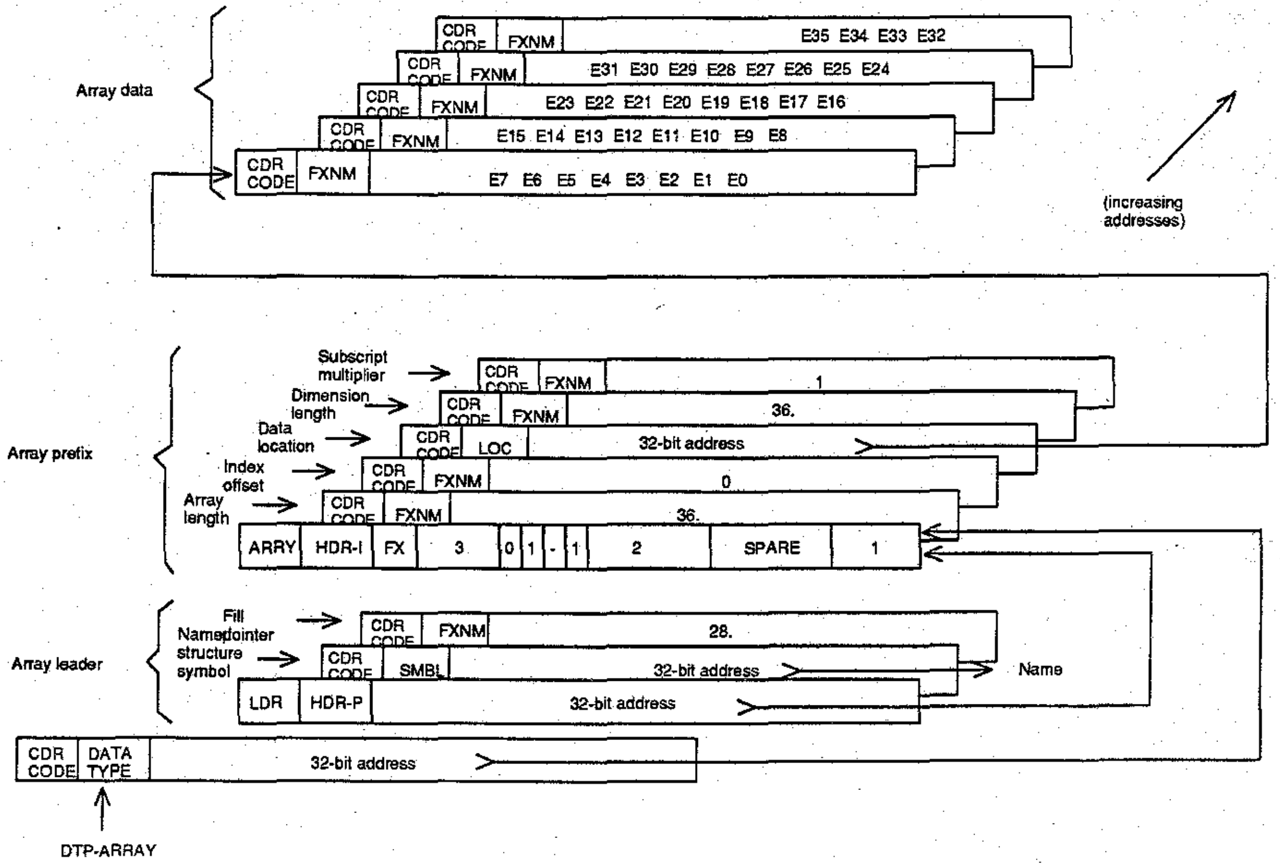


Figure 15. A simple displaced array.

Indirected Array

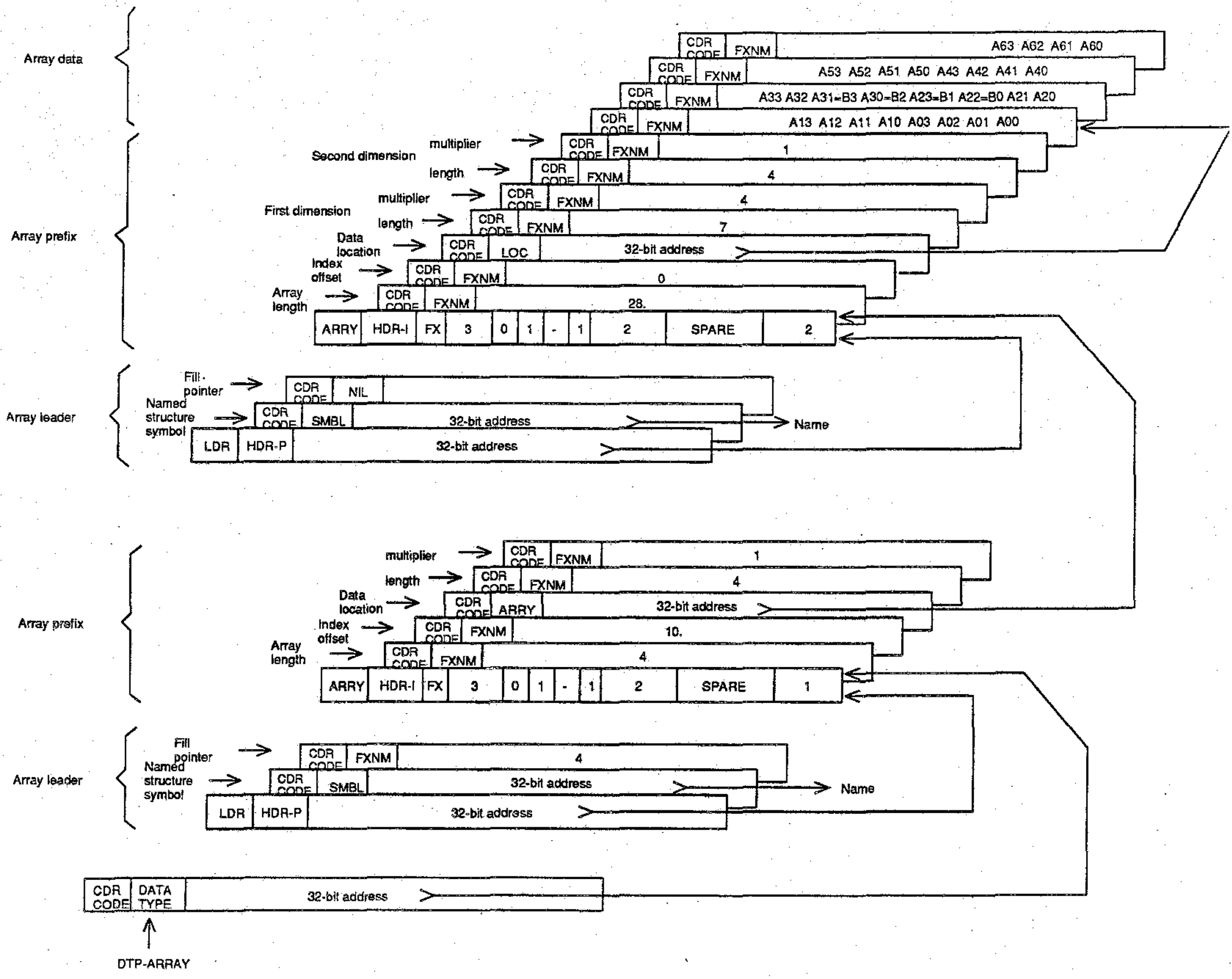


Figure 16. A one-dimensional array indirected to a two-dimensional array.

does not match the array-event-count register. The array-event-count register is incremented by Lisp code whenever the size of an array is changed, invalidating all array registers that have been created. The array-event-count register is by convention always nonzero, forcing the Lisp code to do an extra increment if the new contents would be zero. This convention permits the creation of array registers that always trap (by giving them a zero event count), which may be used for encaching objects of type **dtp-array-instance** and **dtp-string-instance** that do not have encacheable arrays.

To read an element of an array encached in a array register:

1. If the event count is not equal to the contents of the internal processor register array-register-event-count, take an instruction exception and re-decode the array into the array register. This exception need not be handled in hardware/firmware since it will not happen often. It is a post trap, which is responsible for either backing up the PC or for doing the read itself.

2. Compare the subscript against the array length, take an instruction exception unless

   ```
   (%unsigned-lessp subscript length)
   ```

   is true.

3. Add %%byte-offset to the subscript.

4. Read the memory word at

   ```
   (+ base-address (lsh subscript (- %%byte-packing)))
   ```

5. Use the low-order bits of the subscript, %%byte-packing, and %%element-type to extract the array element from the word read from memory. Take an instruction exception if the %%element-type requires a data type different from what was read.

Much of the above happens in parallel, as it does on the L-machine. The comparison against the array length actually happens after the address is sent to memory, but if the subscript is out of bounds the memory read is cancelled and no page fault occurs. Large integers (**dtp-bignum**) are not truncated when stored into an art-nb array; rather, an instruction exception is taken which signals an error. Setting a character with nonzero high bits into an art-string also causes an instruction exception.

Table 2 lists the valid array types for each array element type for all possible values of array byte packing.

Table 2. Valid Array Types for Byte-Packing Values

	fixnum	*character*	*boolean*	*object*
array-byte-packing				
0	art-fixnum	art-fat-string	xxx	art-q
1	art-16b	16-bit-string	xxx	xxx
2	art-8b	art-string	xxx	xxx
3	art-4b	xxx	xxx	xxx
4	art-2b	xxx	xxx	xxx
5	art-1b	xxx	art-boolean	xxx

1.2.8 Representations of Functions and Closures

1.2.8.1 Representation of Compiled Functions

The object reference to a compiled function is a word of data type **dtp-compiled-function**, whose address field points to a word inside a compiled-function structure. The compiled-function structure consists of three parts: the prefix, the body, and the suffix. The prefix is two words long and has a fixed format. The body is a sequence of one or more instructions. The suffix is at least one word long and contains debugging information and constant data. The object reference to a compiled function contains the address of the first word in the body, which is usually the first instruction executed when the function is called. The prefix extends to lower addresses. The suffix is at higher addresses than the body. The hardware, however, knows nothing about the format of the prefix or suffix.

I-Machine compiled functions differ from those of the 3600 by not having a constants/external references table, since references to constants and to external value and function cells are stored in-line in the body. In addition, the "args-info" of an I-Machine compiled function is not stored explicitly, since it can easily be reconstructed from the entry instruction by software.

The first word in the prefix is a header word that identifies this object as a compiled function and specifies its size and the sizes of its parts. The bits in this word are:

<39:38>	%HEADER-TYPE-COMPILED-FUNCTION
<37:32>	DTP-HEADER-I
<31:18>	Size of the suffix (14 bits)
<17:0>	Total size of the object (18 bits)

The second word in the prefix is available for use as the function cell that contains the current definition of the function. Typically the function cell of the symbol that names a function contains a **dtp-one-q-forward** invisible pointer with the address of the function cell of the compiled function, which contains a **dtp-compiled-function** reference to the beginning of its own body. This is the same as on the 3600. If the function is redefined, then the function cell will point someplace else and execution will be slower. If **dtp-call-compiled-even/odd** is used, inter-function references bypass the function cell. This is discussed in detail in the chapter on function calling. See the section "Function Entry."

The even half of the first word in the body is the first instruction of the function, known as the entry instruction. This is the point at which execution usually begins. The entry instruction occupies both halves of the first word. The entry instruction checks the number of arguments. This is discussed in detail in the chapter on function calling. See the section "Function Entry."

The first word in the suffix contains an object reference to a list containing information not needed while executing the function. This information is used mainly by the debugger (also by the compiler and the interpreter). The car of this list is the name of the function and the cdr of the list is an a-list containing information such as names and stack locations of local variables. The cdr code of the first word in the suffix is **cdr-nil** (encoded as 1), which is the illegal instruction sequencing code. This word, with this cdr code, serves as a "fence" that prevents instruction fetchahead from running past the end of the body of a function.

If the body contains any full-word function-calling instructions, the suffix contains linkage information beginning at its second word. The linkage information is a sequence of fixnums joined together by cdr-next codes and terminated by a cdr-nil code. There is a 4-bit byte for each full-word function-calling instruction in the body, which contains the number of arguments to that call (0 to 13), or 14 if the number of arguments is larger than 13, in which case the next two 4-bit bytes contain the number of arguments, or 15 if the compiler does not know the number of arguments or does not want the linker to bypass the entry instruction of the called function. If the linkage information terminates with **cdr-nil** before all of the full-word function-calling instructions have been accounted for, the missing 4-bit bytes are assumed to contain 15.

Succeeding words of the suffix contain the stored representations of list-type constants used by the function (including double-floats, ratios, and complex numbers). Putting these constants in the suffix of the function that uses them minimizes paging. Structure-type constants are typically stored immediately after the function that uses them, again to minimize paging.

See Figure 17

Another section in this chapter discusses the data types of the instructions. (See the section "Instruction Representation.") Refer to the chapter on the instruction set for a discussion of instruction sequencing. See the section "Instruction Sequencing."

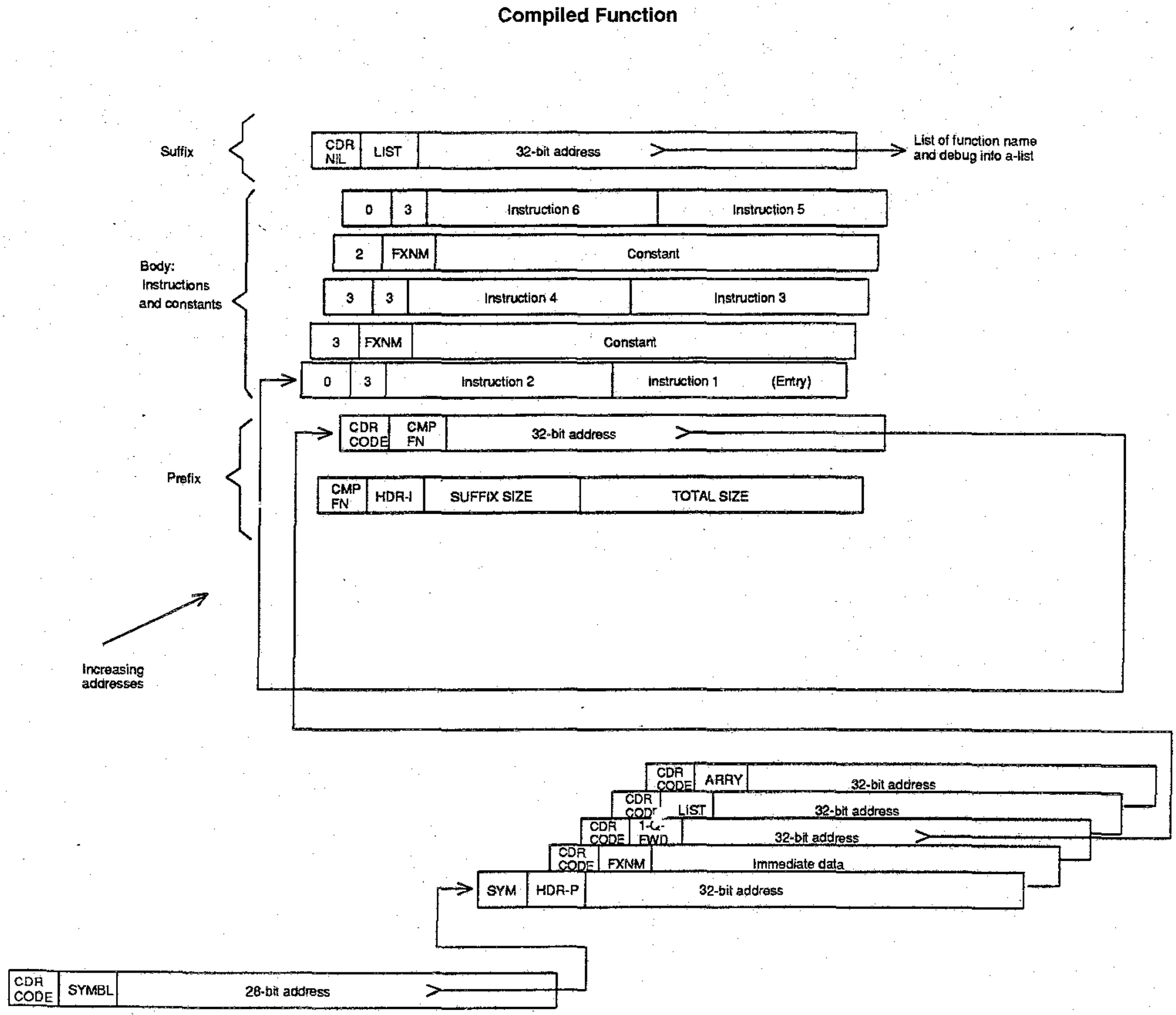


Figure 17. The structure of a compiled function.

1.2.8.2 Generic Functions

An object reference to a generic function has data type **dtp-generic-function**. The address field points to a list-like structure whose content is not architecturally defined; it is used internally by the flavor system. See the section "Generic Functions and Message Passing."

1.2.8.3 Representation of Lexical Closures

The object reference to a lexical closure is a word of data type **dtp-lexical-closure**, which points to a cons pair. The car of the cons is the lexical environment, and the cdr is the function.

The lexical environment, in a typical software implementation, is a cdr-coded list of value cells associated with the closure. In such an implementation, this list must be compact, that is, cdr-coded using **cdr-next**, since instructions that access the lexical variables compute addresses of the variables simply as an offset past the address of the environment. See Figure 18.

When a lexical closure is called as a function, the environment will be made an argument to the function. For more information, refer to the chapter on function calling. See the section "Starting a Function Call."

1.2.8.4 Representation of Dynamic Closures

The object reference to a dynamic closure is a word of data type **dtp-dynamic-closure**, which points to a list structure. The format of a dynamic closure is not architecturally defined, but is determined by software. (The hardware traps to Lisp to funcall dynamic closures.)

The list representation allows closures to be stored in the stack (a la **with-stack-list**); certain special forms such as **error-restart** exploit this.

The list is always cdr-coded, but nothing actually depends on this. The first element of the list is the function. Succeeding elements are taken in pairs. The first element of each pair is a locative pointer to the value cell to be bound when the closure is called. The second element of each pair is a locative pointer to the closure value cell to which that cell is to be linked. See Figure 19.

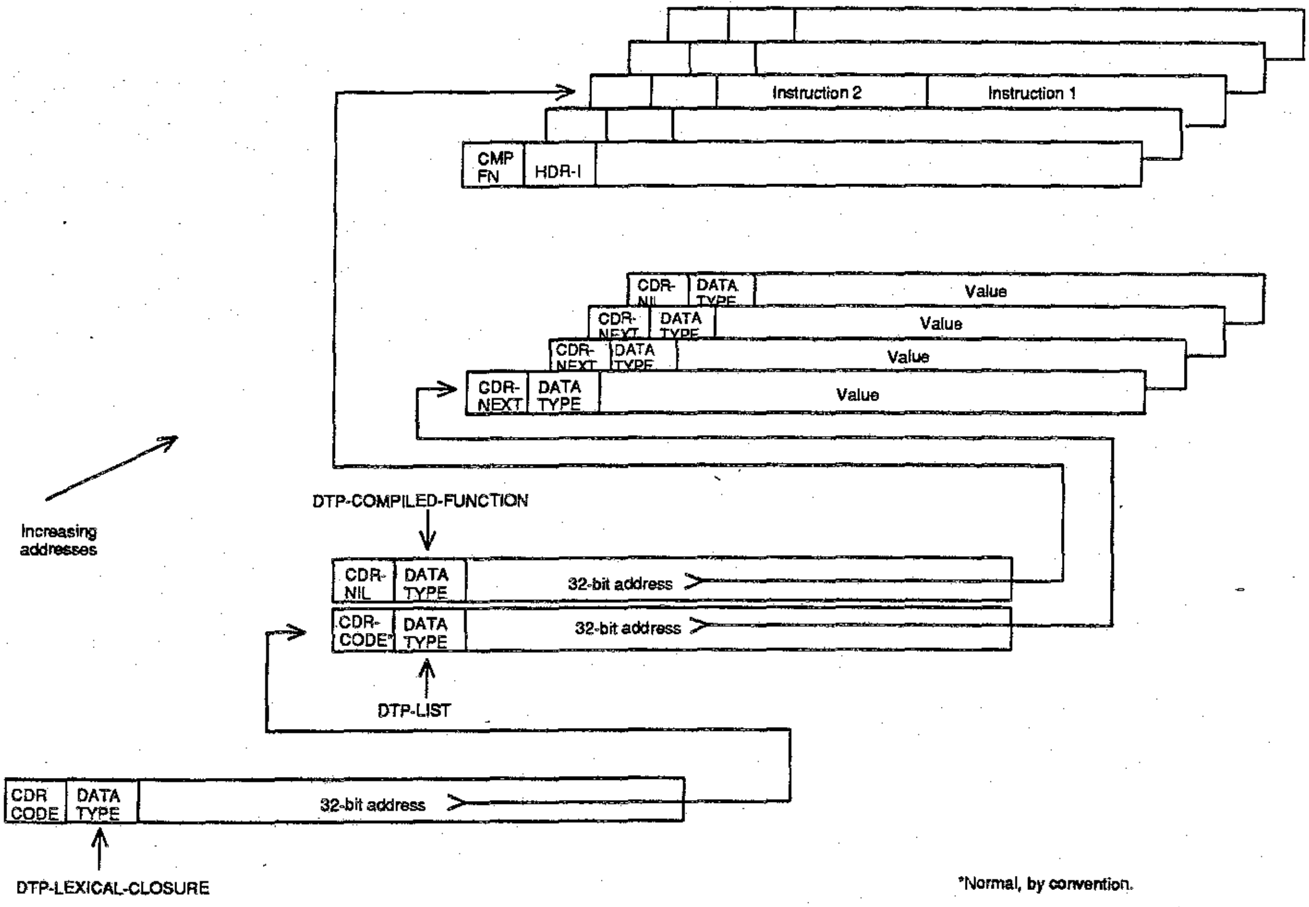
1.2.9 Instruction Representation

The instructions in a compiled function are a sequence of words whose data-type field selects among three types of words:

- *Packed instructions* -- data types with type codes 60-77 are used for words that contain two 18-bit instructions. These are the usual stack-machine type instructions, similar to those of the 3600.

- *Full-word instructions* -- data types coded 50 through 57 are used for words

Lexical Closure



*Normal, by convention.

Figure 18. The structure of a lexical closure.

Dynamic Closure

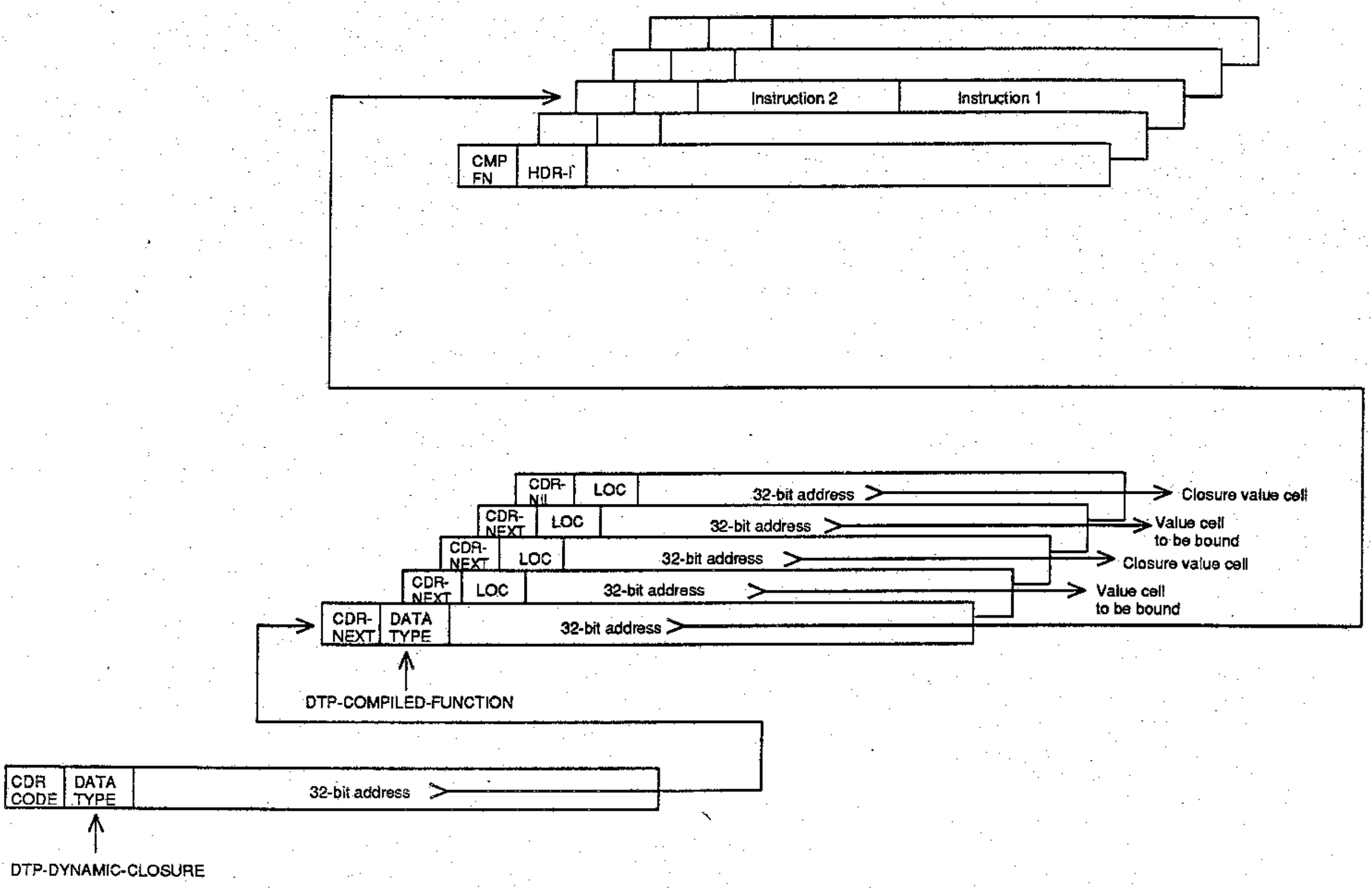


Figure 19. The structure of a dynamic closure.

that contain a single instruction, with an address field. These are used for starting function calls. In addition, data type **dtp-external-value-cell-pointer** (type code 4) is used to fetch the contents of the value cell of a special variable or the function cell of a function and push it on the stack. This is actually an optimization to save space and time (one-half word and one cycle); the value cell address could be pushed as a constant locative and then a **car** instruction could be executed. Besides these, there is one other full-word instruction type, the entry instructions, which do not contain addresses, but instead look like pairs of half-word instructions. These are decoded by their opcode field, not by the data-type field.

- *Constants* -- all other data types encountered among the instructions in a compiled function are constants. The word from the instruction stream is pushed on the stack with the cdr code set to **cdr-next**. The hardware will signal an error if the word is a header or an invisible pointer.

The fields within the various types of instructions are described in the chapter on the instruction set. See the section "Macroinstruction Set."

1.2.10 Program-Counter Representations

The program counter (pc) is a register in the I machine that contains the virtual address of the currently executing instruction. Since most instructions are packed two-to-a-word, that address has to include information about which half-word instruction is executing. This information is included in the data-type code of the pc contents; thus there are two pc data types, **dtp-even-pc** and **dtp-odd-pc**. Words of these data types are not usually found in the stored representations of Lisp objects, but occur within stack frames or inside compiled functions for long branches. See the section "Function Calling, Message Passing, Stack Group Switching."

1.2.11 Representation of Locatives

A locative is a pointer to virtual memory implemented as an object with data type **dtp-locative** and an address field that is the address of the virtual memory word to which it points. It is classified as a *pointer* object reference (See the section "Object References."). Locatives may point to locations *within* objects, such as the value cell of a symbol. Other uses include the pointer to the start of data in long format arrays and the base address of array registers.

1.2.12 Representation of Physical Addresses

The data type **dtp-physical-address** allows unmapped access to the full (up to 32 bits wide) physical address space. Since it is a separate data type it has restricted usage. It cannot, for example, be used as a program counter, nor can it be used as the argument to **car** (as **dtp-locative** can) to get a datum from an arbitrary memory location.

dtp-physical-address is used

- By instructions that do not check the type of their argument. There are two categories of these:
  - Instructions that reference memory, including **%p-ldb**, **%memory-read**, **%p-store-whole-contents**, and their related instructions.
  - Instructions that do not reference memory, including **%pointer-plus**, **%pointer-increment**, and **%pointer-difference**. Note that **%pointer-difference** between a **dtp-physical-address** and a non-**dtp-physical-address** is not meaningful.
- As the indirect pointer to an array or as the base address of an array register. The hardware will never directly see an indirect pointer to an array because indirect pointers imply long prefix arrays, which the hardware does not directly support. Such arrays can be encached in array registers and it is here that a fast-aref/aset-1 instruction will encounter a **dtp-physical-address**.
- In block address registers (BARs). This allows optimized retrieval, copy and/or storing of data into I/O devices. BARs may be used in the implementation of copying fixnum arrays. Therefore, the usage of **dtp-physical-address**, as opposed to non-**dtp-physical-address** types, in BARs may be invisible to the high level application, **copy-array-portion** or **bitblt**. Reading a BAR that was loaded with a **dtp-physical-address** will return a **dtp-physical-address**.

A **dtp-physical-address** typically points to "memory" that does not store all forty bits of a word and therefore cannot be used for paging. I/O devices (disk and network controllers), displays (B&W and color), array processors, floating point processors, and the like often implement buffer memory and device registers that have this characteristic. They typically ignore the tag field when written and return data with a tag of **dtp-fixnum** or **dtp-single-float**. A single I/O register may be referenced with **%p-ldb** of a **dtp-physical-address**. A group of I/O registers may be implemented as a **art-fixnum** array that is indirected, with **dtp-physical-address** to the first I/O register. In this case a reference to one register would be with **aref**. Similarly, buffer memory would be implemented as an array, though not necessarily of type **art-fixnum**, depending on the semantics of the buffer memory.

dtp-physical-address always points to physical memory, not virtual memory, and is therefore an immediate data type. It does not replace the need for the high part of virtual space mapping to a fixed portion of the physical space, known as vma=pma virtual pointers. vma=pma is still needed for certain structures such as the paging system, which requires the PC to have a vma=pma pointer field.

1.3 Data-Type Code Assignments

This section summarizes all of the different data types defined by the architecture. The data type of a word is stored in its tag field.

It is important to note that not all data types are necessarily understood completely by a particular implementation. For example, the hardware understands that **dtp-complex** is a number, but it may not be capable of performing arithmetic operations on complex numbers.

The following tables enumerate all sixty-four data types, along with a brief description of each. Note that the sixty-four types are grouped into several common classes.

1.3.1 Headers, Special Markers, and Forwarding Pointers

Eight data types, as shown in Table 3:
Table 3. Headers, Special Markers, and Forwarding Pointers

Type Code	*Symbolic Name*	*Description*
0	DTP-NULL	Unbound variable/function, uninitialized storage
1	DTP-MONITOR-FORWARD	This cell being monitored
2	DTP-HEADER-P	Structure header, with pointer field
3	DTP-HEADER-I	Structure header, with immediate bits
4	DTP-EXTERNAL-VALUE-CELL-POINTER	Invisible except for binding
5	DTP-ONE-Q-FORWARD	Invisible pointer (forwards 1 cell)
6	DTP-HEADER-FORWARD	Invisible pointer (forwards whole structure)
7	DTP-ELEMENT-FORWARD	Invisible pointer in element of structure

1.3.2 Number Data Types

Eight types as shown in Table 4:

Table 4. Number Data Types

Type Code	*Symbolic Name*	*Description*
10	DTP-FIXNUM	Small integer
11	DTP-SMALL-RATIO	Ratio with small numerator and denominator
12	DTP-SINGLE-FLOAT	Single-precision floating point
13	DTP-DOUBLE-FLOAT	Double-precision floating point
14	DTP-BIGNUM	Big integer
15	DTP-BIG-RATIO	Ratio with big numerator or denominator
16	DTP-COMPLEX	Complex number
17	DTP-SPARE-NUMBER	A number to the hardware trap mechanism

1.3.3 Instance Data Types

Four types as shown in Table 5:

Table 5. Instance Data Types

Type Code	*Symbolic Name*	*Description*
20	DTP-INSTANCE	Ordinary instance
21	DTP-LIST-INSTANCE	Instance that masquerades as a cons
22	DTP-ARRAY-INSTANCE	Instance that masquerades as an array
23	DTP-STRING-INSTANCE	Instance that masquerades as a string

1.3.4 Primitive Data Types

Eleven types as shown in Table 6:

Table 6. Primitive Data Types

Type Code	*Symbolic Name*	*Description*
24	DTP-NIL	The symbol NIL
25	DTP-LIST	A cons
26	DTP-ARRAY	An array that is not a string
27	DTP-STRING	A string
30	DTP-SYMBOL	A symbol other than NIL
31	DTP-LOCATIVE	Locative pointer
32	DTP-LEXICAL-CLOSURE	Lexical closure of a function
33	DTP-DYNAMIC-CLOSURE	Dynamic closure of a function
34	DTP-COMPILED-FUNCTION	Compiled code
35	DTP-GENERIC-FUNCTION	Generic function (see later section)
36	DTP-SPARE-POINTER-1	Spare pointer
37	DTP-SPARE-POINTER-2	Spare pointer
40	DTP-PHYSICAL-ADDRESS	Physical address
41	DTP-SPARE-IMMEDIATE-1	Spare immediate
42	DTP-SPARE-POINTER-3	Spare pointer
43	DTP-CHARACTER	Common Lisp character object
44	DTP-SPARE-POINTER-4	Spare pointer

Note that codes 36, 37, 42, and 44 are spare pointer data types and code 41 is a spare immediate data type. Object references with these data types can be used perfectly normally, but there are no built-in hardware operations that do anything with them.

1.3.5 Special Marker for Garbage Collector

One type as shown in Table 7:
Table 7. Special Marker for Garbage Collector

Type Code	*Symbolic Name*	*Description*
45	DTP-GC-FORWARD	Object-moved flag for garbage collector

1.3.6 Data Types for Program Counter Values

Two types as shown in Table 8:

Table 8. Data Types for Program Counter Values

Type Code	*Symbolic Name*	*Description*
46	DTP-EVEN-PC	PC at first packed instruction in word, or of full-word instruction
47	DTP-ODD-PC	PC at second instruction in word

1.3.7 Full-Word Instruction Data Types

Eight types as shown in Table 9:

Table 9. Full-Word Instruction Data Types

Type Code	*Symbolic Name*	*Description*
50	DTP-CALL-COMPILED-EVEN	Start call, address is compiled-function
51	DTP-CALL-COMPILED-ODD	Start call, address is compiled-function
52	DTP-CALL-INDIRECT	Start call, address is function cell
53	DTP-CALL-GENERIC	Start call, address is generic-function
54	DTP-CALL-COMPILED-EVEN-PREFETCH	Same as DTP-CALL-COMPILED-EVEN but prefetch is desirable
55	DTP-CALL-COMPILED-ODD-PREFETCH	Same as DTP-CALL-COMPILED-ODD but prefetch is desirable
56	DTP-CALL-INDIRECT-PREFETCH	Same as DTP-CALL-INDIRECT but prefetch is desirable
57	DTP-CALL-GENERIC-PREFETCH	Same as DTP-CALL-GENERIC but prefetch is desirable

1.3.8 Half-Word Instruction Data Types

Sixteen types as shown in Table 10:

Table 10. Half-Word Instruction Data Types

Type Code	*Symbolic Name*	*Description*
60-77	DTP-PACKED-INSTRUCTION	Used for instructions in compiled code.

Each word of this type contains two 18-bit instructions, which is why sixteen data types are used up. Bits <37-36> contain 3 to select the instruction data type. Bits <39-38>, the cdr code, contain sequencing information described in the chapter on the instruction set. The instruction in bits <17-0> is executed before the instruction in bits <35-18>. See the section "Instruction Sequencing."

1.4 Appendix: Comparison of 3600-Family and I-Machine Data Representations

The I machine and 3600-family machine data representations are similar in the following ways:

1. They both use a two-bit cdr-code field.

2. They both have sixty-four data types and use a six-bit data-type field, except as noted below.

3. They have twenty-two data types in common (that is, data types with the same name), seventeen of which are alike in all respects except for the word size difference. These similar data types are:

DTP-NIL

DTP-LIST

DTP-SYMBOL

DTP-LOCATIVE

DTP-LEXICAL-CLOSURE

DTP-GENERIC-FUNCTION

DTP-CHARACTER

DTP-NULL

DTP-MONITOR-FORWARD

DTP-EXTERNAL-VALUE-CELL-POINTER

DTP-ONE-Q-FORWARD

DTP-HEADER-FORWARD

DTP-ELEMENT-FORWARD

DTP-INSTANCE

DTP-GC-FORWARD

DTP-EVEN-PC

DTP-ODD-PC

4. Two data types are similar, except that 3600-family machines obtain an extra four bits in the immediate data fields at the expense of the data-type field. These types are:

dtp-fixnum -- uses sixteen data types on 3600-family machines, one on I machine

dtp-float (3600-family) <-> **dtp-single-float** (I) -- uses sixteen data types on 3600-family machines, one on I machine. Both the 3600-family and the I machine use IEEE floating-point formats.

5. The two header data types are similar, but they have slightly different values and possible fields. These are

```
DTP-HEADER-I
DTP-HEADER-P
```

6. The structure of bignums on the two machines is essentially the same, though the I machine has an explicit data type for them, while 3600-family machines use **dtp-extended-number** with the bignum subtype.

The differences between the data representations and types of 3600-family computers and I machines are:

1. The I machine uses a wider memory word (40 bits) than 3600-family machines (36 bits).

2. The I machine always uses the full six bits of the data type field; 3600-family machines use four bits of this field to make thirty-two-bit immediates.

3. The encodings of the data types are completely different: the only type that has the same encoding is **dtp-null**.

4. The I machine has the following data types which 3600-family machines do not have (not including **dtp-single-float** and **dtp-dynamic-closure**, which are simply named differently):

```
DTP-SMALL-RATIO
DTP-DOUBLE-FLOAT
DTP-BIGNUM
DTP-BIG-RATIO
DTP-COMPLEX
DTP-SPARE-NUMBER
DTP-LIST-INSTANCE
DTP-ARRAY-INSTANCE
DTP-STRING-INSTANCE
DTP-SPARE-POINTER-<1-4>
DTP-SPARE-IMMEDIATE
DTP-PHYSICAL-ADDRESS
DTP-CALL-COMPILED-EVEN
DTP-CALL-COMPILED-ODD
DTP-CALL-INDIRECT
DTP-CALL-GENERIC
DTP-CALL-COMPILED-EVEN-PREFETCH
DTP-CALL-COMPILED-ODD-PREFETCH
DTP-CALL-INDIRECT-PREFETCH
DTP-CALL-GENERIC-PREFETCH
DTP-PACKED-INSTRUCTION
```

5. 3600-family machines have the following data types which I machines do not have (not including **dtp-float** and **dtp-closure**):

```
DTP-BODY-FORWARD (obsolete)
DTP-EXTENDED-NUMBER
DTP-LOGIC-VARIABLE
DTP-<16-17,73-77> (spares)
```

6. The following kind of objects are structure objects on the 3600-family and list objects on the I machine:

    - Rational numbers ("big-ratios" on the I machine. "small-ratios" are immediate on the I machine.)

    - Double-precision floating-point numbers

    - Complex numbers

7. Array structures are quite different on the two families of computers. This is elaborated on in a later section.

8. The data words in a fat string have **dtp-fixnum** on the I machine; they are **dtp-character** on 3600-family machines.

9. Compiled functions are quite different on the two families of computers. This is elaborated on in a later section.

1.4.1 Array Differences

These are the main differences between 3600-family arrays and I-machine arrays:

- The format of the I-machine prefix header is simpler and contains more explicit information about the array.

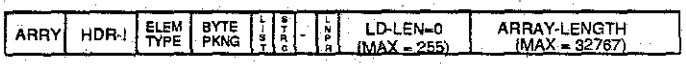
- The optional array leader is stored *before* (at lower memory locations) the array's header on the I machine and *after* it on 3600-family machines. An I machine leader has its own header; a 3600-family leader does not.

- The I machine has two kinds of array prefix, 3600-family machines six. Figure 20 is a detailed comparison of the corresponding array prefix structures, their fields, and the maximum values of the fields.

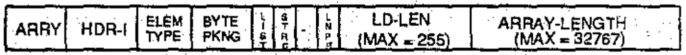
Comparison of Array Prefix Structures

I-Machine

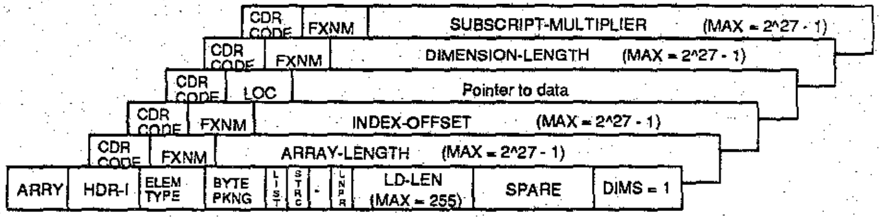
Short-prefix (LNG-PREF=0, LD-LEN=0)



Short-prefix (LNG-PREF=0, LD-LEN not 0)

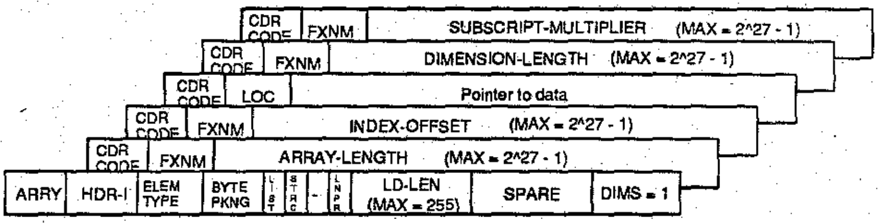


Long-prefix (LNG-PREF=1, LD-LEN = 0)



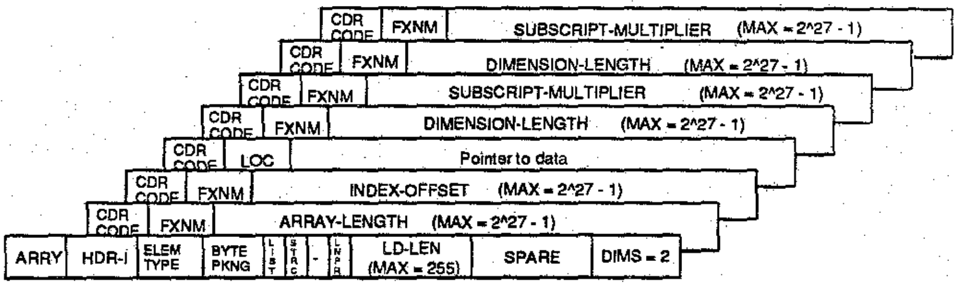
DIMENSION-LENGTH = ARRAY-LENGTH   SUBSCRIPT-MULTIPLIER =1

Long-prefix (LNG-PREF=1, LD-LEN 0 to 255)

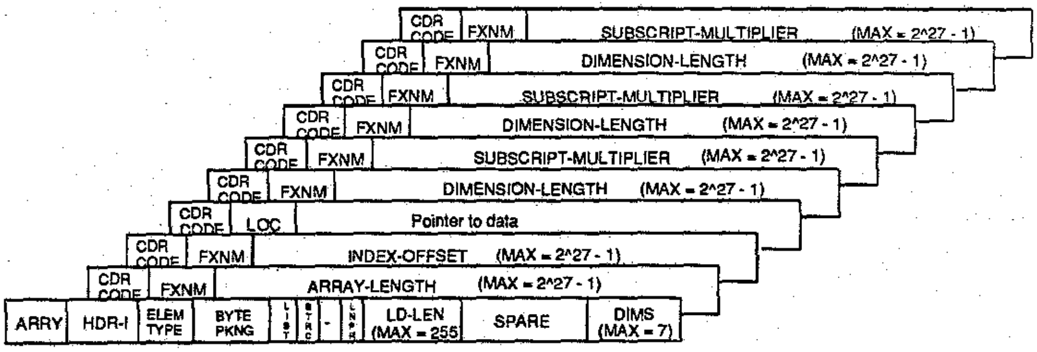


DIMENSION-LENGTH = ARRAY-LENGTH   SUBSCRIPT-MULTIPLIER =1

Long-prefix (LNG-PREF=1, LD-LEN = 0)


CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
LOC
Pointer to data
CDR CODE
FXNM
INDEX-OFFSET (MAX = 2^27 - 1)
CDR CODE
FXNM
ARRAY-LENGTH (MAX = 2^27 - 1)
ARRY
HDR-I
ELEM TYPE
BYTE PKNG
LIST
STRC
-
LNPR
LD-LEN (MAX = 255)
SPARE
DIMS = 2

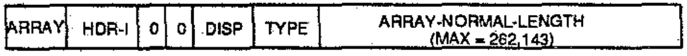
Long-prefix (LNG-PREF=1, LD-LEN = 0 to 255)


CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
LOC
Pointer to data
CDR CODE
FXNM
INDEX-OFFSET (MAX = 2^27 - 1)
CDR CODE
FXNM
ARRAY-LENGTH (MAX = 2^27 - 1)
ARRY
HDR-I
ELEM TYPE
BYTE PKNG
LIST
STRC
-
LNPR
LD-LEN (MAX = 255
SPARE
DIMS (MAX = 7)

Comparison of Array Prefix Structures (continued)

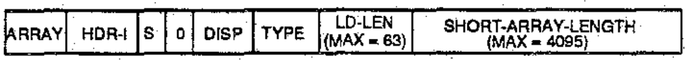
3600-Family

Simple Array (1-dim, no ldr, no ind/disp)



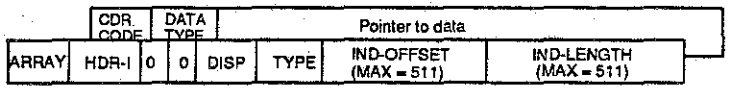
DISP one of {1-BIT, 2-BIT, 4-BIT, 8-BIT, 16-BIT, WORD, FIXNUM, BOOLEAN, CHARACTER, FAT-CHARACTER}

1-Dimension Array with Leader (1-dim, ldr, no ind/disp)



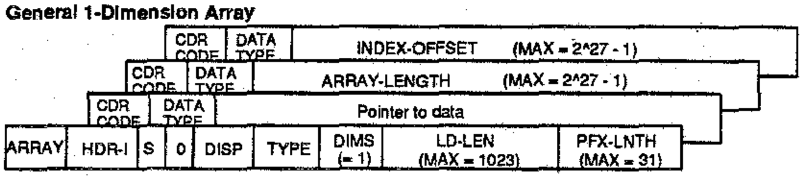
DISP = LEADER

Simple Indirect Array (1-dim, no ldr, ind/disp)



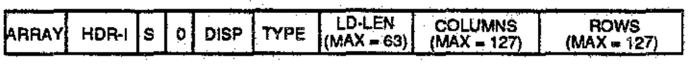
DISP = SHORT-INDIRECT

General 1-Dimension Array



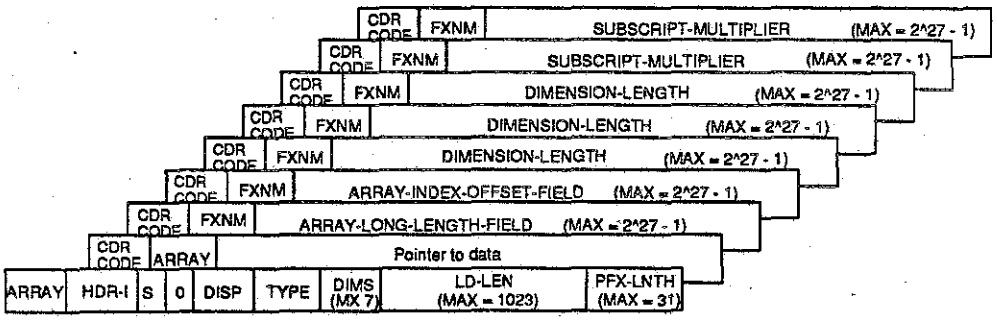
DISP = LONG

Simple 2-Dimension Array (2-dim, no ldr, no ind/disp)



DISP = SHORT-2D

General Multidimension Array


CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
SUBSCRIPT-MULTIPLIER (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
DIMENSION-LENGTH (MAX = 2^27 - 1)
CDR CODE
FXNM
ARRAY-INDEX-OFFSET-FIELD (MAX = 2^27 - 1)
CDR CODE
FXNM
ARRAY-LONG-LENGTH-FIELD (MAX = 2^27 - 1)
CDR CODE
ARRAY
Pointer to data
ARRAY
HDR-I
S
0
DISP
TYPE
DIMS
(MX 7)
LD-LEN
(MAX = 1023)
PFX-LNTH
(MAX = 31)

DISP = LONG-MULTIDIMENSIONAL

1.4.2 Compiled Function Differences

The major difference between the data representations of 3600-family machines and I machines is in the structure of compiled functions.

- 3600-family machines have an *external reference table*, which is stored between the compiled function prefix and the body of instructions. I machines, which omit this table, store the contents of this table -- constants and locatives -- *in line* with the instructions, using the cdr-code field of the packed instruction to indicate sequencing.

- 3600-family machines explicitly store information about the number and type of arguments supplied or required in a field of the compiled function prefix. I machines do not store this information in the prefix: it is supplied in the entry instruction.

- 3600-family machines store in the compiled function's prefix a pointer to debugging information and other information required by the compiler or interpreter. I machines store this pointer in a suffix that follows the body of instructions. They also store linkage information and additional data for the function in this suffix. 3600-family machines have no such suffix.

- Format differences: 3600-family machines have a four-word compiled function prefix; I machines have a two-word prefix and an at-least-one-word suffix. 3600-family machines have seventeen-bit instructions and use the cdr-code field for the high-order bit of each of the two instructions packed in a **dtp-fixnum** word. I machines have eighteen-bit instructions and use the low-order four bits of the data-type field for the high-order bits of the odd instruction.

2. Memory Layout and Addressing

****
This file is confidential. Don't show it to anybody, don't hand it out
to people, don't give it to customers, don't hardcopy and leave it lying
around, don't talk about it on airplanes, don't use it as sales
material, don't give it as background to TSSEs, don't show it off as an
example of our (erodable) technical lead, and don't let our competition,
potential competition, or even friends learn all about it. Yes, this
means you. This notice is to be replaced by the real notice when
someone defines what the real notice is.
****

2.1 Address Space

The architecture provides a single address space which is shared by all processes. An address is thirty-two bits wide, and specifies the location of a word.

The address space is divided into thirty-two zones, each containing 128 megawords. The thirty-two zones are variously assigned to several sections as shown in the table below. Note that ephemeral space is a subset of the virtual address space.

#o00000000000..00777777777	Ephemeral Address Space (zone 0, the low 128 megawords)
#o00000000000..36777777777	Virtual Address Space (zones 0 - 30, the low 3968 megawords)
#o37000000000..37777777777	Unmapped Address Space (zone 31, the high 128 megawords)
#o00000000000..37777777777	Total Address Space (4 gigawords)

2.1.1 Virtual Addresses

The lower 31/32 of the address space is used for virtual addresses. These addresses are subject to page mapping and are used for all allocation of normal objects.

A virtual address is divided into two fields for mapping purposes. These are the virtual page number and the offset within page fields.

Virtual space occupies thirty-one zones. An internal processor register allows each zone to be specified as either old or new space.

Address Fields for Virtual Addresses

Position	*Meaning*
<31:27>	Zone number (zones 0 through 30)
<31:8>	Virtual Page Number (VPN -- 512K virtual pages per zone)
<7:0>	Offset within Page (256 words per page)

The virtual address space is partitioned by software into regions, areas, and quanta. These have no direct hardware impact. Note, however, that the hardware hash function for the Page Hash Table (See the section "Page Hash Table.") is optimized for a quantum size of 65536 words.

2.1.2 Ephemeral Addresses

The lowest zone of the virtual address space is reserved for the storage of ephemeral objects. This space is provided to support a garbage collection strategy that takes advantage of recently created objects usually having a short lifetime.

Ephemeral space is divided into thirty-two levels. Data within an ephemeral level is the same age. The relative ages of different levels is up to software to decide, and would normally change dynamically. Each level is further divided into two halves, old and new space. An internal processor register specifies which half is old and which is new.

The thirty-two ephemeral levels are grouped into four groups of eight levels each. The ephemeral level groups referenced by a page are maintained in the PHT.

Address Fields for Ephemeral Addresses

Position	*Meaning*
<31:27>	00000 => ephemeral, otherwise non-ephemeral
<26>	which half of the ephemeral level
<25:21>	ephemeral level number
<25:24>	ephemeral level group number
<20:0>	word address within an ephemeral level

Static and dynamic data are stored at virtual address 1_27 (2^27) and above. See the section "Revision 0 Implementation Memory Features."

2.1.3 Unmapped Addresses

The upper one-thirty-second of the virtual address space is used to directly address the low portion of the physical address space. The upper five bits of these addresses are translated from all ones to all zeros. They are used primarily to access page tables and paging software, to avoid recursive translation faults. These addresses are sometimes called the virtual=physical or vma=pma region.

Note that there is an aliasing situation for some mapped pages. They have two addresses, one virtual and one vma=pma. A virtual data cache would have to be careful to maintain coherence when writing via one of these addresses and reading via another. A VMA need not translate to a page also accessible by VMA=PMA. (VMA=PMA cannot reference the entire physical address space.)

2.1.4 Wired Addresses

A portion of the system needs to be wired down, that is, not subject to eviction of its pages from main memory. Most obviously, the software that handles page faults needs to be wired.

There are a number of architecturally defined data structures that reside at fixed physical locations. A system implementation must provide memory that responds to these addresses. These locations are as follows (all addresses relative to the beginning of vma=pma space):

000000000..000777777	FEP code, data, and stacks (256K)
001000000..001007777	Trap vectors (refer to chapter 5)
001010000..001010377	FEP communication area
001010400..001010777	System communication area
001011000..001011004	NIL
001011010..001011014	T
777400000..777577777	Boot prom (64K)
777600000..777777777	Reserved for Ibus configuration space (64K)

Init sets the contents of the Program Counter (PC) to VMA=PMA 777400100 (that is, 37777400100 or -377700) with data type **dtp-even-pc**. See the section "Revision 0 Implementation Memory Features."

2.1.5 Pages

The virtual address space is demand-paged with 256-word pages, just as on the 3600.

2.2 GC Support

Two internal processor registers designate sections of the address space as oldspace. These registers can be written via the **%write-internal-register** instruction, allowing the designations to change during execution.

addresses are translated from all ones to all zeros. They are used primarily to access page tables and paging software, to avoid recursive translation faults. These addresses are sometimes called the virtual=physical or vma=pma region.

Note that there is an aliasing situation for some mapped pages. They have two addresses, one virtual and one vma=pma. A virtual data cache would have to be careful to maintain coherence when writing via one of these addresses and reading via another. A VMA need not translate to a page also accessible by VMA=PMA. (VMA=PMA cannot reference the entire physical address space.)

2.1.4 Wired Addresses

A portion of the system needs to be wired down, that is, not subject to eviction of its pages from main memory. Most obviously, the software that handles page faults needs to be wired.

There are a number of architecturally defined data structures that reside at fixed physical locations. A system implementation must provide memory that responds to these addresses. These locations are as follows (all addresses relative to the beginning of vma=pma space):

000000000..000777777	FEP code, data, and stacks (256K)
001000000..001007777	Trap vectors (refer to chapter 5)
001010000..001010377	FEP communication area
001010400..001010777	System communication area
001011000..001011004	NIL
001011005..001011011	T
777400000..777577777	Boot prom (64K)
777600000..777777777	Reserved for Ibus configuration space (64K)

Init sets the contents of the Program Counter (PC) to 777400100. See the section "Revision 0 Implementation Memory Features."

2.1.5 Pages

The virtual address space is demand-paged with 256-word pages, just as on the 3600.

2.2 GC Support

Two internal processor registers designate sections of the address space as oldspace. These registers can be written via the **%write-internal-register** instruction, allowing the designations to change during execution.

The *zone-oldspace register* contains a bit map that specifies for each zone of dynamic space (virtual space minus ephemeral space) whether the zone is newspace or oldspace. A set bit indicates its corresponding zone is oldspace. Bit 0, specifying zone 0, is ignored since that zone is ephemeral space. Bit 31 specifies zone 31, which is vma=pma space. Since vma=pma space cannot be condemned, bit 31 must always be 0 (the hardware may or may not ignore it). The *ephemeral-oldspace register* contains a bit map that specifies for each ephemeral level which half of the level is newspace and which half is oldspace. A set bit indicates the upper half is oldspace.

This scheme never incurs false traps during ephemeral garbage collection, and incurs no false traps during dynamic garbage collection in the usual case where the software allocates addresses according to a certain convention. A false trap is a transport trap for reading a pointer to a zone marked as oldspace in the zone-oldspace register in which the pointer is not actually pointing at a region in oldspace, so the trap handler must recover using the pht.transport-trap bit. This only happens if the software uses a zone in a mixed way, where part of it is oldspace and part is newspace. The first zone of the virtual address space is always used for ephemeral space, while each of the remaining zones can be dedicated to static space, dynamic new/copyspace, or dynamic oldspace. After a garbage collection completes, zones dedicated to dynamic oldspace become free and can be reallocated either to static or to dynamic space, as desired.

2.3 Address Translation

Virtual addresses are mapped before being used to address physical memory. Mapping translates the virtual page number field of the virtual address into a physical page number. Mapping also checks for various exceptions that may result from attempting a memory reference and records information about the reference useful to software.

2.3.1 Page Hash Table

The VPN of a virtual address is translated using the Page Hash Table, or PHT. The PHT is the "backing store" for the hardware map cache: in the event of a map cache miss, the VPN of a virtual address is translated by looking up its entry in the PHT, checking the access attributes, and loading the map cache with the result. Unlike the 3600, the I-machine uses a translation algorithm that is implemented entirely in microcode, so map misses are guaranteed not to cause faults (pclsring) for resident pages.

There are a number of attributes associated with each page. These control access to data in the page, and also record various side effects on the page. These attributes are stored in the PHT along with the translation information. Some of them are also stored in the map cache.

Each entry in the PHT consists of two words, a "key" and a "value" (approximately). Both words' data types are **dtp-fixnum**. The format of an entry is as follows:

Word	*Position*	*Field Name*	*Comments*
PHT0	<39>	spare	
	<38>	end-collision-chain	0 keep searching, 1 stop
	<37:32>	data-type	**dtp-fixnum**
	<31:8>	vpn	-1 for deleted entries
	<7>	fault-request	If 1, this page cannot be accessed in any way
	<6>	pending	For software use only (see the notes section)
	<5:4>	spare	For software use only
	<3:0>	age	Set to 0 when this entry is loaded into the map
PHT1	<39:38>	spare	
	<37:32>	data-type	**dtp-fixnum**
	<31:8>	ppn	(allows 32-bit physical addresses)
	<7>	modified	If 1, this page has been written and probably differs from its on-disk representation
	<6>	write-protect	If 1, this page cannot be written
	<5>	cache-inhibit	If 1, locations in this page are not cached
	<4>	transport-trap	If 1, transport-traps on this page are enabled
	<3:0>	ephemeral-reference	Ephemeral groups referenced by this page

An invalid PHT entry has -1 in its VPN field; since that indicates a VPN=PPN address, it does not usurp any possibly useful page.

The following attributes control access to data in the page. If an instruction attempts an access not allowed by one of these attributes, a fault will be generated. See the section "Translation Algorithm." Note that an implementation should be careful not to cause spurious faults when accessing ahead of instruction execution.

fault-request — fault-request, when 1, indicates that any access to this page should cause a fault. When 0, accesses are allowed according to the write-protect bit.

write-protect

write-protect, when 1, indicates that any attempt to write data into the page should cause a fault. When 0, data can be written into the page. Note: just because a page is write-protected does not mean it is not modified; there are several mechanisms that circumvent this bit. See the modified bit, below.

transport-trap

transport-trap, when 1, enables traps when reading a word from this page that is a potentially a pointer to oldspace. This is used by the garbage collector.

Words are potentially pointers to oldspace if their data-type field contains a pointer type and their address field satisfies a condition based on the address space referenced. See the section "Lisp-Machine Data Types." The condition for a reference to ephemeral space is that the ephemeral-oldspace register indicates the half of the ephemeral level referenced is oldspace. The condition for a reference to dynamic space is that the zone-oldspace register indicates the zone referenced is oldspace. References to physical space never generate transport traps.

If the pointer satisfies the above conditions and the transport-trap bit is set for the page, then a transport trap is taken. The garbage collector is responsible for deciding whether or not the pointer truly points to oldspace.

See the section "Revision 0 Implementation Memory Features."

The following attributes record various side effects that have occurred to data in the page. The hardware maintains these attributes for use by the software.

age<3:0>

The age field is set to 0 when an instruction accesses data in this page, or an instruction is executed from this page.

The paging software interprets this field as either a set of bits, all of which are cleared upon reference, or as a counter which is reset to zero upon reference. Either way, the intent is to assist a pseudo-LRU page replacement algorithm and perhaps allow experimentation with more sophisticated schemes.

Because the age is in the PHT, instead of in the MMPT, as in the 3600, the page replacement algorithm will scan through main memory pages in the order they appear in the PHT rather than in order of increasing physical addresses. Because of this, PHT insertion and deletion may not generally be allowed to relocate PHT entries.

The age is stored only in the PHT. By definition, when an entry is in the map cache, the age is 0.

modified — modified is set to 1 whenever data is written into this page. Paging software clears this bit when it has saved the page.

ephemeral-reference<3:0>

The ephemeral-reference field records which ephemeral level groups are referenced by pointers in this page. Each bit in this field, when set, indicates that a reference to the corresponding ephemeral level group has been stored in this page. A discussion of ephemeral levels and groups occurs in an earlier section. See the section "Address Space."

This information is used by the ephemeral garbage collector to know whether or not it has to scan this page and rescue objects it references, when a portion of ephemeral space is being garbage-collected.

The PHT is a hash table with buckets of four entries of two words each. The number of buckets must be a power of two, and is chosen to yield between 38% and 70% density (PHT density is pages-of-physical-memory/entries-in-pht). Within each bucket, the four entries are simply laid out in order, alternating PHT0 and PHT1 words. The inner loop of the lookup algorithm searches all the PHT0 words in a bucket for a given VPN, using block-mode memory cycles but skipping over the PHT1 words.

The PHT is allocated in vma=pma space at boot time (any time before the first map cache miss). There are two processor registers describing the PHT: PHT-BASE and PHT-MASK. PHT-BASE is set to the physical address of the first word in the PHT, and PHT-MASK is set to (lsh (1- pht-number-of-buckets) 3). See the section "Revision 0 Implementation Memory Features."

2.3.2 PHT Lookup Algorithm

The PHT lookup algorithm is a rehash-on-collision hash lookup. The hash/rehash algorithm generates a sequence of buckets to be probed; each bucket is linearly scanned, at maximum memory bandwidth, for the desired VPN. The lookup terminates successfully when the desired entry is found, or unsuccessfully after scanning a bucket at the end of a collision chain. The lookup is guaranteed to terminate because the rehash algorithm guarantees that every bucket will be probed, and Lisp guarantees that at least one bucket in the PHT will have end-collision-chain=1. [when there are too many collisions in the PHT to satisfy this constraint, the PHT gets rebuilt -- a time-consuming operation that will probably never happen].

The collision-count mechanism is similar to that in the 3600; the PHT insertion and deletion routines maintain a per-bucket count of the number of entries that hashed to a particular bucket, but could not be stored there because of collisions. However, the actual representation of the collision counts (either in a separate table or in some of the spare bits in PHT0) is not used by the hardware. Instead, the software distills the collision count for each bucket into a single bit, pht.end-collision-chain, which is 0 if the collision count is non-zero, otherwise 1. (In SYSDEF, this is called %%pht0-end-collision-chain.) pht.end-collision-chain is only significant for the last entry of a bucket.

The hash function used for the initial probe of the PHT is computed by a bit-shuffle-and-xor hashbox, the exact description of which is given below. This hashbox maps 24-bit virtual page numbers into PHT bucket numbers, which span eleven bits in a minimal (1M main memory) configuration, thirteen bits in a typical (4M main memory) configuration, and twenty-three bits in the maximum configuration (4096M main memory). However, its output is actually left-shifted by three bits to convert it directly into a PHT offset, saving a cycle in the microcode. The field pht-mask is similarly left-shifted.

This hash function was chosen presuming a page size of 2^8 words, a quantum size of 2^16 words, a half-ephemeral-level size of 2^21 words, and a zone size of 2^27 words. All bit numbers are in decimal.

PHT-OFFSET<0..25>

```
HASH< 0> = 0
HASH< 1> = 0
HASH< 2> = 0
HASH< 3> = VMA<12> D VMA<27>
HASH< 4> = VMA<11> D VMA<28>
HASH< 5> = VMA<10> D VMA<29>
HASH< 6> = VMA< 9> D VMA<30>
HASH< 7> = VMA< 8> D VMA<31>
HASH< 8> = VMA<13> D VMA<20>
HASH< 9> = VMA<14> D VMA<22>
HASH<10> = VMA<15> D VMA<21>
HASH<11> = VMA<16> D VMA<26>
HASH<12> = VMA<17> D VMA<25>
HASH<13> = VMA<18> D VMA<24>
HASH<14> = VMA<19> D VMA<23>
HASH<15> = VMA<12> D VMA<16>
HASH<16> = VMA<11> D VMA<17>
HASH<17> = VMA<10> D VMA<18>
HASH<18> = VMA< 9> D VMA<19>
HASH<19> = VMA< 8> D VMA<20>
HASH<20> = VMA<13> D VMA<25>
```

```
HASH<21> = VMA<14> ⊗ VMA<26>
HASH<22> = VMA<15> ⊗ VMA<27>
HASH<23> = VMA<21> ⊗ VMA<31>
HASH<24> = VMA<22> ⊗ VMA<30>
HASH<25> = VMA<23> ⊗ VMA<29>
```

This hashbox is accessible by Lisp via an internal register.

The first bucket probed is computed by the hashbox described above, modulo the table size. If that probe fails, a linear pseudo-random number generator, initialized to 17*vpn+1 and advanced by 17x+1, defines the rehash sequence. A Lisp expression of the lookup algorithm is given below:

```
;; This is just 17x + 1, mod 2^32.
(defmacro pht-next (state)
  '(sys:%32-bit-plus
     (sys:%32-bit-plus
       (sys:%logdpb ,state (byte 28. 4.) 0)
       ,state)
     1))

  (defun pht-lookup (vpn)
    (flet ((search-bucket (pht-offset)
             (loop repeat 4
                   initially (setf (%block-address) (+ pht-base pht-offset))
                   for entry = (%block-read)      ;fetch next pht0 word
                   do (if (= (ldb %%pht0-vpn entry) vpn)
                          (if (= (ldb %%pht0-fault-request entry) 0)
                  ;; This is the correct entry, return pht0 and pht1 words.
                              (return-from pht-lookup entry (%block-read))
                  ;; This is the correct entry, but fault-request is set.
                            (take-page-fault-request-trap))
                  ;; VPN doesn't match, skip over the pht1 word for this entry.
                        (%block-read))
                   finally
                     ;; If at end of collision chain, fail.
                     (when (= (ldb %%pht0-collision-chain entry) 1)
                       (take-page-not-resident-trap)))))
      (search-bucket (logand (pht-hash vpn) pht-mask))
      (loop for state first (pht-next vpn) then (pht-next state)
            do (search-bucket (logand (lsh state 3) pht-mask)))))
```

See the section "Revision 0 Implementation Memory Features." A new entry is inserted into the PHT by hashing/rehashing the VPN into successive bucket numbers and searching each bucket for an invalid entry to reuse. The collision bucket for an invalid entry to reuse. The collision count of each full bucket in the hash sequence is incremented. When incrementing a bucket's collision count from 0 to 1, pht.end-collision-chain for that bucket must be set to 0.

An entry is deleted from the PHT by hashing/rehashing the VPN into successive bucket numbers and searching each bucket for the entry. The collision-count of each bucket in the hash sequence (excepting the one that actually contains the entry) is decremented. If a collision count is decremented below 0, you have tried to delete a nonexistent entry and have corrupted the table by inappropriately decrementing collision counts (the 3600 just crashes in this case). When decrementing a bucket's collision count from 1 to 0, pht.end-collision-chain for that bucket should be set to 1. Deleted entries are marked by setting their VPN field to -1.

2.3.3 Translation Algorithm

When the attributes of a resident virtual page are changed, either by Lisp doing something like aging or replacing the page or by a memory reference causing the age, modified, and/or ephemeral-reference attributes to change, the PHT and the map cache must be synchronized so they both contain the same information. This is a cache/backing-store sort of problem, and the same sort of solutions apply. We use a "write-through" strategy, so the map cache and the PHT are always consistent.

When the storage system wants to change the attributes of a resident page, it updates the PHT entry for the page, and simply invalidates the map cache entry for that page (if one exists). See the section "Internal Registers." The next reference to the page will reload the map.

When a memory reference needs to change the attributes of a page that has an entry in the map cache (modified and ephemeral-reference are the only fields it can change), a microcode trap handler is invoked to update the corresponding fields in the PHT. Whether or not the PHT update occurs before or after the reference is implementation dependent.

The translation/access-checking process for a memory read cycle is:

```
if VMA is of type dtp-physical-address
   access-bits := write-protect=0,
                  transport-trap=0, modified=1,
                  ephemeral-reference=17
   MD := contents of physical address VMA<31:0>
else if VMA is in VMA=PMA space
   access-bits := write-protect=0,
                  transport-trap=0, modified=1,
                  ephemeral-reference=17
   if VMA is shadowed by the stack-cache
      MD := contents of stack-cache address VMA<7:0>
   else
      MD := contents of physical address VMA<26:0>
else if VMA has an entry in the map cache
   PPN, access-bits come from map cache entry
   if VMA is shadowed by the stack-cache
      MD := contents of stack-cache address VMA<7:0>
   else
      MD := contents of physical address PPN|VMA<7:0>
else if PHT contains an entry for VPN with fault-request=0
   if age90, rewrite PHT0 word clearing age
   load map cache with PPN, access-bits from PHT entry
   retry memory cycle
else
   take page-not-resident pre-trap
```

The translation/access-checking process for a memory write cycle is:

```
if VMA is of type dtp-physical-address
   access-bits := write-protect=0,
                  transport-trap=0, modified=1,
                  ephemeral-reference=17
   write MD to physical address VMA<31:0>
else if VMA is in VMA=PMA space
   access-bits := write-protect=0,
                  transport-trap=0, modified=1,
                  ephemeral-reference=17
   if VMA is shadowed by the stack-cache
      write MD to stack cache address VMA<7:0>
   write MD to physical address VMA<26:0>
else if VMA has an entry in the map cache
  PPN, access-bits come from map cache entry
   if write-protect=1, take page-write-fault pre-trap
   if (or (= modified 0)
          (and (pointer-type? MD)
               (ephemeral-address? MD)
               (= (logand (lsh 1 vma-ephemeral-level-group(MD))
                                      ephemeral-reference) 0)))
      trap to microcode to update the pht
   if VMA is shadowed by the stack-cache
      write MD to stack cache address VMA<7:0>
   write MD to physical address PPN|VMA<7:0>
else if PHT contains an entry for VPN with fault-request=0
   if write-protect=1, take page-write-fault pre-trap
   unless age=0, modified=1,
      and the appropr iate ephemeral-reference
      bit is set, rewrite PHT entry with the updated values
   load map cache with PPN, updated access-bits
   retry memory cycle
else
   take page-not-resident pre-trap
```

The fake access-bits for VMA=PMA and dtp-physical-address addresses are chosen to prevent PHT update traps (those addresses are not in the PHT, so you could not update them if you tried). There are two very important consequences of this: vma=pma or dtp-physical-address write cycles do not update the ephemeral-reference bits, and vma=pma or dtp-physical-address read cycles do not take transport traps. Code that uses such addresses when using ephemeral references or references to oldspace must be very careful not to violate the conventions imposed by the garbage collector.

Whether or not a given address is shadowed by the stack cache is determined by examining the virtual address only. Memory operations using dtp-physical-addresses will always bypass the stack cache.

To work properly in a shared-memory multiprocessor, updating a PHT entry should be implemented by reading the entry, ORing in the changed attributes, and writing the entry, using interlocked bus cycles. See the section "Revision 0 Implementation Memory Features." A processor should not presume that its map cache entry is up to date, since other processors may have modified the PHT entry since it was encached. (When software modifies a mapping and adjusts the PHT, software must coordinate the change with all processors, which probably involves the invalidation of previous map-cache entries.)

2.4 Appendix: Comparison of 3600-family and I-machine Memory Layout and Addressing

- 3600-family and I-machine memory layout and addressing are similar in the following ways:

- Both architectures employ a single address space that is shared by all processes.

- The upper portion of either memory space is used for physical address space, that is, for unmapped addressing. On the I machine, the upper one-thirty-second is used; on 3600-family machines, the upper one-sixteenth. On 3600-family machines, physical address space size is 16 Mwords; on I machines, it is 4 Gwords.

- Both architectures employ the same page size: 256 words.

- Both architectures call for a fixed portion of memory that is "wired," that is, not subject to being swapped from main memory out to secondary memory. The architectures have differing requirements for portions of memory that are not subject to address-translation faults.

- Each architecture can designate portions of storage as containing temporary objects, and has hardware support for keeping track of references to those objects.

- Both architectures perform address translation (mapping of virtual addresses to physical addresses) by means of tables that describe pages resident in main memory.

The differences between the memory layouts and addressing schemes of 3600-family and I machines are:

- The I-machine virtual address space is sixteen times bigger.

- On the L machine, the hardware can dynamically designate attributes of portions of storage at the granularity of hardware quanta (16 Kwords). On the I machine, the attributes of portions of storage are designated by a much more rigid scheme. The primary division of storage in the I-machine, for GC purposes, is a zone (128 Mwords), of which there are 32.

- Wired address spaces are different in the two families. On 3600-family machines, wired memory occupies virtual address space from virtual address 0 to %wired-virtual-address-high (contained in a control register), which is mapped to a contiguous set of physical memory addresses starting at %wired-physical-address-low (in another control register). On I machines, wired pages are stored at a predetermined set of physical addresses, starting at address 0.

- Ephemeral spaces are different in the two families. On the I machine, ephemeral space is architecturally defined to be a particular address space -- zone 0 (addresses 0 to 2^27). On 3600-family machines, the gc tag ram allows the ephemerality of each quantum to be specified.

- GC support in general is different in the two families. On a 3600-family machine, ephemeral-reference attributes of a page are stored in a dedicated hardware memory. On the I-machine, these attributes are stored in the PHT.

- The I machine never has to abort (pclsr) an instruction due to a translation for a resident page, while the L machine sometimes has to do so.

- On a 3600-family, the hardware map cache is backed up by a PHTC (page hash table cache), which is referenced by microcode with some hardware assist. If both the map and the PHTC miss for a given address translation, Lisp is called to attempt the translation via the PHT. The I machine has no PHTC, the hardware map cache is backed up directly by the PHT, which is referenced by microcode.

- The 3600-family PHT is optimized for density (about 66%): each entry is one word, and table size is a prime number. The I-machine PHT (about 50% dense) is optimized for simplicity and performance: each entry is two words, and table size is a power of two. As a result of these differing designs, some attributes of resident pages are in the PHT on the I-machine, but in the MMPT on the 3600-family.

- The stack on the L machine is mapped to virtual memory on a per-page basis. In the I machine, the stack cache size (128 words in the first implementation) is less than the size of a page (256 words), so there are registers that indicate the upper and lower bounds of the stack cache. (Actually, any cache size less than twice the page size requires such registers.)

3. Macroinstruction Set

****

This file is confidential. Don't show it to anybody, don't hand it out to people, don't give it to customers, don't hardcopy and leave it lying around, don't talk about it on airplanes, don't use it as sales material, don't give it as background to TSSEs, don't show it off as an example of our (erodable) technical lead, and don't let our competition, potential competition, or even friends learn all about it. Yes, this means you. This notice is to be replaced by the real notice when someone defines what the real notice is.

****

3.1 Introduction

This chapter defines all the instructions executed by the I machine. The instructions are grouped according to their function. The index in the end matter of this manual lists the instructions alphabetically, and an appendix lists them by opcode and by instruction format. Another appendix contains a list of 3600 instructions not implemented by the I-machine and, in some cases, descriptions of how to obtain their results with I-machine instructions.

Before presenting the individual instructions, the chapter includes introductory sections applicable to all instructions: instruction sequencing, internal registers, and explanations of the various fields in the instruction definitions, including instruction formats and control stack addressing modes, argument descriptions, types of instruction exceptions, types of memory references, top-of-stack register effects, and the cdr codes of values returned.

3.1.1 Instruction Sequencing

Instructions are normally executed in the order in which they are stored in memory. Since full-word instructions cannot cross word boundaries, it would occasionally be necessary to insert a no-op instruction in places where a full-word instruction or constant followed a half-word instruction that did not fall on an odd halfword address. This costs address space, I Cache space, and possibly execution time to execute the no-op.

The cdr code field of each word executed contains sequencing information to minimize this waste. The cdr code takes on one of four values, which specify how much the PC is incremented after executing an instruction from this word. Note that the PC contains a half-word address.

Cdr Code	*PC Increment*	*Comment*
0	+1	Normal instruction sequencing
1	*illegal*	Fence; marks end of compiled function
2	-1	On some constants
3	+2 PC even +3 PC odd	Before some constants, on some constants

When a constant follows an odd half-word instruction, the half-word instruction pair has cdr code 0 and the constant has cdr code 3. When a constant follows an even half-word instruction, the constant follows the odd half-word paired with the constant's predecessor. The half-word instruction pair has cdr code 3 and the constant has cdr code 2.

For example, straightline execution of the following sequence of instructions:

Word Address	*Cdr Code*	*Instruction(s)*	*Comment*
100	0	B A	Packed instructions
101	3	C	Constant
102	3	F D	Packed instructions
103	2	E	Constant
104	0	H G	Packed instructions

proceeds as follows:

Current PC	*Instruction Executed*	*Cdr Code*	*PC Increment*
100 even	A	0	+1
100 odd	B	0	+1
101 even	C	3	+2
102 even	D	3	+2
103 even	E	2	-1
102 odd	F	3	+3
104 even	G	0	+1
104 odd	H	0	+1

A cdr-code value of 1 (**cdr-nil**) is used to mark the end of compiled functions. This value is placed in the word after the final instruction of the function. See the section "Representation of Compiled Functions." It is an error if the processor attempts to execute this word. The chapter on traps and handlers contains more information. See the section "Exception Handling."

The cdr code sequencing described above only indicates the default next instruction. When an instruction specifically alters the flow of control (for example, **branch**) the cdr code has no effect.

3.1.2 Internal Registers

Table 11 lists I-machine internal registers. Within this table, an asterisk by an address entry means that the register may be defined by an implementation, and *reserved* means the register may be architecturally defined in the future. The information in this table is specific to Revision 0 of the Ivory chip. As the architecturally defined information in the table becomes determinate, implementation-specific details will be removed to an appendix.

The **%read-internal-register** instruction always returns the object from the specified register with its cdr code set to **cdr-next**. If an internal register has cdr-code bits, they can not be read by this instruction.

The rotate-latch register does not have an internal address and can not be read or written with **%read-internal-register** or **%write-internal-register**.

3.1.3 Memory Side Effects

Reading memory may not cause side effects. The architecture permits an implementation to start a memory read that it will not use, perhaps because of instruction prefetching, perhaps while starting an array reference before an out of bounds check is performed, perhaps because of instruction pipelining (an instruction preceding a memory read takes a trap after the memory read instruction has started its read), or perhaps for something else. Writing memory using a **dtp-physical-address** is allowed to cause side effects; **dtp-physical-address** is guaranteed not to be cached, and the write is guaranteed to happen exactly once. Also, both the **%coprocessor-read** and **%coprocessor-write** instructions may cause side effects; they are guaranteed to be performed exactly once.

3.1.4 Explanation of Instruction Definitions

3.1.4.1 Instruction Formats

In the chapter on data representation, words in Lisp-machine memory were interpreted either as Lisp object references or as parts of the stored representation of these objects. This chapter reinterprets all memory words as *instructions*. The processor treats a memory word as an instruction whenever it is encountered in the body of a compiled function -- or, more specifically, when the program counter points to the memory word and the word is fetched as an instruction.

With the exception of the data types specifically designated as instructions, there is no one-to-one correspondence between data types and instruction formats. Instead, the data types are subdivided into classes, and each class forms the basis

instruction. When an instruction specifically alters the flow of control (for example, **branch**) the cdr code has no effect.

3.1.2 Internal Registers

Table 11 lists I-machine internal registers. Within this table, an asterisk by an address entry means that the register may be defined by an implementation, and *reserved* means the register may be architecturally defined in the future. The information in this table is specific to Revision 0 of the Ivory chip. As the architecturally defined information in the table becomes determinate, implementation-specific details will be removed to an appendix.

The **%read-internal-register** instruction always returns the object from the specified register with its cdr code set to **cdr-next**. If an internal register has cdr-code bits, they can not be read by this instruction.

The rotate-latch register does not have an internal address and can not be read or written with **%read-internal-register** or **%write-internal-register**.

3.1.3 Explanation of Instruction Definitions

3.1.3.1 Instruction Formats

In the chapter on data representation, words in Lisp-machine memory were interpreted either as Lisp object references or as parts of the stored representation of these objects. This chapter reinterprets all memory words as *instructions*. The processor treats a memory word as an instruction whenever it is encountered in the body of a compiled function -- or, more specifically, when the program counter points to the memory word and the word is fetched as an instruction.

With the exception of the data types specifically designated as instructions, there is no one-to-one correspondence between data types and instruction formats. Instead, the data types are subdivided into classes, and each class forms the basis of an instruction type. The packed half-word instruction data type uses two instruction formats. See the section "Half-Word Instruction Data Types."

Table 12 summarizes I-machine instruction formats and lists the data types in each class.

The following paragraphs describe these formats and their interpretations.

Full-Word Instruction Formats

Function-Calling Instruction Formats

A word of data type **dtp-call-xxx** contains a single instruction. The instruction contains a data-type field, which is used as the opcode, and an address field as shown in Figure 21. This kind of instruction starts a function call.

Table 11. I-Machine Internal Registers

Address	*Read*	*Data*	*Register Name*
	/Write	*Type*	
0*			For use by microcode only
1	RW	loc	Frame Pointer (FP)
2	RW	loc	Local Pointer (LP)
3	RW	loc	Stack Pointer (SP)
4*			For use by microcode only
5	RW	loc	Stack Cache Lower Bound
6	RW	loc/pa	BAR0 Contents
206	RW	loc/pa	BAR1 Contents
406	RW	loc/pa	BAR2 Contents
606	RW	loc/pa	BAR3 Contents
7	R	fix	BAR0 Hashed
207	R	fix	BAR1 Hashed
407	R	fix	BAR2 Hashed
607	R	fix	BAR3 Hashed
10*			For use by microcode only
11*			For use by microcode only
12	RW	pc	Continuation
13	RW	fix	DP Op
14	RW	fix	Control Register
15*			For use by microcode only
16	RW	fix	Ephemeral Oldspace Register
17	RW	fix	Zone Oldspace Register
20	R	fix	Implementation Revision
21*	RW	fix	FP coprocessor present
22*			For use by microcode only
23	RW	fix	Preempt Register
24*	RW	fix	Icache Control
25*	RW	fix	Prefetcher Control
26*	RW	fix	Map Cache Control
27*	RW	fix	Memory Control
30*	R	fix	ECC Log
31*	R	fix	ECC Log Address
32*	W	-	Invalidate Matching Map Entry for VMA in BAR0
232*	W	-	Invalidate Matching Map Entry for VMA in BAR1
432*	W	-	Invalidate Matching Map Entry for VMA in BAR2
632*	W	-	Invalidate Matching Map Entry for VMA in BAR3

**Implementation Specific*

Table 11, continued

Address	*Read /Write*	*Data Type*	*Register Name*
33*			For use by microcode only
34	RW	fix	Stack cache overflow limit
35*			For use by microcode only
36*			For use by microcode only
37	*reserved*		
40-47*			For use by microcode only
50*			For use by microcode only
51*			For use by microcode only
52*	W	-	Load Matching Map Word 1 for VMA in BAR0
252*	W	-	Load Matching Map Word 1 for VMA in BAR1
452*	W	-	Load Matching Map Word 1 for VMA in BAR2
652*	W	-	Load Matching Map Word 1 for VMA in BAR3
53-777	*reserved*		
1000	RW	--	Top of Stack (TOS)
1001	RW	--	Array Event Count
1002	RW	--	Binding Stack Pointer
1003	RW	--	Catch Block Pointer
1004	RW	--	Control Stack Limit
1005	RW	--	Control Stack Extra Limit
1006	RW	--	Binding Stack Limit
1007	RW	--	PHT Base
1010	RW	--	PHT Mask
1011	RW	--	Count Map Reloads
1012	RW	--	List Cache Area
1013	RW	--	List Cache Address
1014	RW	--	List Cache Length
1015	RW	--	Structure Cache Area
1016	RW	--	Structure Cache Address
1017	RW	--	Structure Cache Length
1030	RW	--	Maximum Frame Size
1031	RW	--	Stack Cache Dump Quantum

**Implementation Specific*

Table 12. I-Machine Instruction Formats

Class of Packed Half-Word Instructions

Instruction Type	*Data Types Included*	*Data-Type Code*
Operand from stack format	DTP-PACKED-INSTRUCTION	60-77
10-bit immed. operand format	DTP-PACKED-INSTRUCTION	60-77

Class of Full-Word Instructions (all full-word format)

Instruction Type	*Data Types Included*	*Data-Type Code*
Entry instruction	DTP-PACKED-INSTRUCTION	60-77
Function-calling instructions		
	DTP-CALL-COMPILED-EVEN	50
	DTP-CALL-COMPILED-ODD	51
	DTP-CALL-INDIRECT	52
	DTP-CALL-GENERIC	53
	DTP-CALL-COMPILED-EVEN-PREFETCH	54
	DTP-CALL-COMPILED-ODD-PREFETCH	55
	DTP-CALL-INDIRECT-PREFETCH	56
	DTP-CALL-GENERIC-PREFETCH	57
Constants		
	DTP-FIXNUM	10
	DTP-SMALL-RATIO	11
	DTP-SINGLE-FLOAT	12
	DTP-DOUBLE-FLOAT	13
	DTP-BIGNUM	14
	DTP-BIG-RATIO	15
	DTP-COMPLEX	16
	DTP-SPARE-NUMBER	17
	DTP-INSTANCE	20
	DTP-LIST-INSTANCE	21
	DTP-ARRAY-INSTANCE	22
	DTP-STRING-INSTANCE	23
	DTP-NIL	24
	DTP-LIST	25
	DTP-ARRAY	26
	DTP-STRING	27
	DTP-SYMBOL	30
	DTP-LOCATIVE	31

Table 12, continued

Instruction Type	*Data Types Included*	*Data-Type Code*
Constants		
	DTP-LEXICAL-CLOSURE	32
	DTP-DYNAMIC-CLOSURE	33
	DTP-COMPILED-FUNCTION	34
	DTP-GENERIC-FUNCTION	35
	DTP-SPARE-POINTER-1	36
	DTP-SPARE-POINTER-2	37
	DTP-PHYSICAL-ADDRESS	40
	DTP-SPARE-IMMEDIATE-1	41
	DTP-SPARE-POINTER-3	42
	DTP-CHARACTER	43
	DTP-SPARE-POINTER-4	44
	DTP-EVEN-PC	46
	DTP-ODD-PC	47
Value Cell Contents		
	DTP-EXTERNAL-VALUE-CELL-POINTER	4
Illegal Instructions		
	DTP-NULL	0
	DTP-MONITOR-FORWARD	1
	DTP-HEADER-P	2
	DTP-HEADER-I	3
	DTP-ONE-Q-FORWARD	5
	DTP-HEADER-FORWARD	6
	DTP-ELEMENT-FORWARD	7
	DTP-GC-FORWARD	45

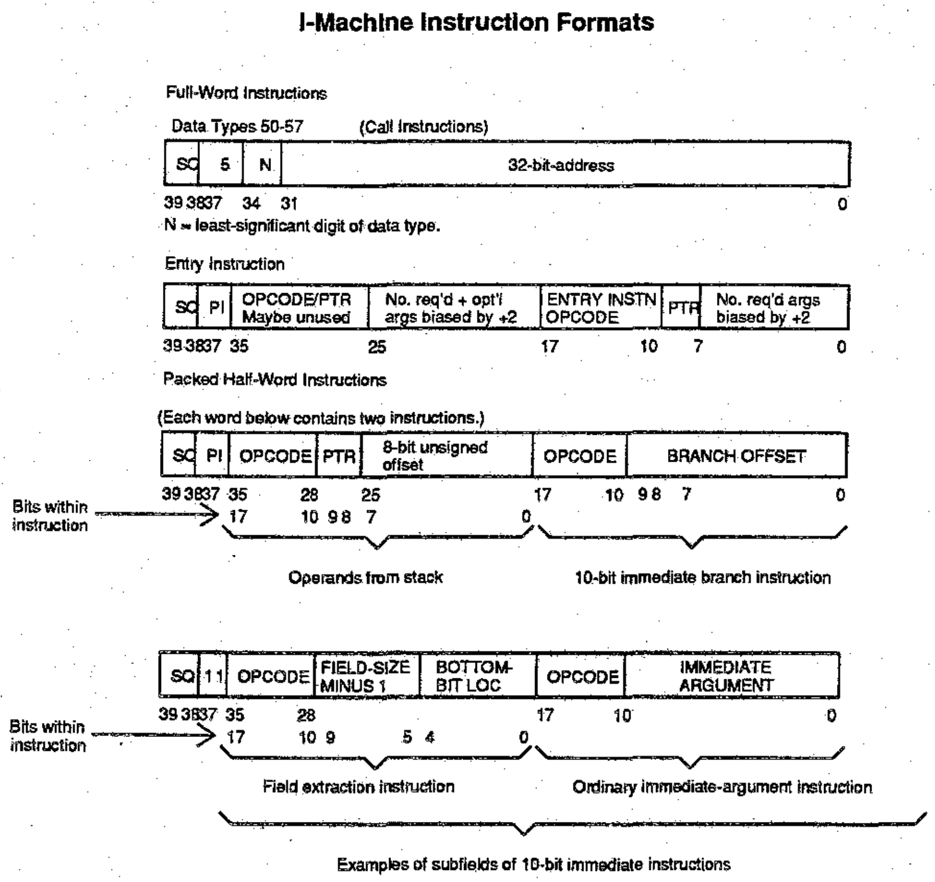


Figure 21. I-machine instruction formats.

Entry-Instruction Format

An entry instruction is a word of type **dtp-packed-instruction** that actually contains one full-word instruction. Its format, shown in Figure 21, is

Bits	*Meaning*
<39:38>	Sequencing code = "add 2 to PC"
<37:36>	**dtp-packed-instruction**
<35:28>	Opcode of second half word, unused
<27:26>	Addressing mode of second half word, unused
<25:18>	Number of required+optional args, biased by +2
<17:10>	**entry** instruction opcode. 1 bit says whether **&rest** is accepted.
<9:8>	Immediate addressing mode
<7:0>	Number of required args, biased by +2

The hardware will dispatch to one of two microcode starting addresses according to the value of the &rest-accepted bit.

Constant Formats

The processor treats any word whose data type is that of an object reference as a constant. The processor pushes the object reference itself onto the control stack and sets its cdr code to **cdr-next** for any object that is pushed onto the control stack, unless otherwise specified.

Value Cell Contents

A word of data type **dtp-external-value-cell-pointer** contains the address of a memory cell. Using a data-read operation, the processor pushes the word contained in the addressed cell onto the control stack, following invisible pointers if necessary. Typically this pointer addresses a symbol's value or function cell.

Illegal Instruction Formats

A word of any data type other than those listed above cannot be executed as an instruction. The processor will trap out if it encounters such a word. A later chapter contains further information on trapping. See the section "Exception Handling."

Packed Half-Word Instruction Formats

This is the most common instruction format. The word with data type **dtp-packed-instruction** contains two 18-bit instructions, which are packed into the word as shown:

```
 ----------------------------------------
|SQ|11|SECND INSTRUCTION|FIRST INSTRUCTION|
 ----------------------------------------
     35                17              0
```

The first instruction executed is called the "even halfword" instruction, and is found in bits 0 through 17. The "odd halfword" instruction is executed later, and is found in bits 18 through 35. Since the data portion of the word is normally only 32 bits, 4 bits are "borrowed" from the data type field. (The ones in bit positions <36-37> are the upper two binary digits of any **dtp-packed-instruction** opcode, a number between 60 and 77 octal.)

Each of the two instructions in this format can be further decomposed. See Figure 21. As the figure shows, there are two basic 18-bit formats.

Format for 10-Bit Immediate Operand

The 10-bit-immediate-operand format is for those instructions that include an immediate operand in their low-order ten bits. The immediate operand can be interpreted as a constant or as an offset -- signed or unsigned, depending on the instruction. There are two special subcases of this instruction format: field extraction instructions and branch and loop instructions.

Format for Field Extraction

The field-extraction format is for instructions used to extract and deposit fields from words of different data types. The field is specified in the instruction by the bottom 10 bits. Bits 0 through 4 specify the location of the bottom bit of the field, -- that is, the *rotate count* -- and bits 5 through 9 specify (field size - 1). For load-byte instructions, **ldb**, **char-ldb**, and the like, the rotate-count that the instruction should specify is (mod (- 32 bottom-bit-location) 32), and for deposit-byte instructions, **dpb** and the like, the rotate-count should specify the bottom-bit location.

The extraction instructions take a single argument. The deposit instructions take two arguments. The first is the new value of the field to deposit into the second argument. It is illegal, though not checked, to specify a field with bits outside the bottom 32 bits.

Format for Branch Instructions

Branch instructions are a subclass of 10-bit-immediate-format instructions. They use the immediate argument as a signed half-word offset.

Format for Operand From Stack

Packed half-word instructions that address the control stack use the operand-from-stack format. They have a 10-bit field that specifies an address into the stack. If one of these instructions takes more than one operand, the addressed operand is the *last* operand of the instruction and the other operands are popped off the top of the stack. If the instruction produces a value, then the value is pushed on top of the stack.

Control Stack Addressing Modes

Operand-from-stack instructions reference operands on the control stack relative to one of three pointers to various regions of the current stack frame. The lower ten-bit field of one of these constitutes the *operand specifier*, whose bits are interpreted as follows. Bits 8 and 9 of the instruction are used to select the pointer, while bits 0 through 7 are used as an unsigned offset. The processor interprets bits 8 and 9 as:

00 Frame Pointer - The address of the operand is the Frame Pointer plus the offset.

01 Local Pointer - The address of the operand is the Local Pointer plus the offset.

10 Stack Pointer - The address of the operand is the Stack Pointer (prior to popping any other operands) plus the offset minus 255, unless the offset is 0.

For example, if the offset is 255, then the operand is the top of stack. Note that this operand will not be popped. If the offset is 1, then the operand is the contents of the word pointed to by (Stack Pointer minus 254). This mode is used for the management of arguments for pop instructions, as described in the next paragraphs.

In the special case when the offset is 0, the operand *is* popped off the top of stack, before any other operands have been popped off (this operand is still the last argument of the function, though). This special case is called the "sp-pop addressing mode." For example, the following sequence could be used to add two numbers, neither of which is to be saved on the stack for later use, and to leave the result of the addition on the stack.

```
push LP|0     ;push arg1 on the stack
push LP|1     ;push arg2 on the stack
add sp-pop    ;pops arg2 then arg1 off stack,
              ;adds, then pushes the result
```

11 Immediate - The last operand is not on the stack at all, but is a fixnum whose value is the offset possibly sign-extended to 32 bits, depending on the instruction. This case is called the "immediate addressing mode," not to be confused with 10-bit immediate *format* instructions, which have no operand specifier since they are always immediate.

In some cases, the stack location address specified is the operand used as an object of the instruction in some way. This case is called "address-operand addressing mode." For instructions that employ the address-operand mode, the immediate and sp-pop modes are illegal.

Note that it is always the *last* argument of an instruction that is specified by the operand specifier of the operand-from-stack format: the others, if there are any, are not explicitly specified by the instruction and are always popped off the stack in order.

Refer to the chapter on function calling for a description of the control stack and the processor's stack pointers. See the section "Control Stack."

3.1.3.2 Arguments: the Data Types Accepted

In the instruction definitions in this document, the *Arguments* field lists the arguments that the instruction requires and the valid data types for these arguments. The data types listed are those that the instruction accepts without taking an error pre-trap. See the section "Operand-Reference Classification."

All numeric instructions, including those listed in the section "Numeric Instructions" as well as **equal-number**, **greaterp**, **lessp**, **plusp**, **minusp**, **zerop**, and **logtest**, accept all numeric data types. The only spare data type that numeric instructions accept is **dtp-spare-number**, which will cause an instruction exception.

The *Exception* field of an instruction definition lists those data types that the instruction accepts as valid (that is, that do not cause an error pre-trap) but that are not supported in hardware.

3.1.3.3 Types of Instruction Exceptions

An instruction exception occurs when an instruction needs to perform some operation that is not an error, but is not directly supported by the hardware. Instruction exceptions are post-traps, called (usually) with whatever arguments the instruction takes. The contract of the trap handler is to emulate the behavior of the particular instruction. See the section "Exception Handling."

The instruction definitions document any instruction exceptions that may occur during execution of the instruction. The description includes the conditions under which an exception will occur, the arguments passed to the exception handler (excluding the trap-vector-index and fault-pc supplied with all traps), and the number of values returned by the exception handler. Exception handlers always return values with **return-kludge**, and TOS is always valid afterwards.

3.1.3.4 Types of Memory References

There is a class of instructions that address main memory (as opposed to stack memory). The operands for these instructions are memory addresses. Different instructions make conceptually different kinds of read and write requests to the memory system. The different types of memory cycles for these different types of memory requests are summarized here and described later in this section. The classification of Lisp data types according to type of operand reference -- data, header, header-forward, and so on -- is made in the chapter on data representation. See the section "Operand-Reference Classification."

Table 13 shows the action taken for each category of data when read from memory in a given type of memory cycle. This table refers only to memory reads and to memory cycles that consist of a read followed by a write. (An instruction that writes memory without reading first is called a "raw write." The table omits these.) Note that the categories overlap.

Table 13. Memory Cycles

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
data-read	0	-	trap	trap	ind	ind	ind	ind	trap	mtrp	trnspt
data-write	1	-	-	trap	ind	ind	ind	ind	trap	mtrp	-
cdr-read	9	-	-	trap	ind	ind	-	-	trap	-	-
bind-read	4	-	-	trap	ind	ind	ind	-	trap	mtrp	trnspt
bind-r-mon	2	-	-	trap	ind	ind	ind	-	trap	ind	trnspt
bind-write	5	-	-	trap	ind	ind	ind	-	trap	mtrp	-
bind-w-mon	3	-	-	trap	ind	ind	ind	-	trap	ind	-
header-rd	6	trap	trap	-	ind	trap	trap	trap	trap	trap	trnspt
struc-offset	7	-	-	-	ind	-	-	-	trap	-	-
scavenge	8	-	-	-	-	-	-	-	trap	-	trnspt
gc-copy	10	-	-	-	-	-	-	-	trap	-	-
raw-read	11	-	-	-	-	-	-	-	-	-	-

Legend:

Normal action

ind — Indirect through forwarding pointer. This also enables transport trap if word addresses oldspace, and transport trap takes precedence if it occurs.

trap — Error trap. Takes precedence over transport.

mtrp — Monitor trap (different trap vector entry than error trap). This also enables transport trap if word addresses oldspace, and transport trap takes precedence if it occurs.

trnspt	Enable transport trap if word addresses oldspace.

Note that the operations described apply only to objects addressed as though they were located in main memory, not those already on the control stack.

If an error occurs during a memory operation, the processor aborts the instruction and invokes a Lisp error handler. The arguments to the error handler are the microstate, and the virtual memory address (VMA). From the microstate, the Lisp handler will look up the type of error in an error table. See the section "Exception Handling."

Data-Read Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
data-read	0	-	trap	trap	ind	ind	ind	ind	trap	mtrp	trnspt

Most operands are fetched with a data-read operation. This reads the word located at the requested memory address. If the word obtained is a forwarding, that is, invisible, pointer (**dtp-header-forward, dtp-element-forward, dtp-one-q-forward,** or **dtp-external-value-cell-pointer**), then the pointer's address field is used as the new address of the cell. The content of this new address is then read and checked to see if it is an invisible pointer. The process is repeated until a non-invisible-pointer data type is encountered. The word finally obtained is returned as the result of the data-read operation. During this pointer following, sequence breaks are allowed so that loops can be aborted. If at any point **dtp-null**, a header (**dtp-header-p, dtp-header-i**), or a special marker (non-invisible pointer -- **dtp-gc-forward**) is encountered, the error causes the instruction performing the data read to take an error trap. If a **dtp-monitor-forward** is encountered, the instruction takes a monitor trap. If a data location that is read contains an address in oldspace and transport traps are enabled for the page containing the word read , a transport trap handler is invoked to evacuate the object and then the data-read is resumed. See the section "I-machine Garbage Collection."

Data-Write Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
data-write	1	-	-	trap	ind	ind	ind	ind	trap	mtrp	-

When most operands are written to memory, a data-write memory read operation is first performed. This checks the requested location to determine whether an invisible pointer is present. If so, the address of the pointer is used as the new address of the cell. The contents of the new address is read and checked to see if it is an invisible pointer. If a header or special marker (**dtp-gc-forward** but not **dtp-null**) is encountered in any location, the error causes the instruction doing the data write to take an error trap. If a **dtp-monitor-forward** is encountered, the instruction takes a monitor trap. If the contents of a location is a forwarding pointer, a check for oldspace is made before indirection. When the process terminates, the contents of the final location, which are being replaced, are not transported. The process is repeated until a non-invisible-pointer data type is found, at which point a write normally follows and the data is stored in the last location, preserving the cdr code of the location into which it stores.

CDR-Read Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
cdr-read	9	-	-	trap	ind	ind	-	-	trap	-	-

Memory references made only to determine the cdr-code of a location use a cdr-read operation. This kind of reference follows pointers of the type **dtp-header-forward** or **dtp-element-forward**, which forward the entire memory word, including the cdr code. (Recall that a **dtp-header-forward** pointer is used by the system to replace an element when it is necessary to change the cdr code of a cell in the middle of a cdr-coded list. See the section "Forwarding (Invisible) Pointers.") The cdr-read operation returns the contents of the cdr-code field of the finally found word.

Forwarding pointers (**dtp-one-q-forward** and **dtp-external-value-cell-pointer**) that forward only the *contents* (that is, the data-type and pointer fields) of the cell are not followed. Instead, the cdr code of the word containing such a pointer is returned.

Having extracted the relevant cdr code, the instruction doing the cdr read takes action according to the value returned, as explained in the section on lists. See the section "Representations of Lists."

If a header or **dtp-gc-forward** data type is encountered, the error causes the instruction making the reference to take an error trap.

Bind-Read Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
bind-read	4	-	-	trap	ind	ind	ind	-	trap	mtrp	trnspt
bind-r-mon	2	-	-	trap	ind	ind	ind	-	trap	ind	trnspt

The binding instructions, **unbind-n, %restore-binding-stack**, and **bind-locative-to-value**, change the value cell, not the *contents* of the value cell, of a variable. **dtp-external-value-cell-pointer** is an invisible pointer that points to the value cell in memory. Since binding should create a new value cell, the system does *not* follow **dtp-external-value-cell-pointer** when doing bindings. In all other respects this operation is the same as a data-read memory operation, except that encountering **dtp-null** does not cause a trap.

A subcategory of this type of operation is the bind-read-no-monitor operation. This operation, as opposed to the normal binding read, does not trap out if a **dtp-monitor-forward** pointer is encountered. Instead, it just follows the pointer.

Bind-Write Operations

Cycle	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-*	*Point-*
Type										*itor*	*er*
bind-write	5	-	-	trap	ind	ind	ind	-	trap	mtrp	-
bind-w-mon	3	-	-	trap	ind	ind	ind	-	trap	ind	-

A bind-write operation is like a data-write memory operation except that it does not follow external-value-cell pointers. See the section "Bind-Read Operations" in *NS Users Manual.* A subcategory of this type of operation is the bind-write-no-monitor operation. This operation, as opposed to the normal binding write, does not trap out if a **dtp-monitor-forward** pointer is encountered. Instead, it just follows the pointer.

Header-Read Operations

Cycle	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-*	*Point-*
Type										*itor*	*er*
header-rd	6	trap	trap		ind	trap	trap	trap	trap	trap	trnspt

Instructions that reference objects represented in memory as structure objects use a header-read operation to access the header. This reads the word at the requested address. If the word is a header, the header is returned. If the word is a header-forward pointer, the address field of this invisible pointer is used as the new address of the header. The word at this new address is checked, and the process repeated until a header is found. If at any point something other than a header or header-forward pointer is found, the error causes the instruction performing the header-read operation to take an error trap. If the data location that is read (without a trap) contains an address in oldspace, a transport trap handler is invoked to evacuate the object and then the header-read is resumed.

Structure-Offset Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
struc-offset	7	-	-	-	ind	-	-	-	trap	-	-

The Lisp operation **%p-structure-offset** uses the struc-offset type of reference to return the structure header. This type of reference follows header-forwarding pointers as necessary and traps out if a **dtp-gc-forward** is encountered. A structure-offset reference is enabled only by bits in a **%memory-read** or block-read type of instruction.

Garbage-Collection Operations

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
scavenge	8	-	-	-	-	-	-	-	trap	-	trnspt
gc-copy	10	-	-	-	-	-	-	-	trap	-	-

Memory references of the types scavenge and gc-copy are used internally by the garbage collector. References of these types trap out when a **dtp-gc-forward** is encountered. Scavenge references perform transports; gc-copy references do not. Either type of reference is enabled only by bits in a **%memory-read** or block-read type of instruction.

Unchecked Operands

Cycle Type	*Code*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
raw-read	11	-	-	-	-	-	-	-	-	-	-

A raw memory reference has all the indirection (pointer following), trapping, and transporting possibilities disabled. During stack encaching and decaching, transfers of data between main memory and the stack cache use raw-read and raw-write operations. **%p-ldb** and **%p-dpb** are among the users of raw references. Note that raw-write operations maintain the modified and ephemeral-reference bits in the PHT just as other write operations do.

3.1.3.5 Top-of-Stack Register Effects

The top-of-stack (TOS) register is a scratchpad location that contains a copy of the contents of the top of the control stack. The possible effects of an instruction on this register affect the code the compiler is allowed to generate. Sometimes the compiler must insert extra **movem** SP|0 instructions to restore the correct value to the TOS register. The TOS register is valid if its contents are known to be identical to the contents of the location indicated by the stack pointer (SP|0); otherwise, the TOS is invalid.

In the instruction descriptions that follow, the possible effects that an instruction can have on the TOS register are indicated by the following phrases:

Valid before The register *must* be valid before the instruction.

Valid after The register will be made valid by the instruction.

Invalid after The register can be made invalid by the instruction.

Unchanged Status after the instruction same as status before, except if an sp-pop operand is used or if the instruction modifies its operand and the operand happens to be the top word in the stack, in which case TOS is invalid after.

3.1.3.6 Cdr Codes of Values Returned

Every operation that returns a value -- this includes all true Lisp operations -- pushes that value on the stack. Thus, after an instruction has executed, the stack no longer contains the instruction's arguments but instead contains the result of the operation. Instructions that do not return a value -- for example, **rplacd, aset, pop** -- pop off all of their arguments. Every instruction that produces a value and pushes it on the stack sets the cdr code of the pushed word to 0 (**cdr-next**). The only exceptions are as follows:

- The start-call instructions produce 3 (illegal in lists) in the cdr-code fields of the frame header on the stack.
- A memory read or block read instruction -- one of **%memory-read**, **%memory-read-addresse**, **%block-n-read**, or **%block-n-read-shift** -- can copy the cdr code of the word from memory into the word on the stack.
- The push-apply-args operation can produce 1 (**cdr-nil**) or 2 (**cdr-normal**) in the cdr-code field of words on the stack.
- The **catch-open** instruction can produce any value in the cdr-code field of certain words in the catch block.
- The **catch-close** instruction produces 2 or 3 in the cdr code of the PC it saves before jumping to an unwind-protect cleanup handler.
- **%p-tag-dpb** can be used to store into the stack.
- **%set-tag** can be used to produce any cdr code but is usually programmed to produce **cdr-next**.
- The instructions **increment, decrement, set-to-car, set-to-cdr, set-to-cdr-push-car** (car pushed with **cdr-next**), **%block-n-read-alu**, and

%pointer-increment store into their stack operands, preserving the cdr code that was in the stack location.

- **movem**, **pop**, **set-sp-to-address-save-tos**, **stack-blt**, **stack-blt-address**, **return-kludge**, **%merge-cdr-no-pop**, and **%set-cdr-code-n** store into their stack operands and set the cdr code to some value other than that of the stack location (that is, these instructions do not preserve the original cdr code). See the section "Revision 0 Stack-blt." See the section "Revision 0 Stack-blt-address." See the section "Revision 0 Return-kludge."

3.2 The Instructions

The I-machine implements 210 instructions in 14 categories. There are:

10	list-function
24	predicate
29	numeric
10	data-movement
8	field-extraction
10	array-operation
19	branch-and-loop
20	block
12	function-calling
4	binding
2	catch
24	lexical-variable-accessing
11	instance-variable-accessing, and
27	subprimitive

instructions.

3.2.1 List-Function Operations

car, cdr, set-to-car, set-to-cdr, set-to-cdr-push-car, rplaca, rplacd, rgetf, member, assoc

The Lisp predicate instructions **eq, eql**, and **endp** are documented elsewhere. The Lisp functions **cons** and **ncons** are implemented in macrocode. Refer also to the following topics:

%allocate-list-block
%allocate-structure-block

car *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 1: *Opcode* 0

arg **dtp-list, dtp-locative, dtp-list-instance**, or **dtp-nil**

Immediate Argument Type Signed

Description
If the type of *arg* is **dtp-list**, pushes the **car** of *arg* on the stack.

If the type of *arg* is **dtp-locative**, pushes the contents of the location *arg* references on the stack.

If the type of *arg* is **dtp-nil**, pushes **nil** on the stack.

Exception
Conditions: Type of *arg* is **dtp-list-instance.**
Arguments: *arg*
Values: 1

Memory Reference Data-read

Register Effects TOS: Valid after

cdr *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg **dtp-list**, **dtp-locative**,
dtp-list-instance, or
dtp-nil

Opcode 1

Immediate Argument Type Signed

Description
If the type of *arg* is **dtp-list**, pushes the **cdr** of *arg* on the stack.

If the type of *arg* is **dtp-locative**, pushes the contents of the location *arg* references on the stack.

If the type of *arg* is **dtp-nil**, pushes **nil** on the stack.

Exception
Conditions: Type of *arg* is **dtp-list-instance**.
Arguments: *arg*
Values: 1

Memory Reference Cdr-read, then data-read if **cdr-normal**

Register Effects TOS: Valid after

set-to-car *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop operand modes undefined)

Value(s) Returned 0

Argument(s) 1:
arg, the address operand, **dtp-list, dtp-locative, dtp-list-instance,** or **dtp-nil**

Opcode 140

Immediate Argument Type Not applicable

Description
Replaces *arg* with the **car** of *arg*. Does not change the cdr code of the operand. See the instruction **car**, page 92.

Exception
Conditions: Type of *arg* is **dtp-list-instance**.
Arguments: *arg* (address of operand as locative)
Values: 0

Memory Reference Data-read

Register Effects TOS: Unchanged

set-to-cdr *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop operand modes undefined)

Value(s) Returned 0

Argument(s) 1:
arg, the address operand, **dtp-list**, **dtp-locative, dtp-list-instance** or **dtp-nil**

Opcode 141

Immediate Argument Type Not applicable

Description
Replaces *arg* with the **cdr** of *arg*. Does not change the cdr code of the operand. See the instruction **cdr**, page 93.

Exception
Conditions: Type of *arg* is **dtp-list-instance.**
Arguments: *arg* (address of operand as locative)
Values: 0

Memory Reference Cdr-read, data-read

Register Effects TOS: Unchanged

set-to-cdr-push-car *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop operand modes undefined)

Value(s) Returned 1

Argument(s) 1:
arg, the address operand, **dtp-list**, **dtp-locative**, **dtp-list-instance**, or **dtp-nil**

Opcode 142

Immediate Argument Type Not applicable

Description
Computes the **car** and the **cdr** of *arg*. Pushes the **car** onto the stack with a cdr code of **cdr-next** and stores the **cdr** back into *arg* leaving the cdr code of the operand unchanged.

Exception
Conditions: Type of *arg* is **dtp-list-instance**.
Arguments: *arg* (address operand as locative)
Values: 1

Memory Reference Data-read, cdr-read, data-read

Register Effects TOS: Valid after

rplaca *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 **dtp-list, dtp-locative** or **dtp-list-instance;**
arg2 any data type

Opcode 200

Immediate Argument Type Signed

Description
Replaces the **car** of *arg1* with *arg2.* Does not change the cdr code

Exception
Conditions: Type of *arg1* is **dtp-list-instance.**
Arguments: *arg1, arg2*
Values: 0

Memory Reference Data-write

Register Effects TOS: Valid before, invalid after

rplacd *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 **dtp-list, dtp-locative**
or **dtp-list-instance**;
arg2 any data type

Opcode 201

Immediate Argument Type Signed

Description
Replaces the **cdr** of *arg1* with *arg2*.

Does not change the cdr code.

Exception
Conditions: Type of *arg1* is **dtp-list-instance.**
Type of **arg1** is **dtp-list** and the cdr code of the referenced cell is not **cdr-normal.**
See the section "Revision 0 Rplacd."

Arguments: *arg1, arg2*
Values: 0

Memory Reference Cdr-read, then data-write

Register Effects TOS: Valid before, invalid after

3.2.1.1 Interruptible Instructions

The next three instructions are interruptible. If a sequence break request arrives while one of these instructions is executing, the instruction is aborted and control passes to the sequence break handler. When the handler returns, the instruction is restarted from the beginning. Similarly, if a page fault or transport trap occurs, the instruction is aborted and restarted from the beginning. None of these instructions store into their arguments. It is possible when processing an extremely long list for the instruction never to complete because sequence breaks occur more often than the time it takes the instruction to complete, or because not all of the pages referenced by the instruction will fit in main memory simultaneously. This condition is detected by software, by comparing the PC on two successive sequence breaks, and causes control to be diverted to a macrocode subroutine that performs the equivalent function of the instruction. This will not happen often.

rgetf *Instruction*

Format Operand from stack *Value(s) Returned* 2

Argument(s) 2: *Opcode* 225
arg1 any data type;
arg2 **dtp-list, dtp-nil,** or **dtp-list-instance**

Immediate Argument Type Signed

Description
Searches the list *arg2* two elements at a time, succeeding if the first element of a pair is **eql** to *arg1*, failing if the end of the list is reached without finding a match. Upon failure, both values returned are **nil**. Upon success, the first value returned is the second element of the matching pair, and the second value returned is the tail of *arg2* whose **car** is that second element. The second value serves as a success/failure indicator and also can be used with **rplaca** to change the property value. The length of the list is supposed to be a multiple of two; if the list is of odd length and a match occurs at the end of the list, an instruction exception occurs so software can decide whether this is an error. If no match occurs, no exception is taken, whether or not the list length is odd. Note that each sublist is subject to the type-checking errors and exceptions that the initial list is subject to. See the section "Interruptible Instructions," page 98.

Exception
Conditions: Type of *arg1* is **dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex,** or **dtp-spare-number** (**eq** test not sufficient).
A match occurs at the end of an odd-length list.
Any sublist of *arg2* is of type **dtp-list-instance**.
Arguments: *arg1, arg2*
Values: 2

Memory reference data-read, cdr-read

Register Effects TOS: Valid before, valid after

member *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2: *Opcode* 226
arg1 any data type;
arg2 **dtp-list**, **dtp-nil**, or **dtp-list-instance**

Immediate Argument Type Signed

Description
Returns **nil** or a tail of *arg2* whose **car** is **eql** to *arg1*. This implements the **cl:member** function and approximates the **zl:memq** function. Note that each sublist is subject to the type-checking errors and exceptions that the initial list is subject to. See the section "Interruptible Instructions," page 98.

Exception
Conditions: Type of *arg1* is **dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex,** or **dtp-spare-number** (**eq** test not sufficient).
Any sublist of *arg2* is of type **dtp-list-instance**.
Arguments: *arg1, arg2*
Values: 1

Memory Reference Cdr-read, data-read

Register Effects TOS: Valid before, valid after

assoc *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2: *Opcode* 227
arg1 any data type;
arg2 **dtp-list**, **dtp-nil**, or **dtp-list-instance**

Immediate Argument Type Signed

Description
Returns **nil** or an element of *arg2* whose car is **eql** to *arg1*. This implements the **cl:assoc** function and approximates the **zl:assq** function. Note that each sublist is subject to the type-checking errors and exceptions that the initial list is subject to. See the section "Interruptible Instructions," page 98.

Exception
Conditions: Type of *arg1* is **dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex,** or **dtp-spare-number** (**eq** test not sufficient).
Any sublist or element of *arg2* is of type **dtp-list-instance**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference Cdr-read, data-read

Register Effects TOS: Valid before, valid after
BAR-1 modified

3.2.2 Predicate Instructions

Binary predicates: **eq**, **eq-no-pop**, **eql**, **eql-no-pop**, **equal-number**, **equal-number-no-pop**, **greaterp**, **greaterp-no-pop**, **lessp**, **lessp-no-pop**, **logtest**, **logtest-no-pop**, **type-member-n** (four instructions), **type-member-n-no-pop** (four instructions). Unary predicates: **endp**, **plusp**, **minusp**, **zerop**.

Refer also to the subprimitive instructions **%unsigned-lessp** and **%ephemeralp**.

eq *Instruction*

eq-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any data type
arg2 any data type

Opcode 270 (274 for no-pop)

Immediate Argument Type Signed

Description
Pushes **t** on the stack if the operands reference the same Lisp object; otherwise, pushes **nil** on the stack. The no-pop version of this instruction leaves the first argument *arg1* on the stack. (Note that, in the presence of forwarding pointers, two references may refer to the same object but not be **eq** or **eql**.)

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

eql *Instruction*

eql-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any data type
arg2 any data type

Opcode 263 (267 for no-pop)

Immediate Argument Type Signed

Description
Returns **t** if the two arguments are **eq** or if they are numbers of the same type with the same value; otherwise returns **nil**. Note that for **dtp-single-float**, +0 and -0 are not **eql**. Also, (**eql** 0 0.0) is false. The no-pop version of this instruction leaves the first argument on the stack. **eql** returns **nil** without trapping any time the data types of the arguments are different. (Note that, in the presence of forwarding pointers, two references may refer to the same object but not be **eq** or **eql**.)

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are equal and one of **dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex,** or **dtp-spare-number** (but *arg1* and *arg2* are not **eq**).
Arguments: *arg1*, *arg2*
Values: 1 for normal version
2 for no-pop version (returns *arg1* to become the non-popped argument).

Memory Reference None

Register Effects TOS: Valid before, valid after

equal-number *Instruction*

equal-number-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 260 (264 for no-pop)

Immediate Argument Type Signed

Description
Tests the two arguments for numerical equality and pushes **t** or **nil** on the stack according to the result. Note that (**equal-number** 0 0.0), which is also written (= 0 0.0), is true, in contrast to (**eql** 0 0.0), which is false. The no-pop version of this instruction leaves the first argument on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both
dtp-fixnum or **dtp-single-float**.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1 for normal version
2 for no-pop version (returns *arg1* to become the
non-popped argument).

Note that **equal-number** or **equal-number-no-pop** will take an exception even if the arguments are **eq** but are not **dtp-fixnum** or **dtp-single-float**.

Memory Reference None

Register Effects TOS: Valid before, valid after

greaterp *Instruction*

greaterp-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 262 (266 for no-pop)

Immediate Argument Type Signed

Description
Tests if *arg1* > *arg2*, and pushes **t** or **nil** on the stack according to the result. The no-pop version of this instruction leaves the first argument on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both
dtp-fixnum or **dtp-single-float**.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1 for normal version
2 for no-pop version (returns *arg1* to become the
non-popped argument).

Memory Reference None

Register Effects TOS: Valid before, valid after

lessp *Instruction*

lessp-no-pop [illegible]

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2
arg1 any numeric data type
arg2 any numeric data type

Opcode 261 (265 for no-pop)

Immediate Argument Type Signed

Description
Tests if *arg1* < *arg2*, and pushes **t** or **nil** on the stack according to the result. The no-pop version of this instruction leaves the first argument on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both
dtp-fixnum or **dtp-single-float**.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1 for normal version
2 for no-pop version (returns *arg1* to become the non-popped argument).

Memory Reference None

Register Effects TOS: Valid before, valid after

logtest *Instruction*

logtest-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 273 (277 for no-pop)

Immediate Argument Type Signed

Description
Pushes **t** on the stack if any of the bits designated by 1s in the first argument are 1s in the second argument; otherwise, pushes **nil**. The no-pop version of this instruction leaves the first argument on the stack. The effect of this instruction is

```
    (not (zerop (logand arg1 arg2))).
```

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric,
but not both **dtp-fixnum**.
Arguments: *arg1*, *arg2*
Values: 1 for normal version
2 for no-pop version (returns *arg1* to become
the non-popped argument).

Memory Reference None

Register Effects TOS: Valid before, valid after

type-member-n *I* *Instruction*

type-member-n-no-pop *I*

Format 10-bit immediate

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 any data type
I **dtp-fixnum** (the immediate)

Opcode 40-43 (44-47 for no-pop)

Immediate Argument Type 10-bit mask

Description
n is a number between 0 and 15 inclusive. Two bits of *n* are part of the opcode and two bits are taken from the immediate argument. *n* specifies which 8-bit field, aligned on a 4-bit boundary, of a 64-bit vector the immediate is specifying. The 8 least-significant bits of the immediate field *I* are then inserted into a background of 64 zero bits. The data type of *arg1*, the argument on top of the stack, is then used to create a bit vector of zeros, except with a one in the slot for the data type. The two vectors are then ANDed together. If the result is nonzero, then **t** is returned, otherwise **nil** is returned. The no-pop version of this instruction leaves the argument on the stack.

The fields specified by **type-member-n** are shown below.

```
n=15   n=13    n=11     n=9     n=7    n=5      n=3     n=1   n=15
 ---\/------\/--  ---\/------\/------\/------\/------\/-----\/---
-----------------------------------------------------------------
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
-----------------------------------------------------------------
63      55      47      39      31      23      15       7      0
\------/\------/\------/\------/\------/\------/\------/\-------/
  n=14    n=12     n=10    n=8     n=6     n=4     n=2     n=0
```

Exception None

Memory Reference None

Register Effects TOS: Valid after

endp *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg **dtp-list**, **dtp-list-instance**, or **dtp-nil**

Opcode 2

Immediate Argument Type Signed

Description
Pushes **t** on the stack if *arg* is **nil**; otherwise pushes **nil**.

Exception None

Memory Reference None

Register Effects TOS: Valid after

plusp *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any numeric data type

Opcode 36

Immediate Argument Type Signed

Description
Pushes **t** on the stack if the argument is a positive number strictly greater than zero; otherwise pushes **nil** on the stack. This is an optimization of (> *arg* 0).

Exception
Type: Arithmetic dispatch
Conditions: Type of *arg* is numeric, but not **dtp-fixnum**
or **dtp-single-float**.
Floating-point exceptions.
Arguments: *arg*
Values: 1

Memory Reference None

Register Effects TOS: Valid after

minusp *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any numeric data type

Opcode 35

Immediate Argument Type Signed

Description
Pushes **t** on the stack if the argument is a negative number strictly less than zero; otherwise pushes **nil** on the stack. This is an optimization of (< *arg* 0).

Exception
Type: Arithmetic dispatch
Conditions: Type of *arg* is numeric, but not **dtp-fixnum**
or **dtp-single-float**.
Floating-point exceptions.
Arguments: *arg*
Values: 1

Memory Reference None

Register Effects TOS: Valid after

zerop *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any numeric data type

Opcode 34

Immediate Argument Type Signed

Description
Pushes **t** on the stack if the argument is zero; otherwise pushes **nil** on the stack. This is an optimization of (= *arg* 0).

Exception
Type: Arithmetic dispatch
Conditions: Type of *arg* is numeric, but not **dtp-fixnum**
or **dtp-single-float**.
Floating-point exceptions.
Arguments: *arg*
Values: 1

Memory Reference None

Register Effects TOS: Valid after

3.2.3 Numeric Operations

add, **sub**, **unary-minus**, **increment**, **decrement**, **multiply**, **quotient**, **ceiling**, **floor**, **truncate**, **round**, **remainder**, **rational-quotient**, **max**, **min**, **logand**, **logior**, **logxor**, **ash**, **rot**, **lsh**, **%32-bit-plus**, **%32-bit-difference**, **%multiply-double**, **%add-bignum-step**, **%sub-bignum-step**, **%divide-bignum-step**, **%lshc-bignum-step**, **%multiply-bignum-step**

Refer also to the following:

equal-number
greaterp
lessp
%unsigned-lessp
plusp
minusp
zerop

If either argument to a numeric instruction is a non-number, then the instruction will take an error pre-trap. Otherwise, if both arguments are hardware supported for the instruction, and no exceptions occur, then the instruction will perform the specified operation. If the arguments are numeric, but the data types of the arguments are not hardware supported or an exception occurs, then the instruction will take an instruction exception and let Lisp code decide whether the arguments, although numeric, are illegal for this instruction.

Note that, if there is no floating-point coprocessor, all the numeric operations will take an instruction exception on encountering operands of type **dtp-single-float**. This instruction exception is in addition to any mentioned in the instruction definitions. See the section "Revision 0 Numeric Operations," page 299.

add *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 300

Immediate Argument Type Unsigned

Description
Pushes the sum of the two arguments on the stack.

See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both
dtp-fixnum or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

sub *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 301

Immediate Argument Type Unsigned

Description
Subtracts *arg2* from *arg1*, and pushes the result on the stack. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both
dtp-fixnum or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

unary-minus *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any numeric data type

Opcode 114

Immediate Argument Type Unsigned

Description
Pushes the negation of *arg* on the stack: if the data type of *arg* is **dtp-fixnum**, subtracts *arg* from zero, and pushes the result, the two's complement of *arg*, on the stack. If *arg* is of **dtp-single-float**, complements the sign bit and pushes the result on the stack. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Type of *arg* is numeric, but not **dtp-fixnum**
or **dtp-single-float.**
Type of *arg* is **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg*
Values: 1

Memory Reference None

Register Effects TOS: Valid after

increment *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg, the address operand, any numeric data type

Opcode 143

Immediate Argument Type Not applicable

Description
Adds 1 to *arg* and stores the result back into the operand.

See the section "Revision 0 Numeric Operations," page 299.

Exception
Conditions: Type of *arg* is numeric, but not **dtp-fixnum** or **dtp-single-float.**
Type of *arg* is **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg* (address operand as locative)
Values: 0

Memory Reference None

Register Effects TOS: Unchanged

decrement *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg can be any numeric data type

Opcode 144

Description
Subtracts 1 from *arg* and stores the result back into the operand. See the section "Revision 0 Numeric Operations," page 299.

Exception
Conditions: Type of *arg* is numeric, but not **dtp-fixnum** or **dtp-single-float**.
Type of *arg* is **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg* (address operand as locative)
Values: 0

Memory Reference None

Register Effects TOS: Unchanged

multiply *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 202

Immediate Argument Type Signed

Description
Computes *arg1***arg2* and pushes the result on the stack. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum** or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

quotient *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Opcode 203

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, and pushes the quotient on the stack. If both operands are integers, the result is the integer obtained by truncating the quotient toward 0; otherwise, the result is a single-precision floating-point number. **quotient** implements the function **zl:/** of two arguments. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum** or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

3.2.3.1 Division Operations That Return Two Values

Note that, if only one of the two results is desired, the division instruction can be followed by an instruction to discard the unwanted result: to discard the first result (quotient), use **set-sp-to-address-save-tos SP|-1**; to discard the second result (remainder), use **set-sp-to-address SP|-1**. Trap handlers for division operations, on encountering these particular instructions, can avoid computing results that are going to be discarded.

ceiling *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 2:
arg1 any numeric data type (an integer)
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Opcode 204

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, pushes the quotient on the stack, then pushes the remainder on the stack. If the remainder is not zero, the resulting quotient (*NOS*) is truncated toward positive infinity, and the remainder (*TOS*) is such that *arg1* = *arg2* * *NOS* + *TOS*. See the section "Division Operations That Return Two Values," page 120. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Arguments: *arg1*, *arg2*
Values: 2
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

floor *Instruction*

Format Operand from stack *Value(s) Returned* 2

Argument(s) 2: *Opcode* 205
arg1 any numeric data type (an integer)
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, pushes the quotient on the stack, then pushes the remainder on the stack. If the remainder is not zero, the resulting quotient (*NOS*) is truncated toward negative infinity, and the remainder (*TOS*) is such that *arg1* = *arg2* * *NOS* + *TOS*. See the section "Division Operations That Return Two Values," page 120. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Arguments: *arg1*, *arg2*
Values: 2
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

truncate *Instruction*

Format Operand from stack *Value(s) Returned* 2

Argument(s) 2:
arg1 any numeric data type (an integer)
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Opcode 206

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, pushes the quotient on the stack, then pushes the remainder on the stack. If the remainder is not zero, the resulting quotient (*NOS*) is truncated toward zero, and the remainder (*TOS*) is such that *arg1* = *arg2* * *NOS* + *TOS*. See the section "Division Operations That Return Two Values," page 120. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both **dtp-fixnum**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Arguments: *arg1*, *arg2*
Values: 2
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

round *Instruction*

Format Operand from stack *Value(s) Returned* 2

Argument(s) 2: *Opcode* 207
arg1 any numeric data type (an integer)
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, pushes the quotient on the stack, then pushes the remainder on the stack. If the remainder is not zero, the resulting quotient (*NOS*) is rounded toward the nearest integer, and the remainder (*TOS*) is such that *arg1* = *arg2* * *NOS* + *TOS*. If the resulting quotient (NOS) is exactly halfway between two integers, it is rounded to the one that is even. See the section "Division Operations That Return Two Values," page 120. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
Arguments: *arg1*, *arg2*
Values: 2
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

remainder *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Opcode 210

Immediate Argument Type Signed

Description
Divides *arg1* by *arg2*, adjusts the remainder to have the same sign as the dividend, and pushes the remainder on the stack. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum.**
arg1 and *arg2* are both **dtp-fixnum,**
but result overflows.
Arguments: *arg1, arg2*
Values: 1
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.
This overflow is only in an intermediate result, some
implementations may in fact return 0 without trapping.

Memory Reference None

Register Effects TOS: Valid before, valid after

rational-quotient *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type;
if **dtp-fixnum**, must not be zero

Opcode 211

Immediate Argument Type Signed

Description

Divides *arg1* by *arg2*, and pushes the quotient on the stack. If both operands are integers and the remainder is not zero, the instruction traps to a routine that returns the ratio (**dtp-small-ratio** or **dtp-big-ratio**) of *arg1/arg2* reduced to lowest terms. If the remainder is zero, the result is an integer if both arguments are integers, or the result type is **dtp-single-float** if either or both arguments are **dtp-single-float** types. See the section "Revision 0 Numeric Operations," page 299.

Exceptions
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not both **dtp-fixnum** or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.

arg1 and *arg2* are both **dtp-fixnum**, but remainder is non-zero.
Floating point exceptions.
Arguments: *arg1, arg2*
Values: 1
Note: the only possible fixnum-fixnum overflow is -1_31. / -1 = 1_31.

Memory Reference None

Register Effects TOS: Valid before, valid after

max *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 213

Immediate Argument Type Signed

Description
Pushes the greater of the two arguments on the stack.

If the arguments are a mixture of rationals and floating-point numbers, and the largest argument is a rational, then the implementation is free to produce either that rational or its floating-point approximation; if the largest argument is a floating-point number of a smaller format than the largest format of any floating-point argument, then the implementation is free to return the argument in its given format or expanded to the larger format. (Note that all of these cases are implemented by trap-handlers, since they all involve data types that cause instruction exceptions.)

The implementation has a choice of returning the largest argument as is or applying the rules of floating-point contagion. If the arguments are equal, then either one of them may be returned.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum** or **dtp-single-float.**
Floating point exceptions.
Arguments: *arg1, arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

min *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 212

Immediate Argument Type Signed

Description
Pushes the lesser of the two arguments on the stack.

If the arguments are a mixture of rationals and floating-point numbers, and the smallest argument is a rational, then the implementation is free to produce either that rational or its floating-point approximation; if the smallest argument is a floating-point number of a smaller format than the largest format of any floating-point argument, then the implementation is free to return the argument in its given format or expanded to the larger format. (Note that all of these cases are implemented by trap-handlers, since they all involve data types that cause instruction exceptions.)

The implementation has a choice of returning the smallest argument as is or applying the rules of floating-point contagion. If the arguments are equal, then either one of them may be returned.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum** or **dtp-single-float**.
Floating point exceptions.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

logand *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 215

Immediate Argument Type Signed

Description
Forms the bit-by-bit logical AND of *arg1* and *arg2*, and pushes the result on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric,
but not both **dtp-fixnum**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

logior *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 217

Immediate Argument Type Signed

Description
Forms the bit-by-bit inclusive OR of *arg1* and *arg2*, and pushes the result on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric,
but not both **dtp-fixnum**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

logxor *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 216

Immediate Argument Type Signed

Description
Forms the bit-by-bit exclusive OR of *arg1* and *arg2*, and pushes the result on the stack.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric,
but not both **dtp-fixnum**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after

ash *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type

Opcode 232

Immediate Argument Type Signed

Description
Shifts *arg1* left *arg2* places when *arg2* is positive, or right |*arg2*| places when *arg2* is negative, and pushes the result on the stack. Unused positions are filled by zeroes from the right or by copies of the sign bit from the left. This is Common Lisp **ash**.

Exception
Type: Arithmetic dispatch
Conditions: Types of *arg1* and *arg2* are numeric, but not
both **dtp-fixnum**.
arg1 and *arg2* are both **dtp-fixnum**,
but result overflows.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference None

Register Effects TOS: Valid before, valid after
DP Op register modified

rot *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 220

Immediate Argument Type Signed

Description
Rotates *arg1* left *arg2* bit positions when *arg2* is positive, or rotates *arg1* right |*arg2*| bit positions when *arg2* is negative, then pushes the result on the stack. Bits that are shifted out one side are shifted in the other side.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after
DP Op register modified

lsh *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 221

Immediate Argument Type Signed

Description
Shifts *arg1* left *arg2* places when *arg2* is positive, or shifts *arg1* right |*arg2*| places when *arg2* is negative. Unused positions are filled by zeroes.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after
DP Op register modified

%32-bit-plus *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 302

Immediate Argument Type Unsigned

Description
Pushes *arg1* + *arg2* on the stack, ignoring overflow (addition uses signed 32-bit arithmetic).

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%32-bit-difference *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 303

Immediate Argument Type Unsigned

Description
Pushes *arg1* - *arg2* on the stack, ignoring overflow.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%multiply-double *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 222

Immediate Argument Type Signed

Description
Multiplies *arg1* * *arg2*, and pushes the two-word result on the stack, low-order word first. Note that, unlike **%multiply-bignum-step**, this is a *signed* multiplication.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%add-bignum-step *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**
arg3 **dtp-fixnum**

Opcode 304

Immediate Argument Type Unsigned

Description
Adds all three arguments, pushes the result on the stack, then pushes the carry (2, 1, or 0) on the stack.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%sub-bignum-step *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**
arg3 **dtp-fixnum**

Opcode 305

Immediate Argument Type Unsigned

Description
Computes ((*arg1* - *arg2*) - *arg3*), pushes this value on the stack, then pushes the value 1 on the stack if a "borrow" was necessary or 2 if a double borrow was necessary; otherwises pushes a 0.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%multiply-bignum-step *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 306

Immediate Argument Type Unsigned

Description
Pushes the 2-word result of multiplying 32-bit unsigned *arg1* by 32-bit unsigned *arg2* on the stack: first the least-significant word, then the most-significant word.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%divide-bignum-step *Instruction*

Format Operand from stack

Value(s) Returned 2

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**
arg3 **dtp-fixnum**, must not be 0

Opcode 307

Immediate Argument Type Unsigned

Description
Performs an unsigned divide of the 64-bit number (+ *arg1* (**ash** *arg2* 32.)) by *arg3*, pushes the quotient on the stack, then pushes the remainder on the stack. Only the low 32 bits of the quotient and remainder are pushed (implying that *arg3* is expected to be greater than or equal to *arg2* using an unsigned compare). If *arg3* is 0, the instruction takes a divide-by-zero error pre-trap.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%lshc-bignum-step *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**
arg3 **dtp-fixnum** (Values not between 0 and 32. inclusive will cause undefined results.)

Opcode 223

Immediate Argument Type Signed

Description
arg1 and *arg2* are unsigned digits. Has the effect of pushing (**ldb** (byte 32. 32.) (**ash** (+ *arg1* (**ash** *arg2* 32.)) *arg3*)) on the stack as a fixnum.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after
DP Op register modified
Rotate-latch modified

3.2.4 Data-Movement Instructions

push, pop, movem, push-n-nils, push-address, set-sp-to-address, set-sp-to-address-save-tos, push-address-sp-relative, stack-blt, stack-blt-address

push *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any data type

Opcode 100

Immediate Argument Type Unsigned

Description
Pushes *arg* on stack.

Exception None

Memory Reference None

Register Effects TOS: Valid after

pop *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
arg2 address-operand

Opcode 340

Immediate Argument Type Not applicable

Description
Pops *arg1* off the top of stack and stores it in the stack location addressed by *arg2*. Note that all 40 bits of the top of stack are stored into the operand.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

movem *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 1

Argument(s) 2:
arg1 any data type
arg2 address operand

Opcode 341

Immediate Argument Type Not applicable

Description
Writes the contents of *arg1*, the top of stack, without popping, into the stack location addressed by *arg2*. Note that all 40 bits of the top of stack are stored into the operand. This instruction restores the top of stack. The way to fix up the top of stack that is equivalent to executing the 3600 **fixup-tos** instruction is to execute **movem** SP|0.

Exception None

Memory Reference None

Register Effects TOS: Valid after

push-n-nils *I* *Instruction*

Format Operand from stack, immediate (sp-pop addressing mode illegal)

Value(s) Returned I

Argument(s) 1:
I **dtp-fixnum**

Opcode 101

Immediate Argument Type Unsigned

Description
Pushes *I* **nils** on the stack. *I* is the immediate argument, which must be greater than 1. (Pushing one **nil** can be done with **plusp** 0.)

Exception None

Memory Reference None

Register Effects TOS: Valid after

push-address *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 1

Argument(s) 1:
arg address operand

Opcode 150

Immediate Argument Type Not applicable

Description
Pushes a locative that points to *arg* onto the top of the stack.

Exception None

Memory Reference None

Register Effects TOS: Valid after

set-sp-to-address *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg is address operand

Opcode 151

Immediate Argument Type Not applicable

Description
Sets the stack pointer to the address of *arg*. This can be used to pop a constant number of values with **set-sp-to-address** SP|-n.

Exception None

Memory Reference None

Register Effects TOS: Valid after

set-sp-to-address-save-tos *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg is address operand

Opcode 152

Immediate Argument Type Not applicable

Description

Sets the stack pointer to the address of *arg*. All forty bits of the new top of stack are set to the value that was previously on the top of stack.

Exception None

Memory Reference None

Register Effects TOS: Valid after

push-address-sp-relative *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg **dtp-fixnum**

Opcode 102

Immediate Argument Type Unsigned

Description
Computes (stack-pointer minus *arg* minus 1) and pushes it on the stack with data type **dtp-locative**. If sp-pop addressing mode is used, the value of the stack-pointer used in calculating the result is the original value of the stack-pointer before the pop.

Exception None

Memory Reference None

Register Effects TOS: Valid after

stack-blt *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 **dtp-locative** pointing to a location in the current stack frame;
arg2 **dtp-locative** pointing to a location in the current stack frame

Opcode 224

Immediate Argument Type Signed

Description
With the value of *arg1* being *TO* and the value of *arg2* being *FROM*, moves all forty bits of the contents of successive locations starting at *FROM* into successive locations starting at *TO* until the top of the stack is moved, and then changes the stack-pointer to point to the last location written. The last word moved is the stack location just below *arg1*. This instruction is not interruptible. Note that this instruction only works if it moves at least one word. Results are undefined if *arg1* is greater than *arg2* (unsigned). See the section "Revision 0 Stack-blt," page 300.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

stack-blt-address *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 2:
arg1 **dtp-locative,** pointing to a location in the current stack frame
arg2 is an address operand

Opcode 352

Immediate Argument Type Not applicable

Description
With the value of *arg1* being *TO* and *arg2* being *FROM-ADDR*, moves all forty bits of the contents of successive locations starting at the address in the location pointed to by *FROM-ADDR* into successive locations starting at *TO* until the top of the stack is moved, and then changes the stack-pointer to point at the last location written. Note that **stack-blt-address** is the same as **stack-blt** except that *arg2* of **stack-blt-address** is the address of the operand, whereas *arg2* for **stack-blt** is the contents of the operand. This instruction is not interruptible. Note that this instruction only works if it moves at least one word. Results are undefined if *arg1* is less than or equal to the address of *arg2*. *FROM-ADDR* is less than or equal to SP after the arguments have been removed. See the section "Revision 0 Stack-blt-address," page 300.

The instruction sequence

```
push arg1
stack-blt-address arg2
```

is equivalent to the instruction sequence

```
push arg1
push-address arg2
stack-blt sp-pop
```

Where *arg2* is a stack-frame address such as, for example, FP|2.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

3.2.5 Field-Extraction Instructions

ldb, dpb, char-ldb, char-dpb, %p-ldb, %p-dpb, %p-tag-ldb, %p-tag-dpb

The following instructions are used to extract and deposit fields from different data types. The extraction instructions take one argument from the stack. The deposit instructions take two arguments from the stack; the first is the new value of the field to deposit into the second argument. Both kinds of instructions take an immediate argument as well. It is illegal, though not checked, to specify a field with bits outside the bottom 32 bits. See the section "Format for Field Extraction," page 82.

ldb *BB FS* *Instruction*

Format Field-Extraction *Value(s) Returned* 1

Argument(s) 2:
arg1 any numeric data type
BB and FS 10-bit immediate

Opcode 170

Description
Extracts the field specified by *BB* and *FS* from *arg1*, then pushes the result on the stack. See the section "Format for Field Extraction," page 82.

Exception
Conditions: Type of *arg1* is numeric, but not **dtp-fixnum**
Arguments: *arg1*
Values: 1
Note: The trap handler is responsible for manually
extracting the byte specifier from the trapped instruction.

Memory Reference None

Register Effects TOS: Valid after

dpb *BB FS* *Instruction*

Format Field-Extraction

Value(s) Returned 1

Argument(s) 3:
arg1 any numeric data type
arg2 any numeric data type
BB and FS 10-bit immediate

Opcode 370

Description
Deposits the value *arg1* into the field in *arg2* specified by *BB* and *FS*, then pushes the result on the stack.

See the section "Format for Field Extraction," page 82.

Exception
Conditions: Types of *arg1* and *arg2* are numeric, but not both **dtp-fixnum**.
Arguments: *arg1*, *arg2*
Values: 1
Note: The trap handler is responsible for manually extracting the byte specifier from the trapped instruction.

Memory Reference None

Register Effects TOS: Valid before, valid after

char-ldb *BB FS* *Instruction*

Format Field-Extraction *Value(s) Returned* 1

Argument(s) 2:
arg1 **dtp-character**
BB and FS 10-bit immediate

Opcode 171

Description
Extracts the field specified by *BB* and *FS* from *arg1*, then pushes the result, a **dtp-fixnum** object, on the stack. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference None

Register Effects TOS: Valid after

char-dpb *BB FS* *Instruction*

Format Field-Extraction *Value(s) Returned* 1

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 **dtp-character**
BB and FS 10-bit immediate

Opcode 371

Description
Deposits the value *arg1* into field in *arg2* specified by *BB* and *FS*, then pushes the result, a **dtp-character** object, on the stack. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference None

Register Effects TOS: Valid before, valid after

%p-ldb *BB FS* *Instruction*

Format Field-Extraction

Value(s) Returned 1

Argument(s) 2:
arg1 any data type
BB and FS 10-bit immediate

Opcode 172

Description

Extracts the field specified by *BB* and *FS* from the bottom 32 bits of the word at the address contained in *arg1,* then pushes the extracted field on the stack. The data type of the result is **dtp-fixnum**. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference Raw-read

Register Effects TOS: Valid after

%p-dpb *BB FS* *Instruction*

Format Field-Extraction

Value(s) Returned 0

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 any Lisp data type
BB and FS 10-bit immediate

Opcode 372

Description
Deposits the value *arg1* into the field in the contents of the location addressed by *arg2* specified by *BB* and *FS*. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference Raw-read followed by raw-write

Register Effects TOS: Valid before, invalid after

%p-tag-ldb *BB FS* *Instruction*

Format Field-Extraction

Value(s) Returned 1

Argument(s) 2:
arg1 any Lisp data type
BB and FS 10-bit immediate

Opcode 173

Description
Extracts the field specified by *BB* and *FS* from the top 8 bits of the word at the address contained in *arg1* and pushes it on the stack. The data type of the result is **dtp-fixnum**. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference Raw-read

Register Effects TOS: Valid after

%p-tag-dpb *BB FS* *Instruction*

Format Field-Extraction

Value(s) Returned 0

Argument(s) 3:
arg1 **dtp-fixnum**
arg2 any Lisp data type
BB and FS 10-bit immediate

Opcode 373

Description
Deposits the value *arg1* into the field specified by *BB* and *FS* in the top 8 bits of the word at the address contained in *arg2*. It is illegal, though not checked, to specify a field with bits outside the top 8 bits. See the section "Format for Field Extraction," page 82.

Exceptions None

Memory Reference Raw-read followed by raw-write

Register Effects TOS: Valid before, invalid after

3.2.6 Array Operations

aref-1, aset-1, aloc-1, setup-1d-array, setup-force-1d-array, fast-aref-1, fast-aset-1, array-leader, store-array-leader, aloc-leader

See the section "I-Machine Array Registers," page 36.

3.2.6.1 Instructions for Accessing One-Dimensional Arrays

Each of the next three instructions accesses a one-dimensional array.

aref-1 *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-array, dtp-array-instance,**
dtp-string, or **dtp-string-instance**
arg2 **dtp-fixnum**

Opcode 312

Immediate Argument Type Unsigned

Description
Pushes the element of *arg1* specified by *arg2* on the stack.

Checks the array *arg1* to insure it is a one-dimensional array, and also checks to insure that the index *arg2* is a fixnum and falls within the bounds of the array.

Exception
Conditions: Type of *arg1* is **dtp-array-instance** or
dtp-string-instance.
arg1 is an array with array-long-prefix = 1.
Arguments: *arg1, arg2*
Values: 1

Memory Reference Header-read, data-read

Register Effects TOS: Valid before, valid after
DP Op register modified

aset-1 *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 3:
arg1 any Lisp data type (See description)
arg2 **dtp-array**, **dtp-array-instance**,
dtp-string, or **dtp-string-instance**
arg3 **dtp-fixnum**

Opcode 310

Immediate Argument Type Unsigned

Description

Stores *arg1* into the element of array *arg2* specified by index *arg3*.

does not change
cdr-codes as [illegible]

Checks the array to insure it is a one-dimensional array, and also checks to insure that the index is a fixnum and falls within the bounds of the array.

When the array-element-type is **dtp-fixnum** or **dtp-character**, takes an error trap unless the data type of *arg1* matches the array element type. When the array element-type is **dtp-character** and the array byte-packing is 8-bit bytes, the instruction takes an error trap if bits <31:8> of *arg1* are nonzero. Similarly, the instruction takes an error trap if bits <31:16> are nonzero in the case of 16-bit characters. It does not check that fixnums are within range when storing into a fixnum array. See the section "Revision 0 Aset-1," page 298.

Exception

Conditions: Type of *arg2* is **dtp-array-instance** or
dtp-string-instance.
arg2 is an array with array-long-prefix = 1.
Arguments: *arg1*, *arg2*, *arg3*
Values: 0

Memory Reference Header-read, data-write

Register Effects TOS: Valid before, invalid after
DP Op register modified

aloc-1 *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2: *Opcode* 313
arg1 **dtp-array**, **dtp-array-instance**,
dtp-string, or **dtp-string-instance**
(array must contain full-word Lisp references);
arg2 **dtp-fixnum**

Immediate Argument Type Unsigned

Description
Pushes a locative to the element of *arg1* addressed by *arg2* on the stack.

Checks the array *arg1* to insure it is a one-dimensional array containing object references (that is, checks that the array-element-type field of the array header is object reference), and also checks to insure that the index *arg2* is a fixnum and falls within the bounds of the array.

Exception
Conditions: Type of *arg1* is **dtp-array-instance** or
dtp-string-instance.
arg1 is an array with array-long-prefix = 1.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference Header-read

Register Effects TOS: Valid before, valid after

3.2.6.2 Instructions for Creating Array Registers

Each of the next two instructions creates an array register describing a one-dimensional array.

setup-1d-array *Instruction*

Format Operand from stack

Value(s) Returned 4

Argument(s) 1:
arg **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**

Opcode 3

Immediate Argument Type Signed

Description
Creates an array register describing array *arg*. The array register will be four words pushed on top of the stack. *arg* must be a one-dimensional array. See the section "I-Machine Array Registers," page 36.

Exception
Conditions: Type of *arg* is **dtp-array-instance** or
dtp-string-instance.
arg is an array with array-long-prefix = 1.
Arguments: *arg*
Values: 4 (array register)

Memory Reference Header-read

Register Effects TOS: Valid after

setup-force-1d-array *Instruction*

Format Operand from stack

Value(s) Returned 4

Argument(s) 1:
arg **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**

Opcode 4

Immediate Argument Type Signed

Description
Creates an array register describing a unidimensional array. *arg* can be any array. The array register will be four words pushed on top of the stack. See the section "I-Machine Array Registers," page 36.

Causes multidimensional arrays to be accessed as if they were unidimensional arrays, with the order of elements depending on row-major or column-major ordering.

Exception
Conditions: Type of *arg* is **dtp-array-instance** or
dtp-string-instance.
arg is an array with array-long-prefix = 1.
Arguments: *arg*
Values: 4 (array register)

Memory Reference Header-read

Register Effects TOS: Valid after

3.2.6.3 Instructions for Fast Access of Arrays

The next two instructions access single dimensional arrays stored in array register variables.

fast-aref-1 *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 the address operand (address of control word of array register)

Opcode 350

Immediate Argument Type Not applicable

Description
Pushes on the stack the element of *arg2* specified by index *arg1*.

Checks to insure that the index is a fixnum and falls within the bounds of the array; if the check fails, the instruction takes an error trap.

This instruction takes an instruction exception if the current event count does not equal the array-register event count. See the section "I-Machine Array Registers," page 36.

Exception
Conditions: Array register is obsolete (current
array-register-event-count does not equal that
encached in the array register).

Arguments: *arg1*, *arg2* (address operand as locative)
Values: 1

Memory Reference Data-read

Register Effects TOS: Valid before, valid after
DP Op register modified

fast-aset-1 *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 3:
arg1 any Lisp data type (See description.)
arg2 **dtp-fixnum**
arg3 the address operand (address of control word of array register)

Opcode 351

Immediate Argument Type Not applicable

Description
Stores *arg1* into the element of *arg3* indexed by *arg2*.

does not change
the old code.

Checks to insure that the index is a fixnum and falls within the bounds of the array. When the array-element-type is **dtp-fixnum** or **dtp-character**, checks the data type of the argument. Does not check that a fixnum is in range when the array-element-type is **dtp-fixnum** and the array-byte-packing field is nonzero. When the array element-type is **dtp-character** and the array byte-packing is 8-bit bytes, the instruction takes an error trap if bits <31:8> of the character are nonzero. Similarly, the instruction takes an error trap if bits <31:16> are nonzero in the case of 16-bit characters. See the section "Revision 0 Fast-aset-1," page 299.

This instruction takes an instruction exception if the current event count does not equal the array-register event count. See the section "I-Machine Array Registers," page 36.

Exception
Conditions: Array register is obsolete (current array-register-event-count does not equal that encached in the array register).
Arguments: *arg1*, *arg2*, *arg3* (address operand as locative)
Values: 0

Memory Reference Data-write

Register Effects TOS: Valid before, invalid after
DP Op register modified

3.2.6.4 Instructions for Accessing Array Leaders

Each of the next three instructions accesses the array leader of any type of array.

array-leader *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**
arg2 **dtp-fixnum** (See description.)

Opcode 316

Immediate Argument Type Unsigned

Description
Pushes on the stack the leader element of *arg1* that is specified by *arg2*.

Checks the array *arg1* to insure it has a leader, and checks the index *arg2* to insure it is a fixnum and falls within the bounds of the array leader; if the checks fail, the instruction takes an error trap.

Exception
Post Trap
Conditions: Type of *arg1* is **dtp-array-instance** or **dtp-string-instance**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference Header-read, data-read

Register Effects TOS: Valid before, valid after

store-array-leader *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 3:
arg1 any Lisp data type
arg2 **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**
arg3 **dtp-fixnum** (See description.)

Opcode 314

Immediate Argument Type Unsigned

Description
Stores *arg1* into the element specified by *arg3* of the leader of *arg2*. Returns no values. Doesn't change the cdr code.

Checks the array *arg2* to insure it has a leader, and checks the index *arg3* to insure it is a fixnum and falls within the bounds of the array leader; if the tests fail, the instruction takes an error trap.

Exception
Conditions: Type of *arg2* is **dtp-array-instance** or **dtp-string-instance**.
Arguments: *arg1*, *arg2*, *arg3*
Values: 0

Memory Reference Header-read, data-write

Register Effects TOS: Valid before, invalid after

aloc-leader *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**
arg2 **dtp-fixnum** (See description.)

Opcode 317

Immediate Argument Type Unsigned

Description
Pushes on the stack a locative to the leader element of *arg1* indexed by *arg2*. Checks the array *arg1* to insure it has a leader, and checks the index *arg2* to insure it is a fixnum and falls within the bounds of the array leader; if the checks fail, the instruction takes an error trap.

Exception
Conditions: Type of *arg1* is **dtp-array-instance** or **dtp-string-instance**.
Arguments: *arg1*, *arg2*
Values: 1

Memory Reference Header-read

Register Effects TOS: Valid before, valid after

3.2.6.5 Branch and Loop Instructions

branch, branch-true{-else}{-and}{-no-pop}{-extra-pop}, Branch-false{-else}{-and}{-no-pop}{-extra-pop}, loop-decrement-tos, loop-increment-tos-less-than

The branch and loop instructions contain a 10-bit signed offset. This offset is in halfwords from the address of the branch or loop instruction. When a conditional branch instruction with an offset of zero is executed and the branch would be taken, the instruction takes an error trap instead. See the section "Revision 0 Branch and Loop Instructions," page 299. This does not apply to the unconditional branch or loop instructions with an offset of zero. If the branch distance is too large to be expressed as a 10-bit signed number, then the compiler must generate the code to compute the target pc and follow this with a **%jump** instruction.

branch *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 0

Argument(s) 1:
I is **dtp-fixnum**

Opcode 174

Immediate Argument Type Not applicable

Description
Continues execution at the location offset *I* halfwords from the current program counter (PC). Note that instruction tracing may ignore this instruction.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

branch-true{-else}{-and}{-no-pop}{-extra-pop} *I* *Instruction*

branch-false{-else}{-and}{-no-pop}{-extra-pop} *I*

Format 10-bit immediate

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
I is **dtp-fixnum**

Opcodes 60-77 (see below)

Immediate Argument Type Not applicable

Description
branch-false branches if the top of stack is **nil**. **branch-true** branches if the top of stack is *not* **nil**. A branch instruction always pops the argument off the top of stack whether or not the branch is taken unless otherwise specified by one of the no-pop conditions.

If the branch is taken, and **-and-no-pop** is specified, the stack is not popped. If **-else-no-pop** is specified, and the branch is not taken, the stack is not popped.

If **extra-pop** is specified then the stack is popped one time in addition to any pop performed as specified by the rest of the instruction. For clarification, see the list below.

If the branch is taken, execution continues at the location offset *I* halfwords from the current program counter (PC). The instruction takes an error trap if the branch condition is met but the offset is zero.

The sixteen combinations of options for the conditional branch instructions are listed here. Note that there are some combinations that the compiler never generates.

Instruction	*Opcode*	*Description*
branch-true	60	Always pop once, whether or not branch taken.
branch-false	70	Always pop once, whether or not branch taken.
branch-true-no-pop	64	Do not pop, whether or not branch taken.
branch-false-no-pop	74	Do not pop, whether or not branch taken.
branch-true-else-no-pop	66	No pop if no branch, pop once if branch.
branch-false-else-no-pop	76	No pop if no branch, pop once if branch.
branch-true-and-no-pop	65	No pop if branch taken, pop if no branch.

branch-false-and-no-pop
75 No pop if branch taken, pop if no branch.

branch-true-and-extra-pop
62 Pop twice if branch, pop once if no branch.

branch-false-and-extra-pop
72 Pop twice if branch, pop once if no branch.

branch-true-else-extra-pop
61 Pop once if branch, pop twice if no branch.

branch-false-else-extra-pop
71 Pop once if branch, pop twice if no branch.

branch-true-extra-pop
63 Always pop twice, whether or not branch taken.

branch-false-extra-pop
73 Always pop twice, whether or not branch taken.

Not generated:
branch-true-and-no-pop-else-no-pop-extra-pop
67 Same as **branch-true**
branch-false-and-no-pop-else-no-pop-extra-pop
77 Same as **branch-false**

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

loop-decrement-tos *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 2:
arg1 any numeric data type
I **dtp-fixnum**

Opcode 175

Immediate Argument Type Not applicable

Description
Decrements *arg1*, the top of stack. If the result is greater than zero, then branches to the location offset from the current program counter (PC) by *I* halfwords. Changes the cdr code of TOS to **cdr-next**. Does not pop the stack, whether or not the branch is taken.

Exception
Conditions: Type of *arg1* is not **dtp-fixnum**.
Decrementing *arg1* overflows (should turn into an error).
See the section "Revision 0 Loop-decrement-tos," page 299.
Arguments: *arg1*
Values: 1 (decremented value; may return to a different PC)

Note: when an instruction exception is taken, the continuation is the PC of the top of the loop, not the successor to the loop instruction. The exception handler may have to alter the contents of the Continuation register. The net effect of taking and returning from an exception is such that the stack is not popped.

Memory Reference None

Register Effects TOS: Valid after

loop-increment-tos-less-than *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 2

Argument(s) 3:
arg1 any numeric data type
arg2 any numeric data type
I **dtp-fixnum**

Opcode 375

Immediate Argument Type Not applicable

Description
If *arg2,* the top of stack, is less than *arg1,* the next on stack, then branches by the number of halfwords from the current program counter (PC) specified by *I.* In any case, increments the top of stack. Changes the cdr code of TOS to **cdr-next**. Does not pop the stack, whether or not the branch is taken.

Exception
Conditions: Type of *arg1* or *arg2* is other than **dtp-fixnum** or **dtp-single-float**.
arg1 and *arg2* are both **dtp-fixnum**, but result overflows.
See the section "Revision 0 Loop-increment-tos-less-than," page 299.
Floating point exceptions.
Arguments: *arg1, arg2*
Values: 2 (bound, incremented value) and may return to different pc.

Note: when an instruction exception is taken, the continuation is the PC of the top of the loop, not the successor to the loop instruction. The exception handler may have to alter the contents of the Continuation register. The net effect of taking and returning from an exception is such that the stack is not popped.

Memory Reference None

Register Effects TOS: Valid before, valid after

3.2.7 Block Instructions

%block-n-read (four instructions), **%block-n-read-shift** (four instructions), **%block-n-read-alu** (four instructions), **%block-n-read-test** (four instructions), **%block-n-write** (four instructions).

A block instruction uses part of its opcode to select the desired Block Address Register (BAR). A BAR is an internal register that must be loaded by means of a **%write-internal-register** instruction before any of the block instructions are executed. For the instructions that use the 10-bit immediate format, the argument is the following mask of bits:

cycle-type <9:6> (4 bits) Select one of the twelve memory-cycle types. See the section "Types of Memory References."

fixnum-only <5> (1 bit) If set, the instruction will take an error trap if the memory data type is not **dtp-fixnum**.

set-cdr-next <4> (1 bit) For **%block-n-read** and **%block-n-read-shift**: if set, the cdr code of the result is 0; otherwise, the cdr code of the result is the cdr code of memory.

last-word <3> (1 bit) If set, do not prefetch words after this one.

no-increment <2> (1 bit) If set, do not increment the Block Address Register (BAR) after executing this instruction.

If an invisible pointer is fetched from memory, and the memory-cycle type specifies that the invisible pointer should be followed, the BAR is always changed to point to the new location. If the BAR is incremented, that happens afterwards.

The **%block-n-read-shift** instruction uses the rotate-latch register and the byte-r and byte-s fields of the DP Op register. DP Op is an internal register that must be loaded by means of a **%write-internal-register** instruction before the **%block-n-read-shift**, **%block-n-read-alu**, or **%block-n-read-test** instruction is executed.

%block-n-read *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 1:
I 10-bit immediate

Opcodes 120-123

Immediate Argument Type Not applicable

Description
In accordance with the setting of the bits in the immediate control mask, reads the word addressed by the contents of the Block Address Register (BAR) specified by *n*, and pushes it on the stack. *n* is a number between 0 and 3 inclusive that is part of the opcode. The specified BAR is incremented according to the bit in the mask as a side effect.

Exception None

Memory Reference Cycle-type specified by instruction

Register Effects TOS: Valid after

%block-n-read-shift *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 1:
I 10-bit immediate

Opcodes 124-127

Immediate Argument Type Not applicable

Description
Reads the word addressed by the contents of the Block Address Register (BAR) specified by *n* and rotates it left by the amount specified in the byte-r field of the DP Op register. The top (byte-s + 1) bits come from this rotated word, and the bottom bits come from the rotate-latch register, and this value is pushed onto the stack. The rotate-latch register is then loaded from rotated memory word. The effect of this operation is to perform a **dpb** (deposit-byte) of the word from memory into the rotate-latch register. *n* is a number between 0 and 3 inclusive that is part of the opcode. The specified BAR is incremented according to the bit in the immediate-operand mask as a side effect. See the section "Revision 0 %Block-n-read-shift," page 298.

Exception None

Memory Reference Cycle-type specified

Register Effects TOS: Valid after

%block-n-read-alu *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg is any numeric data type

Opcodes 160-163

Immediate Argument Type Not applicable

Description
Performs the ALU operation specified in the alu-op field of the DP Op register using *arg* and the word addressed by the contents of the Block Address Register (BAR) specified by *n* as operands. *n* is a number between 0 and 3 inclusive that is part of the opcode. Writes the result of the ALU operation back into the addressed operand, *arg*. The cdr code of the operand is set to the cdr code from memory. The specified BAR is incremented as a side effect.

The values used for the block instruction mask bits are

```
CYCLE TYPE -- data read
FIXNUM-ONLY -- the usual generic-arithmetic post traps apply
SET-CDR-NEXT -- not applicable
LAST-WORD -- false
NO-INCREMENT -- false
```

Exception
Conditions: Traps according to the generic-arithmetic traps associated with the specified ALU operation.
Arguments: *arg* (address operand as locative)
Values: 0 (increments the BAR)
Note: The operation to be performed is indicated by the DP Op register. The trap handler must save this away before it can get clobbered by other processes, interrupt handlers, or complex instructions. See the section "Revision 0 %Block-n-read-alu," page 298.

Memory Reference Data-read

Register Effects TOS: Unchanged

%block-n-read-test *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 2:
arg(s) can be any Lisp data type, except for when a test that requires **dtp-fixnum** is selected

Opcodes 130-133

Immediate Argument Type Not applicable

Description
Performs the test selected by the contents of the condition field and alu-op fields of the DP Op register. See the section "Revision 0 %Block-n-read-test," page 299. Some of the tests that could be performed are:

ephemeralp(memory (BAR))
oldspacep(memory (BAR))
eq(memory(BAR),top-of-stack)
logtest(memory(BAR),top-of-stack)

where memory(BAR) specifies the object reference addressed by the *n*th BAR. (*n* is a number between 0 and 3 inclusive that is part of the opcode.) Does not pop arguments off the stack.

If the test succeeds, transfers control to the program counter next on the stack. If the test fails, increments the BAR contents. Execution then proceeds with the next instruction.

This instruction is typically used for searching tables and bitmaps, and by the garbage collector. Note that the logtest option produces meaningful results only for **dtp-fixnum** operands; in particular, it does not work for **dtp-bignum** operands. (Actually, the logtest test ignores the data type of its operand.) Typically, the programmer would set the fixnum-only bit in the 10-bit immediate field when using this test. See the section "Block Instructions," page 169. The oldspacep test is true exactly when a transport trap would occur if the cycle type allowed it. For this to be useful, the cycle type selected for **%block-n-read-test** oldspacep test must disallow transport traps. See the section "Revision 0 %Block-n-read-test," page 299.

Exception None

Memory Reference Cycle-type specified.

Register Effects TOS: Valid before for 2-operand tests, valid after

%**block-n-write** *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 1:
arg can be any Lisp data type

Opcodes 30-33

Immediate Argument Type Signed

Description
Writes *arg* into the word addressed by the contents of the Block Address Register (BAR) specified by *n*. *n* is a number between 0 and 3 inclusive that is part of the opcode. All 40 bits, including cdr code, of this word are written into memory. The specified BAR is incremented as a side effect. If *arg* is immediate, the tag bits will specify **dtp-fixnum** and **cdr-next**.

Exception None

Memory Reference Raw-write

Register Effects TOS: Unchanged

3.2.8 Function-Calling Instructions

dtp-call-compiled-even, **dtp-call-compiled-odd**, **dtp-call-indirect**, **dtp-call-generic**, and the **-prefetch** versions of these last four, **start-call**, **finish-call-n**, **finish-call-apply-n**, **finish-call-tos**, **finish-call-apply-tos**, **entry-rest-accepted**, **entry-rest-not-accepted**, **locate-locals**, **return-single**, **return-multiple**, **return-kludge**, **take-values**

3.2.8.1 Function-Calling Data Types

Each of the following data types when executed as an instruction starts a function call. Only very brief descriptions of these instructions are presented in this chapter. Complete information is contained in a separate chapter. See the section "Function Calling, Message Passing, Stack-Group Switching," page 241.

dtp-call-compiled-even *Instruction*

dtp-call-compiled-even-prefetch *Instruction*

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1:
Included in the instruction is *addr*,
the address of the first
instruction to be executed
in the target function.

Immediate Argument Type Not applicable

Description
Starts a function call that will commence execution at the even instruction of the word addressed by *addr*. The prefetch version of this instruction indicates that the hardware should initiate an instruction-prefetch operation. See the section "Starting a Function Call," page 249.

Exception None

Memory Reference None

Register Effects TOS: Valid after

dtp-call-compiled-odd *Instruction*

dtp-call-compiled-odd-prefetch *Instruction*

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1:
Included in the instruction is *addr*,
the address of the first
instruction to be executed
in the target function

Immediate Argument Type Not applicable

Description
Starts a function call that will commence execution at the odd instruction of the word addressed by *addr*. The prefetch version of this instruction indicates that the hardware should initiate an instruction-prefetch operation. See the section "Starting a Function Call," page 249.

Exception None

Memory Reference None

Register Effects TOS: Valid after

dtp-call-indirect *Instruction*

dtp-call-indirect-prefetch *Instruction*

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1
Included in the instruction is *addr*, the address of a word, whose contents can be of any data type. The contents of the word is the function to call.

Immediate Argument Type Not applicable

Description
Starts a call of the function addressed by *addr* or by a forwarding pointer addressed by *addr*. Use of the prefetch version suggests to the hardware that an instruction-prefetch operation is desirable. See the section "Starting a Function Call," page 249.

Exception None

Memory Reference Data-read

Register Effects TOS: Valid after

dtp-call-generic *Instruction*

dtp-call-generic-prefetch *Instruction*

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1:
Included in the function is *addr*, the address of a generic function

Immediate Argument Type Not applicable

Description
Starts a call of the generic function addressed by *addr*. Use of the prefetch version suggests to the hardware that an instruction-prefetch operation is desirable. See the section "Calling a Generic Function," page 277.

Exception None

Memory Reference None

Register Effects TOS: Valid after

3.2.8.2 Instructions for Starting and Finishing Calls

The following instructions are used to implement function calling. Only brief descriptions of these are presented here. See the section "Function Calling, Message Passing, Stack-Group Switching," page 241.

start-call *Instruction*

Format Operand from stack *Value(s) Returned* Not applicable

Argument(s) 1:
arg is any data type

Opcode 10

Immediate Argument Type Signed

Description
Starts a function call of the function specified by *arg*. See the section "Starting a Function Call," page 249.

Exception None

Memory Reference Data-read (sometimes)

Register Effects TOS: Valid after

finish-call-n *I* *Instruction*

finish-call-n-apply *I*

Format 10-bit immediate

Value(s) Returned Not applicable

Argument(s) 1:
I **dtp-fixnum**

Opcode 134 (135 for apply)

Immediate Argument Type Unsigned

Description
Finishes a function-calling sequence: builds the new stack frame, checks for control stack overflow, and enters the called function at the appropriate starting instruction. The low-order eight bits of the immediate argument *I* specify a number that is equal to one more than the number of arguments explicitly supplied with the call, including the apply argument but not including the extra argument if any. For example, if one argument is supplied with **finish-call-n**, then *I<7:0>* = 2.

The two high-order bits of *I* are the *value-disposition*, which specifies what should be done with the result of the called function. The possible values of value-disposition are:

- Effect
- Value
- Return
- Multiple

The function-calling chapter explains the meaning of this field. See the section "Finishing the Call," page 253.

finish-call-n-apply is the same as **finish-call-n**, except that its use indicates that the top word of the stack is a list of arguments.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

finish-call-tos *I* *Instruction*

finish-call-tos-apply *I*

Format 10-bit immediate

Value(s) Returned Not applicable

Argument(s) 2:
I **dtp-fixnum**
arg **dtp-fixnum**

Opcode 136 (137 for apply)

Immediate Argument Type Unsigned

Description
Finishes a function-calling sequence: builds the new stack frame, checks for control stack overflow, and enters the called function at the appropriate starting instruction. *arg*, which is popped off the top of stack, specifies the number of arguments explicitly supplied with the call,including the apply argument in the case of **finish-call-tos-apply**. Note that *arg* differs from the immediate argument count in **finish-call-n** by not including the bias of +1.

The two high-order bits of the immediate argument *I* are the *value-disposition*, which specifies what should be done with the result of the called function. The possible values of value-disposition are:

- Effect
- Value
- Return
- Multiple

The function-calling chapter explains the meaning of this field. The low-order eight bits of I are ignored by this instruction. See the section "Finishing the Call," page 253.

finish-call-tos-apply is the same as **finish-call-tos**, except that its use indicates that the top word of the stack is a list of arguments.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

entry-rest-accepted *Instruction*

entry-rest-not-accepted

Format Entry instruction

Value(s) Returned Not applicable

Argument(s) 2:
arg1 8-bit immediate
arg2 8-bit immediate

Opcode 176 (177 for not-accepted)

Immediate Argument Type Unsigned

Description
Performs an argument match-up process that either takes an error trap, if the wrong number of arguments has been supplied, or adjusts the control stack and branches to the appropriate instruction of the entry vector or to the instruction after the entry vector.

arg1 is two greater than the number of arguments that the function requires, and *arg2* is two greater than the number of required arguments plus the number of optional arguments that the function will accept. See the section "Entry-Instruction Format," page 81.

The difference between **entry-rest-accepted** and **entry-rest-not-accepted** is in how the argument matchup and stack-adjustment process are controlled as explained in the chapter on function calling. See the section "Function Entry," page 257. See the section "Revision 0 Entry-rest-accepted," page 299.

Exception See the section "Trapping Out of Entry and Restarting," page 266.

Memory Reference See the section "Pull-apply-args," page 261.

Register Effects TOS: Invalid after

locate-locals *Instruction*

Format Operand from stack

Value(s) Returned Not applicable

Argument(s) 0

Opcode 50

Immediate Argument Type Not applicable

Description
Pushes (control-register.arg_size - 2) onto the stack, as a fixnum. This is the number of spread arguments that were supplied (this is less than the number of spread arguments now in the stack if some **&optional** arguments were defaulted); sets LP to (new-SP - 1) so that LP|0 is now the **&rest** argument and LP|1 is the argument count; and sets control-register.arg_size to (LP - FP). Note that (new-SP - 1) here refers to the SP after the incrementation caused by this instruction pushing its result. Thus the value of LP *after* the instruction is equal to the value in the SP *before* the instruction. See the section "Pull-apply-args," page 261.

Exception None

Memory Reference None

Register Effects TOS: Valid after

return-single *I* *Instruction*

Format 10-bit immediate

Value(s) Returned Not applicable

Argument(s) 1:
I (should be 1000(octal),
1040(octal), or 1041(octal),
but not checked)

Opcode 115

Immediate Argument Type Unsigned

Description
Specifies the value to be returned on the top of stack according to the immediate operand: 1000(octal), the current top of stack; 1040(octal), **nil**; 1041(octal), **t**. When the value disposition is "for value" or "for multiple," the cdr code of the top of stack is set to **cdr-next**. See the section "Revision 0 Return-single," page 299. Removes the returning function's frames from the control and binding stacks; unthreads catch blocks and executes unwind-protects; restores the state of the caller; and resumes execution of the caller with the returned values on the stack in the form specified by the caller. May do a check-preempt-request operation. See the section "Function Returning," page 266.

Exception None

Memory Reference None

Register Effects TOS: Valid before if TOS is the source of the operand. Status afterwards is determined by value disposition and seen as status after **finish-call** in the caller. If the value disposition is for-effect, then the TOS register is invalid; otherwise, it is valid.

return-multiple *Instruction*

Format Operand from stack, immediate or sp-pop addressing modes only

Value(s) Returned Not applicable

Argument(s) 1: arg is **dtp-fixnum**, non-negative

Opcode 104

Immediate Argument Type Unsigned

Description
Returns, in accordance with the value disposition specified by the contents of the Control register, the number of values specified by *arg* in a multiple group, which includes as the top entry the number of values returned, on top of the stack. Removes the returning function's frames from the control and binding stacks, unthreads catch blocks, restores the state of the caller, and resumes execution of the caller with the returned values on the stack in the form specified by the caller. May perform a check-preempt-request operation. See the section "Function Returning," page 266.

Exception None

Memory Reference None

Register Effects TOS: Status afterwards is determined by value disposition and seen as status after **finish-call** in caller

return-kludge *Instruction*

Format Operand from stack, immediate or sp-pop addressing modes only

Value(s) Returned Not applicable

Argument(s) 1: arg **dtp-fixnum**, non-negative

Opcode 105

Immediate Argument Type Unsigned

Description

Returns the number of values specified by *arg* on top of the stack, ignoring the value-disposition. Removes the returning function's frames from the control and binding stacks, unthreads catch blocks, restores the state of the caller, and resumes execution of the caller. May perform a check-preempt-request operation. Used for certain internal stack-manipulating subroutines and for all trap handlers. See the section "Function Returning," page 266.

Exception None

Memory Reference None

Register Effects TOS: Valid after

take-values *I* *Instruction*

Format Operand from stack, immediate addressing mode only
Argument(s) 1:
I

Value(s) Returned *arg*

Opcode 106

Immediate Argument Type Unsigned

Description
Pops a multiple group of values off the top of stack, using the first value as the number of additional words to pop. Pushes the number of words specified by *arg* back on the stack, discarding extras if too many values are in the multiple group, or pushing enough **nils** to equal the number desired if too few values are in the multiple group.

Exception None

Memory Reference None

Register Effects TOS: Valid after

3.2.9 Binding Instructions

bind-locative-to-value, bind-locative, unbind-n, %restore-binding-stack

Instructions that perform binding operations check for stack overflow using the contents of the Binding-Stack-Limit register as the limit. See the section "Binding Stack," page 244. Those that perform unbinding operations check for stack underflow. See the section "Revision 0 Binding Instructions," page 298. These take an error trap if an unbinding instruction tries to undo a binding and control-register.cleanup-binding = 0. There is no fence-post error in the case of a **%restore-binding-stack** that is a no-op because the two pointers are equal; the instruction never traps in this case.

bind-locative-to-value *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 **dtp-locative**
arg2 any data type

Opcode 236

Immediate Argument Type Signed

Description
Pushes *arg1* onto the binding stack, along with the contents of the cell it points to, then stores *arg2* into the location pointed to by *arg1*. Does not change the cdr-code. Copies the Control register binding-cleanup bit into bit 38 of *arg1* on the binding stack and sets this Control register bit to 1. Does not follow external-value-cell pointers as invisible pointers when reading and writing the cell. Takes an error trap if the binding-stack pointer would be greater than the contents of the Binding-Stack-Limit register. See the section "Binding Stack," page 244.

Exception None

Memory Reference Bind-read, followed by two raw-writes, followed by bind-write

Register Effects TOS: Valid before, invalid after
BAR-1 is modified

bind-locative *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 1:
arg **dtp-locative**

Opcode 5

Immediate Argument Type Signed

Description
Pushes *arg* onto the binding stack, along with the contents of the cell it points to. Copies the Control register binding-cleanup bit into bit 38 of *arg* on the binding stack and sets this Control register bit to 1. Does not follow external-value-cell pointers as invisible pointers when reading the cell. Takes an error trap if the binding-stack pointer would be greater than the contents of the Binding-Stack-Limit register. See the section "Binding Stack," page 244.

Exception None

Memory Reference Bind-read, followed by two raw-writes

Register Effects TOS: Invalid after
BAR-1 is modified

unbind-n *Instruction*

Format Operand from stack (only sp-pop operands and the immediate constant 1 are legal)

Value(s) Returned 0

Argument(s) 1:
arg **dtp-fixnum**

Opcode 107

Immediate Argument Type Unsigned

Description
Unbinds the top *arg* variables on the binding stack. It unbinds a variable by popping the variable's old value and the locative to that variable off the binding stack and storing the old value back into the location pointed to by the locative with the cdr-code unchanged. Copies bit 38 of each locative word on the binding stack into the Control register binding-cleanup bit as it pops the locative. After all the unbindings have been accomplished, does a check-preempt-request operation. See the section "Binding Stack," page 244. See the section "Revision 0 Unbind-n," page 300.

Exception None

Memory Reference Two bind-reads, followed by bind-write

Register Effects TOS: Unchanged

%restore-binding-stack *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 1:
arg dtp-locative

Opcode 6

Immediate Argument Type Signed

Description
Unbinds special variables until the binding-stack pointer equals *arg*, that is, until all variables up to the one pointed to by *arg* have been unbound. It unbinds a variable by popping the variable's old value and the locative to that variable off the binding stack and storing the old value back into the location pointed to by the locative. Copies bit 38 of each locative word on the binding stack into the Control register binding-cleanup bit as it pops the locative. After all the unbindings have been accomplished, does a check-preempt-request operation. It is legal for *arg* to equal the binding-stack pointer at the beginning of the instruction; in this case, the instruction does nothing. See the section "Binding Stack," page 244.

with the cdr code unchanged

Exception None

Memory Reference Two bind-reads, followed by bind-write

Register Effects TOS: Valid after

3.2.10 Catch Instructions

catch-open, catch-close

Catch Blocks

A catch block is a sequence of words in the control stack that describes an active catch or unwind-protect operation. All catch blocks in any given stack are linked together, each block containing the address of the next outer block. They are linked in decreasing order of addresses. An internal register (scratchpad location) named *catch-block-pointer* contains the address of the innermost catch block, as a **dtp-locative** word, or contains **nil** if there are no active catch blocks. The address of a catch block is the address of its *catch-block-pc* word.

The format of a catch block for the catch operation is:

Word Name	*Bit 39*	*Bit 38*	*Contents*
catch-block-tag	0	invalid flag	any object reference
catch-block-pc	0	0	catch exit address
catch-block-binding-stack-pointer	0	0	binding stack level
catch-block-previous	extra-arg	cleanup-catch	previous catch block
catch-block-continuation	value-disposition		continuation

The format of a catch block for the unwind-protect operation is:

Word Name	*Bit 39*	*Bit 38*	*Contents*
catch-block-pc	0	0	cleanup handler
catch-block-binding-stack-pointer	0	1	binding stack level
catch-block-previous	extra-arg	cleanup-catch	previous catch block

The *catch-block-tag* word refers to an object that identifies the particular catch operation, that is, the first argument of **catch-open** or **catch-close**. The catch-block-invalid-flag bit in this word is initialized to 0, and is set to 1 by the **throw** function when it is no longer valid to throw to this catch block; this addresses a problem with aborting out of the middle of a throw and throwing again. This word is not used by the unwind-protect operation and is only known about by the **throw** function, not by hardware.

The *catch-block-pc* word has data type **dtp-even-pc** or **dtp-odd-pc.** For a *catch* operation, it contains the address to which **throw** function should transfer control. For an *unwind-protect* operation, it contains the address of the first instruction of the cleanup handler. The cdr code of this word is set to zero (cdr-next) and not used. For a catch operation with a value disposition of Return, the catch-block-pc word contains **nil.**

The *catch-block-binding-stack-pointer* word contains the value of the binding-stack-pointer hardware register at the time the catch or unwind-protect operation started. An operation that undoes the catch or unwind-protect will undo special-variable bindings until the binding-stack-pointer again has this value. The cdr-code field of this word uses bit 38 to distinguish between catch and unwind-protect; bit 39 is set to zero and not used.

The *catch-block-previous* word contains a **dtp-locative** pointer to the catch-block-pc word of the previous catch block, or else contains **nil.** The cdr-code field of this word saves two bits of the Control register that need to be restored.

The *catch-block-continuation* word saves the Continuation hardware register so that a **throw** function can restore it. The cdr-code field of this word saves the value disposition of a catch; this tells the **throw** function where to put the values thrown. This word is not used by the unwind-protect operation.

The compilation of the **catch** special form is approximately as follows:

> Code to push the catch tag on the stack.
> Push a constant PC, the address of the first instruction after the catch.
> A **catch-open** instruction.
> The body of the catch.
> A **catch-close** instruction.
> Code to move the values of the body to where they are wanted; this usually includes removing the 5 words of the catch block from the stack.

The compilation of the **unwind-protect** special form is approximately as follows:

> Push a constant PC, the address of the cleanup handler.
> A **catch-open** instruction.
> The body of the unwind-protect.
> A **catch-close** instruction.
> Code to move the values of the body to where they are wanted; this usually includes removing the 3 words of the catch block from the stack.

Somewhere later in the compiled function:

> The body of the cleanup handler.
> A **%jump** instruction.

Catch blocks are created in the stack by executing the **catch-open**/unwind-protect instruction, and they are removed from the stack by executing the **catch-close** instruction.

An unwind-protect cleanup handler terminates with a **%jump** instruction. This instruction checks that the data type of the top word on the stack is **dtp-even-pc** or **dtp-odd-pc**, jumps to that address, and pops the stack. In addition, if bit 39 of the top word on the stack is 1, it stores bit 38 of that word into control-register.cleanup-in-progress. If bit 39 is 0, it leaves the control register alone.

catch-open *N* *Instruction*

Format 10-bit immediate *Value(s) Returned* 2 or 3

Argument(s) 1:
N **dtp-fixnum**

Opcode 376

Description
This instruction has two versions, catch and unwind-protect, which are specified by bit 0 of the immediate argument, *n*. Bit 0 is for catch, bit 1 for unwind-protect. Bits 6 and 7 of *n* contain the value disposition. Bits 1-5 and 8-9 must be 0. This instruction, when bit 0 is 1 (unwind-protect), must be preceded by instructions that push the catch-block-pc on the stack. When bit 0 is 0 (catch), preceding instructions must push the catch-block-tag and the catch-block-pc as well. See the section "Catch Blocks," page 192. The catch version operates as follows:

1. Push the binding-stack-pointer, with 0 in the cdr code.

2. Push the catch-block-pointer, with control-register.extra-arg and control-register.cleanup-catch bits in the cdr code.

3. Push the Continuation register, with bits 6 and 7 of the **catch-open** instruction in the cdr code.

4. Set catch-block-pointer to the value stack-pointer had at the beginning of the instruction, and set control-register.cleanup-catch to 1.

The unwind-protect version operates as follows:

1. Push the binding-stack-pointer, with 1 in the cdr code.

2. Push the catch-block-pointer, with control-register.extra-arg and control-register.cleanup-catch bits in the cdr code.

3. Set catch-block-pointer to the value stack-pointer had at the beginning of the instruction, and set control-register.cleanup-catch to 1.

Exception None

Memory Reference None

Register Effects TOS: Valid after

catch-close *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0 *Opcode* 51

Description
The compiler emits this instruction at the end of a catch or unwind-protect operation. It is used internally to the **throw** function and is called as a subroutine by the **return** instructions when they find the **control-register.cleanup-catch** bit set. Instruction operation is:

1. Set the virtual memory address to the contents of the catch-block-pointer register and fetch three words: catch-block-pc, catch-block-binding-stack-pointer, and catch-block-previous. These words will always come from the stack cache, but the instruction may not need to rely on that.

2. If catch-block-binding-stack-pointer does not equal binding-stack-pointer, undo some bindings. This can be done by calling the **%restore-binding-stack-level** instruction as a subroutine. The instruction can be aborted (for example, by a page fault) and retried.

3. Restore the catch-block-pointer register, control-register.cleanup-catch bit, and control-register.extra-argument bit that were saved in the catch-block-previous word.

4. Check the unwind-protect flag which is bit 38 of the catch-block-binding-stack-pointer word. If this bit is 0, the instruction is done. Note that stack-pointer is not changed. If this bit is 1, push the next PC (or the current PC if **catch-close** was called as a subroutine by **return**) onto the stack, with the current value of control-register.cleanup-in-progress in bit 38 and 1 in bit 39; then jump to the address that was saved in the catch-block-pc word and turn on the control-register.cleanup-in-progress bit.

When the next instruction after **catch-close** is reached, the value of SP is the same as it was before **catch-close**. The catch block is still in the stack, but is no longer linked into the catch-block pointer list. See the section "Catch Blocks," page 192.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

3.2.11 Lexical Variable Accessors

push-lexical-var-n (eight instructions), **pop-lexical-var-n** (eight instructions), **movem-lexical-var-n** (eight instructions).

The three instructions described in this section allow the first eight lexical variables in a lexical environment to be accessed.

push-lexical-var-n *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg **dtp-list**
or **dtp-locative**

Opcodes 20-27

Immediate Argument Type Signed

Description
Pushes on the stack the lexical variable of environment *arg* indexed by *n*. *arg* must be a cdr-coded lexical environment, but this is not checked. *n* is a number between 0 and 7 that is stored in the bottom three bits of the opcode.

Exception None

Memory Reference Data-read

Register Effects TOS: Valid after

pop-lexical-var-n *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
arg2 **dtp-list**
or **dtp-locative**

Opcodes 240-247

Immediate Argument Type Signed

Description
Pops *arg1* off the stack and stores the result into the lexical variable of environment *arg2* indexed by *n*. *arg2* must be a cdr-coded lexical environment, but this is not checked. *n* is a number between 0 and 7 that is stored in the bottom three bits of the opcode. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Exception None

Memory Reference Data-write

Register Effects TOS: Valid before, invalid after

movem-lexical-var-n *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any data type
arg2 **dtp-list**
or **dtp-locative**

Opcodes 250-257

Immediate Argument Type Signed

Description
Stores *arg1*, without popping, into the lexical variable of environment *arg2* indexed by *n*. *arg2* must be a cdr-coded lexical environment, but this is not checked.*n* is a number between 0 and 7 that is stored in the bottom three bits of the opcode. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Exception None

Memory Reference Data-write

Register Effects TOS: Valid before, valid after

3.2.12 Instance Variable Accessors

push-instance-variable, pop-instance-variable, movem-instance-variable, push-address-instance-variable, push-instance-variable-ordered, pop-instance-variable-ordered, movem-instance-variable-ordered, push-address-instance-variable-ordered, %instance-ref, %instance-set, %instance-loc

3.2.12.1 Mapped Accesses to Self

The next four instructions are called within methods or **defun-in-flavors**. Each of these instructions is an access to **self**, mapped.

With the instance in FP|3 and the mapping table in FP|2, the instruction uses the immediate argument, *I*, as the index into the mapping table to get the offset to an instance variable. The type of the value in the mapping table must be **dtp-fixnum**; reference to a deleted variable results in **nil** being found in the mapping table, which causes an error trap.

These instructions check that the argument *I* is within the bounds of the mapping table. If it is not, an error trap occurs. The bounds check is performed by fetching the array header of the mapping table, assuming it is a short-prefix array, and comparing *I* against the array-short-length field. These instructions do check that the data type of the mapping table (FP|2) is **dtp-array**, but do not check to make sure that the mapping table is a short-prefix array, though this is required for correct operation.

Each of these instructions checks the offset to insure that it is a fixnum, but does not check whether it is within bounds. Note that this check is of the element of the mapping table, not of the index into the mapping table. This type of instruction does not check to make sure that the mapping table is a short-prefix array, though this is required for correct operation. That is, the instruction checks that the data type of the mapping table (FP|2) is **dtp-array** and then proceeds with the assumption that the array is a non-forwarded, short-prefix array.

Each of these instructions checks the offset obtained from the mapping table to insure that it is a fixnum. They do not check whether the offset is within bounds of the instance; the flavor system software guarantees that all offsets are within bounds.

These instructions use the following forwarding procedures:

If the cdr code of **self** (FP|3) is 1, accesses the location in the instance that is selected by the mapping table.

If the cdr code of **self** (FP|3) is 0, does a structure-offset memory reference to the header of the instance to check forwarding. If there is no forwarding pointer, sets the cdr code of FP|3 to 1 and proceeds. Otherwise, uses the forwarded address in place of FP|3 (does not change FP|3).

3.2.12 Instance Variable Accessors

push-instance-variable, pop-instance-variable, movem-instance-variable, push-address-instance-variable, push-instance-variable-ordered, pop-instance-variable-ordered, movem-instance-variable-ordered, push-address-instance-variable-ordered, %instance-ref, %instance-set, %instance-loc

3.2.12.1 Mapped Accesses to Self

The next four instructions are called within methods or **defun-in-flavors**. Each of these instructions is an access to **self**, mapped.

With the instance in FP|3 and the mapping table in FP|2, the instruction uses the immediate argument, *I*, as the index into the mapping table to get the offset to an instance variable. Reference to a deleted variable results in **nil** being found in the mapping table, which causes an error trap; the type of the value in the mapping table must be **dtp-fixnum**.

Each of these instructions checks the offset to insure that it is a fixnum, but does not check whether it is within bounds. Note that this check is of the element of the mapping table, not of the index into the mapping table. This type of instruction does not check to make sure that the mapping table is a short-prefix array, though this is required for correct operation. That is, the instruction checks that the data type of the mapping table (FP|2) is **dtp-array** and then proceeds with the assumption that the array is a non-forwarded, short-prefix array.

These instructions check that the argument *I* is within the bounds of the mapping table. If it is not, a trap occurs. The bounds check is performed by fetching the array header of the mapping table, assuming it is a short-prefix array, and comparing *I* against the array-short-length field. Implementation note: it is useful to cache the array header to avoid making a memory reference to get it every time. For an example of how to do this using two scratchpad locations and one cycle of overhead, see the 3600 microcode.

These instructions use the following forwarding procedures:

If the cdr code of **self** (FP|3) is 1, accesses the location in the instance that is selected by the mapping table.

If the cdr code of **self** (FP|3) is 0, does a structure-offset memory reference to the header of the instance to check forwarding. If there is no forwarding pointer, sets the cdr code of FP|3 to 1 and proceeds. Otherwise, uses the forwarded address in place of FP|3 (does not change FP|3).

push-instance-variable *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 1

Argument(s) 1:
I **dtp-fixnum** (Note that the implicit argument **self** must be an instance data type and the mapping table must be a one-dimensional array.)

Opcode 110

Immediate Argument Type Unsigned

Description
Pushes the instance variable indexed by *I* on the stack. See the section "Mapped Accesses to Self," page 201.

Exception None

Memory Reference Header-read (to header of mapping table), data-read (to mapping table), data-read

Register Effects TOS: Valid after

pop-instance-variable *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 0

Argument(s) 2:
arg1 any Lisp data type
I **dtp-fixnum**
(Note that the implicit argument **self** must be an instance data type and the mapping table must be a one-dimensional array.)

Opcode 320

Immediate Argument Type Unsigned

Description
Pops *arg1* off of the top of stack and stores it into the instance variable. See the section "Mapped Accesses to Self," page 201. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Exception None

Memory Reference Header-read (to header of mapping table), data-read (to mapping table), data-write

Register Effects TOS: Invalid after

movem-instance-variable *I* *Instruction*

Format Operand from stack, immediate

Argument(s) 2:
arg1 any Lisp data type
I **dtp-fixnum**
(Note that the implicit argument **self** must be an instance data type and the mapping table must be a one-dimensional array.)

Value(s) Returned 1

Opcode 321

Immediate Argument Type Unsigned

Description
Stores *arg1*, the contents of the top of stack, into the instance variable indexed by the immediate argument *I*. Does not pop the stack. See the section "Mapped Accesses to Self," page 201. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Exception None

Memory Reference Header-read (to header of mapping table), data-read (to mapping table), data-write

Register Effects TOS: Valid after

push-address-instance-variable *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 1

Argument(s) 1:
I **dtp-fixnum**
(Note that the implicit argument **self** must be an instance data type and the mapping table must be a one-dimensional array.)

Opcode 111

Immediate Argument Type Unsigned

Description
Pushes the address of the instance variable indexed by *I* on the stack. See the section "Mapped Accesses to Self," page 201.

Exception None

Memory Reference Header-read (to header of mapping table), data-read (to mapping table)

Register Effects TOS: Valid after

3.2.12.2 Unmapped Accesses to Self

The next four instructions are called within methods or **defun-in-flavor**. Each of these instructions is an access to **self**, unmapped.

With the instance in FP|3, such an instruction uses the operand-from-stack immediate-mode argument *I* as the offset to an instance variable. These instructions do not check whether the offset is within bounds.

push-instance-variable-ordered *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 1

Argument(s) 1: *Opcode* 322
I **dtp-fixnum** Must not be 0.
(Note that the implicit argument
self must be an instance data type.)

Immediate Argument Type Unsigned

Description
Pushes the variable indexed by *I* on the stack. See the section "Unmapped Accesses to Self," page 205.

Exception None

Memory Reference Data-read

Register Effects TOS: Valid after

pop-instance-variable-ordered *I* *Instruction*

Format Operand from stack, immediate

Value(s) Returned 0

Argument(s) 2:
arg1 any Lisp data type
I *arg2* **dtp-fixnum**, must not be 0
(Note that the implicit argument
self must be an instance data type.)

Opcode 322

Immediate Argument Type Unsigned

Description
Pops *arg1* off the top of stack and stores it into the instance variable indexed by *I*. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged. See the section "Unmapped Accesses to Self," page 205.

Exception None

Memory Reference Data-write

Register Effects TOS: Invalid after

movem-instance-variable-ordered *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 1

Argument(s) 2: *Opcode* 323
arg1 any Lisp data type
arg2 **dtp-fixnum** Must not be 0.
(Note that the implicit argument **self** must be an instance data type.)

Immediate Argument Type Unsigned

Description
Stores *arg1*, the contents of the top of stack, into the instance variable indexed by *I*. Does not pop the stack. Note that only 38 bits are stored: the cdr-code bits of memory are unchanged. See the section "Unmapped Accesses to Self," page 205.

Exception None

Memory Reference Data-write

Register Effects TOS: Valid after

push-address-instance-variable-ordered *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 1

Argument(s) 1:
I **dtp-fixnum**, must not be 0
(Note that the implicit argument
self must be an instance data type.)

Opcode 113

Immediate Argument Type Unsigned

Description
Pushes the address of the instance variable indexed by *I* on the stack. See the section "Unmapped Accesses to Self," page 205.

Exception None

Memory Reference None

Register Effects TOS: Valid after

3.2.12.3 Accesses to Arbitrary Instances

As a side effect of the bounds checking, each of these instructions makes a structure-offset reference to the header of the instance and, if the instance has been forwarded, uses the forwarded address as the base to which *arg2* is added.

%instance-ref *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2: *Opcode* 324
arg1 **dtp-instance, dtp-list-instance,**
dtp-array-instance, or **dtp-string-instance**
arg2 **dtp-fixnum**

Immediate Argument Type Unsigned

Description
Pushes on the stack the instance variable of instance *arg1* at the offset specified by *arg2*. Takes an error pre-trap if *arg2* is greater than or equal to the size field of the flavor, using unsigned comparison. See the section "Accesses to Arbitrary Instances," page 209.

Exception None

Memory Reference Header-read, data-read (to flavor descriptor),
data-read (to instance-variable slot)

Register Effects TOS: Valid before, valid after

%instance-set *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 3: *Opcode* 325
arg1 any Lisp data type;
arg2 **dtp-instance**, **dtp-list-instance**,
dtp-array-instance, or **dtp-string-instance**;
arg3 **dtp-fixnum**

Immediate Argument Type Unsigned

Description
Pops *arg1* off of the stack and stores it into the instance variable of instance *arg2* at the offset specified by *arg3*. Takes an error pre-trap if *arg2* is greater than or equal to the size field of the flavor, using unsigned comparison. See the section "Accesses to Arbitrary Instances," page 209.

does not change the cdr code.

Exception None

Memory Reference Header-read, data-read, data-write

Register Effects TOS: Valid before, invalid after

%instance-loc *Instruction*

Format Operand from stack *Value(s) Returned* 1

Argument(s) 2: *Opcode* 326
arg1 **dtp-instance**, **dtp-list-instance**,
dtp-array-instance, or **dtp-string-instance**;
arg2 **dtp-fixnum**

Immediate Argument Type Unsigned

Description
Pushes on the stack the address of the instance variable of instance *arg1* at the offset specified by *arg2*. Takes an error pre-trap if *arg2* is greater than or equal to the size field of the flavor, using unsigned comparison. See the section "Accesses to Arbitrary Instances," page 209.

Exception None

Memory Reference Header-read, data-read

Register Effects TOS: Valid before, valid after

3.2.13 Subprimitive Instructions

%ephemeralp, **%unsigned-lessp**, **%unsigned-lessp-no-pop**, **%allocate-list-block**, **%allocate-structure-block**, **%pointer-plus**, **%pointer-difference**, **%pointer-increment**, **%read-internal-register**, **%write-internal-register**, **no-op**, **%coprocessor-read**, **%coprocessor-write**, **%memory-read**, **%memory-read-address**, **%memory-write**, **%tag**, **%set-tag**, **store-conditional**, **%p-store-contents**, **%set-cdr-code-n** (two instructions), **%merge-cdr-no-pop**, **%generic-dispatch**, **%message-dispatch**, **%jump**, **%check-preempt-request**, **%halt**

%ephemeralp *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any data type

Opcode 7

Immediate Argument Type Signed

Description
Pushes **t** on the stack if the data type of the argument is a pointer data type and the address lies in ephemeral space (bits <31:27> are 0); otherwise pushes **nil** on the stack.

Exception None

Memory Reference None

Register Effects TOS: Valid after

%unsigned-lessp *Instruction*

%unsigned-lessp-no-pop

Format Operand from stack

Value(s) Returned 1 (2 for no-pop)

Argument(s) 2:
arg1 **dtp-fixnum**
arg2 **dtp-fixnum**

Opcode 331 (335 for no-pop)

Immediate Argument Type Unsigned

Description
Tests if, as 32-bit unsigned numbers, *arg1* < *arg2*, and pushes **t** or **nil** on the stack according to the result. The no-pop version of this instruction leaves the first argument on the stack.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%allocate-list-block *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any type
arg2 **dtp-fixnum**

Opcode 311

Immediate Argument Type Unsigned

Description
Using three internal registers, named *list-cache-area, list-cache-length,* and *list-cache-address,* this instruction:

1. Takes an instruction exception (post trap) unless (**eq** *arg1* list-cache-area).

2. Computes list-cache-length minus *arg2*. Takes an instruction exception if the result is negative. Stores the result into list-cache-length unless an exception is taken.

3. Pops the arguments and pushes the list-cache-address. Writes the list-cache-address into BAR-1 (Block-Address-Register-1). Sets the control-register trap-mode field to (max 1 current-trap-mode) so that there can be no interrupts until storage is initialized.

4. Stores (list-cache-address + *arg2*) into list-cache-address (*arg2* must be latched since the third step may overwrite its original location in the stack).

Example:

```
(defun cons (car cdr)
  (%set-cdr-code-normal car)
  (%set-cdr-code-nil cdr)
  (%make-pointer dtp-list
            (prog1 (%allocate-list-block default-cons-area 2)
                   (%block-1-write car)
                   (%block-1-write cdr))))
```

Exceptions

Conditions: *arg1* is not **eq** to list-cache-area.
arg2 is greater than list-cache-length.
See the section "Revision 0 %Allocate-list-block," page 298.

Arguments: *arg1*, *arg2*
Values: 1
Note: Trap handler must insure that control-register.trap-mode will be at least 1 after it returns.

Memory Reference None

Register Effects TOS: Valid before, valid after
BAR-1 is modified

%allocate-structure-block *Instruction*

Format Operand from stack *Value(s) Returned* 1

Immediate Argument Type Unsigned

Argument(s) 2:
arg1 any type
arg2 **dtp-fixnum**

Opcode 315

Description
Using three internal registers, named *structure-cache-area, structure-cache-length,* and *structure-cache-address,* this instruction:

1. Takes an instruction exception unless (**eq** *arg1* structure-cache-area).

2. Computes structure-cache-length minus *arg2*. Takes an instruction exception if the result is negative. Stores the result into structure-cache-length unless an exception is taken.

3. Pops the arguments and pushes the structure-cache-address. Writes the structure-cache-address into BAR-1 (Block-Address-Register-1). Sets the control-register trap-mode field to (max 1 current-trap-mode) so that there can be no interrupts until storage is initialized.

4. Stores (structure-cache-address + *arg2*) into structure-cache-address (*arg2* must be latched since the third step may overwrite its original location in the stack).

Exception
Conditions: *arg1* is not **eq** to structure-cache-area.
arg2 is greater than structure-cache-length.
See section "Revision 0 %Allocate-structure-block," page 298.

Arguments: *arg1, arg2*
Values: 1
Note: Trap handler must insure that control-register.trap-mode will be at least 1 after it returns.

Memory Reference None

Register Effects TOS: Valid before, valid after

%pointer-plus *Instruction*

Format Operand from stack

Argument(s) 2:
arg1 can be any data type,
but **dtp-locative** is expected;
arg2 any data type, but
dtp-fixnum expected

Value(s) Returned 1

Opcode 230

Immediate Argument Type Signed

Description
Pushes the result of adding *arg2* to the pointer field of *arg1*. The data type of the result is the type of *arg1*.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%pointer-difference *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any data type, but a pointer type is expected;
arg2 any data type, but a pointer type is expected

Opcode 231

Immediate Argument Type Signed

Description
Pushes the result of subtracting the pointer field of *arg2* from the pointer field of *arg1*. The data type of the result is **dtp-fixnum**.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%pointer-increment *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg any data type

Opcode 145

Immediate Argument Type Not applicable

Description
Adds 1 to the pointer field of *arg* and stores the result back into the operand. The data-type and cdr-code fields of the operand are not changed.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

%read-internal-register *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 1:
I 10-bit immediate

Opcode 154

Immediate Argument Type Unsigned

Description
Pushes the contents of the internal register specified by *arg* on top of the stack, with the cdr code set to **cdr-next**. See the section "Internal Registers," page 75.

Exception None

Memory Reference None

Register Effects TOS: Valid after

%write-internal-register *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
I 10-bit immediate

Opcode 155

Immediate Argument Type Unsigned

Description
Pops *arg1* off the top of the stack and writes it into the internal register specified by *I*. See the section "Internal Registers," page 75.

Exception None

Memory Reference None

Register Effects TOS: Invalid after

no-op *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 0

Opcode 56

Immediate Argument Type Not applicable

Description
Does nothing. Used when the implementation requires a delay.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

%coprocessor-read *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 1:
I **dtp-fixnum**

Opcode 156

Description
Reads the coprocessor register specified by the immediate field *I* and pushes the result on the stack, with the cdr code set to **cdr-next**.

Exception None

Memory Reference None

Register Effects TOS: Valid after

%coprocessor-write *I* *Instruction*

Format 10-bit immediate

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
I 10-bit immediate

Opcode 157

Description
Writes *arg1* into the coprocessor register specified by the immediate field *I*.

Exception None

Memory Reference None

Register Effects TOS: Invalid after

%memory-read *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 2:
arg1 any Lisp data type
I 10-bit immediate (mask)

Opcode 116

Immediate Argument Type Not applicable

Description
Reads the memory location addressed by *arg1* and pushes its contents on the stack in accordance with the operation specifiers in the immediate, *I*:

cycle-type <9:6> (4 bits) Select one of the 12 memory-cycle types

fixnum-only <5> (1 bit) If set, the instruction will trap if the memory data type is not **dtp-fixnum**.

set-cdr-next <4> (1 bit) If set, the cdr code of the result is 0; otherwise, the cdr code of the result is the cdr code of memory.

See the section "Types of Memory References," page 85.

Exception None

Memory Reference Controlled by the immediate field.

Register Effects TOS: Valid after

%memory-read-address *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 2: *Opcode* 117
arg1 any Lisp data type
I 10-bit immediate (mask)

Immediate Argument Type Not applicable

Description
Reads the memory location addressed by *arg1*, according to the specified cycle type, and returns the updated argument (the address field is changed to be the final address the access arrives at, while the data-type field remains the same) in accordance with the operation specifiers in the immediate, *I*:

cycle-type <9:6> (4 bits)Select one of the 12 memory-cycle types See the section "Memory References."

fixnum-only <5> (1 bit) If set, the instruction will trap if the memory data type is not **dtp-fixnum**.

set-cdr-next <4> (1 bit) If set, the cdr code of the result is 0; otherwise, the cdr code of the result is the cdr code of memory.

Exception None

Memory Reference Controlled by the immediate field.

Register Effects TOS: Valid after

%tag *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 1:
arg any data type

Opcode 12

Immediate Argument Type Signed

Description
Returns the tag of *arg* as a fixnum.

Exception None

Memory Reference None

Register Effects TOS: Valid after

%set-tag *Instruction*

Format Operand from stack

Value(s) Returned 1

Argument(s) 2:
arg1 any data type
arg2 **dtp-fixnum**

Opcode 327

Immediate Argument Type Unsigned

Description
Sets the 8 tag bits of *arg1* to be the bottom eight bits of *arg2*. This is **%make-pointer,** with the arguments reversed so that immediates can be used.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

store-conditional *Instruction*

Format Operand from stack

Value(s) Returned 1

Immediate Argument Type Signed

Argument(s) 3:
arg1 **dtp-locative**
arg2 any type
arg3 any type

Opcode 233

Description
If the content of the location specified by *arg1* is **eq** to *arg2*, then stores *arg3* into that location and returns **t**; otherwise, leaves the location unchanged and returns **nil**. Note that **store-conditional** does not write to memory when it returns **nil**. The cdr code of the specified location is not changed. Other processes (and other hardware processors, to the extent made possible by the system architecture) are prevented from modifying the location between the read and the write.

Exception None

Memory Reference Data-read, followed by raw-write (using the possibly followed pointer) with interlock

Register Effects TOS: Valid before, invalid after

%p-store-contents *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
arg2 any data type

Opcode 235

Immediate Argument Type Signed

Description
Stores *arg2* into memory location addressed by *arg1*, preserving the cdr code but not following invisible pointers.

Exception None

Memory Reference Raw-read followed by raw-write

Register Effects TOS: Valid before, invalid after

%memory-write *Instruction*

Format Operand-from-stack

Value(s) Returned 0

Argument(s) 2:
arg1 any data type
arg2 any data type

Opcode 234

Immediate Argument Type Signed

Description
Stores *arg2* into the memory location addressed by *arg1,* storing all 40 bits including the cdr code, and not following invisible pointers. This replaces the 3600's **%p-store-cdr-and-contents** and **%p-store-tag-and-pointer** instructions. The second argument is typically constructed with the **%set-tag** instruction; in the I-Machine it is legal to have invisible pointers and special markers in the stack temporarily for this purpose.

Exception None

Memory Reference Raw-write

Register Effects TOS: Valid before, invalid after

%set-cdr-code-n *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 0

Argument(s) 1:
arg any data type

Opcodes 146 (n=1), 147 (n=2)

Description
N, which is part of the opcode, is either 1 or 2. Sets the cdr code field of *arg* to *N*.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

%merge-cdr-no-pop *Instruction*

Format Operand from stack, address-operand mode (immediate and sp-pop addressing modes illegal)

Value(s) Returned 1

Argument(s) 2:
arg1 any data type
arg2 (address operand) any data type

Opcode 342

Description
Sets the cdr-code field of *arg2* to the cdr-code field of *arg1*. *arg1* is not popped off the stack.

Exception None

Memory Reference None

Register Effects TOS: Valid before, valid after

%generic-dispatch *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 0

Opcode 52

Immediate Argument Type Not applicable

Description
This is used in calling a generic function. The details of its operation are completely described in the function-calling chapter. In brief, it performs the following operations:

Makes sure that the number of spread arguments is at least 2, doing a pull-lexpr-args operation if necessary.

Gets the address of the interesting part of the flavor, which specifies the size and address of the handler hash table. Checks whether the data type of FP|3 is one of the instance data types and performs the appropriate operations in any case. See the section "Calling a Generic Function," page 277. Fetches two words from the flavor and performs a handler hash table search using the (usually) generic function in FP|2 as the key. Takes an error trap if the method found is not **dtp-even-pc** or **dtp-odd-pc**. Continues execution at the PC.

Exception None

Memory Reference Several data-reads

Register Effects TOS: Invalid after

%message-dispatch *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0 *Opcode* 53

Immediate Argument Type Not applicable

Description
This is used in sending a message. The details of its operation are completely described in the function-calling chapter. See the section "Sending a Message," page 278. In brief, it performs the following operations:

Makes sure that the number of spread arguments is at least 2. Performs a pull-lexpr-args operation if necessary.

Gets the address of the interesting part of the flavor, which specifies the size and address of the handler hash table. Checks whether the data type of FP|2 is one of the instance data types and performs the appropriate operations in any case.

Fetches two words from the flavor and performs a handler hash table search using the message in FP|3 as the key. Takes an error trap if the method found is not **dtp-even-pc** or **dtp-odd-pc**. Puts the instance (from FP|2) in FP|3 and the parameter in FP|2, then continues execution at the fetched PC.

Exception None

Memory Reference Several data-reads

Register Effects TOS: Invalid after

%jump *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 1:
arg **dtp-even-pc** or **dtp-odd-pc**

Opcode 11

Immediate Argument Type Signed

Description
Causes the processor to start executing macroinstructions at the specified PC. This instruction checks that the data type of *arg* is **dtp-even-pc** or **dtp-odd-pc** and jumps to the address. In addition, if bit 39 of *arg* is 1, this instruction stores bit 38 of that word into control-register.cleanup-in-progress. If bit 39 is 0, it leaves the Control register alone. An unwind-protect cleanup handler terminates with a **%jump** instruction.

Exception None

Memory Reference None

Register Effects TOS: Valid after

%check-preempt-request *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 0

Opcode 54

Immediate Argument Type Not applicable

Description
Performs a check-preempt-request operation, that is, sets the preempt-pending flag if the preempt-request flag is set. This causes a trap at the end of the current instruction if the processor is in emulator mode, or when control returns to emulator mode if the processor is in extra-stack mode. See the section "Preemption," page 291.

Exception None

Memory Reference None

Register Effects TOS: Unchanged

%halt *Instruction*

Format Operand from stack

Value(s) Returned 0

Argument(s) 0

Opcode 377

Immediate Argument Type Not applicable

Description
Always takes an exception.

Exception Always

Memory Reference None

Register Effects TOS: Unchanged

```
L Machine: 438 instructions        I Machine: 210 instructions

  15 list-function                 10 list-function
   8 symbol
```

25	predicate	24	predicate
57	numeric	29	numeric
24	data-movement	10	data-movement
7	field-extraction	8	field-extraction
33	array-operation	10	array-operation
15	branch-and-loop	19	branch-and-loop
6	miscellaneous special-purpose		
75	function-calling	12	function-calling (+8 dtps)
18	binding and function-entry	4	binding
7	catch	2	catch
27	lexical-variable-accessing	24	lexical-variable-accessing
11	instance-variable-accessing	11	instance-variable-accessing
34	subprimitive	27	subprimitive
36	hardware subprimitive		
8	graphics		
26	Prolog		

Note: instructions that are listed as being the same in both architectures are those that have identical names. This does not necessarily imply that those instructions perform exactly the same operations.

List-Function Operations

Instructions common to I and L/G:
car, cdr, rplaca, rplacd, member, assoc
Similar instructions:

```
L/G: getf-internal  I: rgetf
     set-cdr-local     set-to-cdr
```

Only on L: cons, ncons, get, memq, assq, last, length-internal
Only on I: set-to-car, set-to-cdr-push-car

Symbol Operations

Only on L: all 8 symbol instructions --
set, symeval, fsymeval, get-pname, value-cell-location, function-cell-location, property-cell-location, package-cell-location

Predicate Instructions

Instructions common to I and L/G: eq, eql, equal-number, greaterp, lessp, endp, plusp, minusp, zerop, %ephemeralp

Similar instructions:

L/G: not, atom, fixp, numberp, | I:
symbolp, single-float-p, array-p,> type-member-n
cl-listp, double-float-p, floatp |

Only on L: char-equal, char=, boundp, fboundp, location-boundp
Only on I: logtest, %unsigned-lessp

Numeric Operations

Instructions common to I and L/G:
unary-minus, %32-bit-plus, %32-bit-difference, %add-bignum-step, %sub-bignum-step, %multiply-bignum-step, %divide-bignum-step, %lshc-bignum-step, %multiply-double
Similar instructions:

L/G: add-stack, add-local, add-immed	I: add
sub-stack, sub-local, sub-immed	sub
increment-local	increment
decrement-local	decrement
multiply-stack, multiply-immed	multiply
quotient-stack	quotient
ceiling-stack	ceiling
floor-stack	floor
truncate-stack	truncate
round-stack	round
remainder-stack	remainder
rational-quotient-stack	rational-quotient
logand-stack	logand
logior-stack	logior
logxor-stack	logxor
ash-stack	ash
rot-stack	rot
lsh-stack	lsh

Only on L: mod-stack, %numeric-dispatch-index, %convert-single-to-double, %convert-double-to-single, %convert-double-to-fixnum, %convert-fixnum-to-double, %convert-single-to-fixnum, float, %double-floating-compare, %double-floating-add, %double-floating-subtract, %double-floating-multiply, %double-floating-divide, %double-floating-abs, %double-floating-minus, %double-floating-scale, set-float-operating-mode, float-operation-status, set-float-operation-status

Only on I: max, min

Data-Movement Instructions

Instructions common to I and L/G:
`push-n-nils`

Similar instructions:

L/G: push-local, push-immed	I: push
pop-local	pop
movem-local	movem
push-address-local	push-address

Only on L: push-indirect, push-constant, push-nil, push-2-nils, push-t, push-character, push-from-beyond-multiple, push-car-local, push-cdr-local, pop-indirect, pop-n, pop-n-save-1, pop-n-save-m, pop-n-save-multiple, pop-multiple-save-n, pop-multiple-save-multiple, movem-indirect, fixup-tos

Only on I: set-sp-to-address, set-sp-to-address-save-tos, push-address-sp-relative, stack-blt, stack-blt-address

Field-Extraction Instructions

Instructions common to I and L/G:
Similar instructions:

L/G: ldb-immed	I: ldb
dpb-immed	dpb
char-ldb-immed	char-ldb
%p-ldb-immed	%p-ldb
%p-dpb-immed	%p-ldb
%p-tag-ldb-immed	%p-tag-ldb
%p-tag-dpb-immed	%p-tag-dpb

Only on L:
Only on I: char-dpb

Array Operations

Instructions common to I and L/G: setup-1d-array, setup-force-1d-array, array-leader, store-array-leader

Similar instructions:

L/G: ar-1, ar-1-immed, ar-1-local	I: aref-1
as-1, as-1-immed, as-1-local	aset-1
ap-1	aloc-1
fast-aref	fast-aref-1
fast-aset	fast-aset-1

ap-leader	aloc-leader

Only on L: ar-2, as-2, ap-2, setup-1d-array-sequential, setup-force-1d-array-sequential, array-register-event, array-leader-immed, store-array-leader-immed, %1d-aref, %1d-aset, %1d-aloc, array-length, array-active-length, ftn-ar-1, ftn-as-1, ftn-ap-1, ftn-load-array-register, ftn-double-ar-1, ftn-double-ar-1

Branch Instructions

Instructions common to I and L/G: branch

Similar instructions:

L/G: branch-true	I: branch-true-no-pop
branch-false	branch-false-no-pop
branch-true-else-pop	branch-true-and-no-pop
branch-false-else-pop	branch-false-and-no-pop
branch-true-and-pop	branch-true-else-no-pop
branch-false-and-pop	branch-false-else-no-pop

Only on L: branch-eq, branch-not-eq, branch-atom, branch-not-atom, branch-endp, branch-not-endp, long-branch, long-branch-immed

Only on I: branch-true, branch-false, branch-true-and-extra-pop, branch-false-and-extra-pop, branch-true-else-extra-pop, branch-false-else-extra-pop, branch-true-extra-pop, branch-false-extra-pop, (branch-true-no-pop-extra-pop, branch-false-no-pop-extra-pop), loop-decrement-tos, loop-increment-tos-less-than

Miscellaneous Special-Purpose Instructions

Similar instructions:

L/G: error-if-true	I: branch-true (0 offset)
error-if-false	branch-false (0 offset)

Only on L: all 6 special-purpose instructions -- push-microcode-escape-constant, funcall-microcode-escape-constant, instruction, %funcall-in-auxiliary-stack-buffer

Function-Calling Instructions

Instructions common to I and L/G: return-multiple, take-values

Similar instructions:

L/G: return-stack/return-nil	I: return-single

Only on L: call-{0/1/2/3}-{ignore/stack/return/multiple}, call-n-{ignore/stack/return/multiple}, funcall-n-{ignore/stack/return/multiple}, funcall-ni-{ignore/stack/return/multiple}, lexpr-funcall-{ignore/stack/return/multiple}, lexpr-funcall-n-{ignore/stack/return/multiple}, return-n, popj, popj-n, popj-multiple, restart-trapped-call, un-lexpr-funcall, stack-dump, stack-load, %assure-pdl-room

Only on I: dtp-call-compiled-even, dtp-call-compiled-odd, dtp-call-indirect, dtp-call-generic, dtp-call-compiled-even-prefetch, dtp-call-compiled-odd-prefetch, dtp-call-indirect-prefetch, dtp-call-generic-prefetch, start-call, finish-call-n, finish-call-apply-n, finish-call-tos, locate-locals, return-kludge

Binding and Function-Entry Instructions

Instructions common to I and L/G: unbind-n, %restore-binding-stack, take-values

Similar instructions:

L/G: %restore-binding-stack-level	I: %restore-binding-stack
bind-locative	bind-locative-to-value

Only on L: bind-specvar, %save-binding-stack-level, optional-arg-supplied-p, append-multiple-groups, take-arg, require-args, take-keyword-argument, take-n-args, take-n-args-rest, take-rest-arg, take-n-optional-args, take-n-optional-args-rest, take-m-required-n-optional-args, take-m-required-n-optional-args-rest

Only on I: bind-locative, entry-rest-accepted, entry-rest-not-accepted

Catch Instructions

Instructions common to I and L/G: none

Similar instructions:

L/G: catch-open-{ignore/stack/return/multiple}/ unwind-protect-open	I: catch-open
catch-close, catch-close-multiple	catch-close

Lexical Variable Accessors

Instructions common to I and L/G: none
Similar instructions:

L/G:	I:
fetch-freevar-n, fetch-freevar-{0/1/2/3/4/5/6/7}	push-lexical-var-n
%pop-freevar-n, %pop-freevar-{0/1/2/3/4/5/6/7}	pop-lexical-var-n
%movem-freevar-n, %movem-freevar-{0/1/2/3/4/5/6/7}	movem-lexical-var-n

Instance Variable Accessors

Instructions common to I and L/G: all 11 instructions -- push-instance-variable, pop-instance-variable, movem-instance-variable, push-address-instance-variable, push-instance-variable-ordered, pop-instance-variable-ordered, movem-instance-variable-ordered, push-address-instance-variable-ordered, %instance-ref, %instance-set, %instance-loc

Subprimitive Instructions

Instructions common to I and L/G: %allocate-list-block, %allocate-structure-block, %pointer-difference, store-conditional, %p-store-contents, %halt

Similar instructions:

L/G: %set-cdr-code-1, %set-cdr-code-2	I: %set-cdr-code-n
popj	%jump
%check-preempt-pending	%check-preempt-request

Only on L: %frame-consing-done, %allocate-list-transport-block, %allocate-structure-transport-block, %pointer, %make-pointer, %make-pointer-immed, %make-pointer-immed-offset, %p-store-contents-increment-pointer, %p-store-contents-pointer-decrement, %p-store-tag-and-pointer, %p-store-cdr-and-contents, %p-contents-as-locative, %p-contents-increment-pointer, %p-contents-pointer-decrement, %p-structure-offset, %set-preempt-pending, %data-type, %fixnum, %flonum, %stack-group-switch, follow-structure-forwarding, follow-cell-forwarding, %block-store-cdr-and-contents, %block-store-tag-and-pointer, %block-search-eq-internal, %trap-on-instance

Only on I: %unsigned-lessp, %pointer-plus, %pointer-increment, %read-internal-register, %write-internal-register, %coprocessor-read, %coprocessor-write, %memory-read, %memory-read-address, %memory-write, %tag, %set-tag, %merge-cdr-no-pop, %generic-dispatch, %message-dispatch, no-op

Hardware Subprimitives

Instructions common to I and L/G: %ephemeralp

Only on L: 35 remaining hardware subprimitives -- %map-cache-write, %phtc-read, %phtc-write, %phtc-setup, %reference-tag-read, %reference-tag-write, %scan-reference-tags, %gc-tag-read, %gc-tag-write, %scan-gc-tags, %gc-map-write, %meter-on, %meter-off, %block-gc-copy, %block-transport, %scan-for-oldspace, %clear-caches, %physical-address-cache, %scan-for-ephemeral-space, %clear-instruction-cache, %scan-for-ecc-error, %io-read-until-bit-test, %io-read-while-bit-test, %io-read, %io-write, %unsynchronized-device-read, %microsecond-clock, %block-checksum-copy, %block-32-36-checksum-copy, %block-36-32-checksum-copy, %audio-start, %fep-doorbell, %disk-start, %net-wakeup, %tape-wakeup

Graphics Instructions

Instructions common to I and L/G: none

Only on L: all the graphics instructions -- %bitblt-short-row, %bitblt-long-row, %bitblt-long-row-backwards, %bitblt-decode-arrays, %draw-line-loop, %draw-string-step, %draw-triangle-segment, %bitblt-short, %bitblt-long, %draw-string-loop, soft-matte-decode-arrays, soft-matte-internal

Prolog Instructions

Instructions common to I and L/G: none

Only on L: all 26 Prolog instructions -- proceed, assure-prolog-frame-room, push-choice-pointer, cut, neck-cut, fail, fail-if-false, fail-if-true, push-goal, execute-goal, execute-stack, dereference-local, dereference-stack, globalize-var, globalize-var-for-neck-cut, push-var, push-void, push-list, push-list*, unify-nil, unify-constant, unify-immediate, unify-local, unify-list, unify-list*, unify-list*-1

4. Function Calling, Message Passing, Stack-Group Switching

****
This file is confidential. Don't show it to anybody, don't hand it out to people, don't give it to customers, don't hardcopy and leave it lying around, don't talk about it on airplanes, don't use it as sales material, don't give it as background to TSSEs, don't show it off as an example of our (erodable) technical lead, and don't let our competition, potential competition, or even friends learn all about it. Yes, this means you. This notice is to be replaced by the real notice when someone defines what the real notice is.
***]

4.1 Stacks

The architecture defines three stacks:

- control stack,
- binding stack, and
- data stack.

Each type of stack is described in the sections that follow. All the stacks grow in the direction of increasing memory addresses. A stack pointer addresses the top word on a stack. A stack limit is the address of the highest location that can be used. A stack base register addresses the lowest entry in the stack.

4.1.1 Control Stack

The control stack holds control information necessary on a per function invocation basis. It also holds the arguments and local and temporary variables of a function.

4.1.1.1 Control Stack Frames

The environment of an executing function is stored in a frame on the control stack. A control stack frame consists of a two-word header, the arguments, and then the local variables and temporaries. Note that there are no separate copies of the arguments for caller and callee; in this respect the I Machine architecture is like the LM-2 and unlike the 3600.

See Figure 22.

The first word in a control stack frame header contains a saved copy of the caller's Continuation register. This is either the caller's caller's PC or the address of a function the caller is going to call later. The second word in a frame header contains a saved copy of the caller's Control register.

When a function returns, the saved values are restored into the Continuation and Control registers. At the same time, the caller's PC is restored from the previous contents of the Continuation register. When a function is first entered, the contents of the Continuation register normally points at the next instruction after a finish-call instruction, except in a trap handler, where it points either at the instruction that trapped or at the following instruction, depending on the type of trap.

Note that the Continuation and Control registers stored in a frame header belong to the caller's frame, not to the frame where they are stored. The values for the current frame are kept in live (hardware) registers instead of the stack because special hardware uses them.

The maximum size of a control stack frame is

```
(- stack-cache-size
                  2   ;For trap-out stack frame
                  2   ;For pushing the vector and PC
                  3   ;Increment in PHT-SEARCH code
                  2   ;Used in PHT-SEARCH code
                  ))
```

4.1.1.2 Base Registers

There are three base registers that point to the current control stack frame. These can be used to calculate instruction operand addresses. See the section "Macroinstruction Set."

The frame pointer (FP) points to the first word of the frame header. This register is used to locate the function's arguments, which start at a fixed offset past FP. The local pointer (LP) points after the spread arguments. (Spread arguments are arguments that are not part of a **&rest** parameter.) It is used to locate local variables to the function. The stack pointer (SP) points to the highest word in the frame. SP is incremented or decremented as execution proceeds and pushes or pops the stack. These registers are discussed further in another section. See the section "Registers Important to Function Calling and Returning."

See the section "Revision 0 Implementation Function-Calling Features."

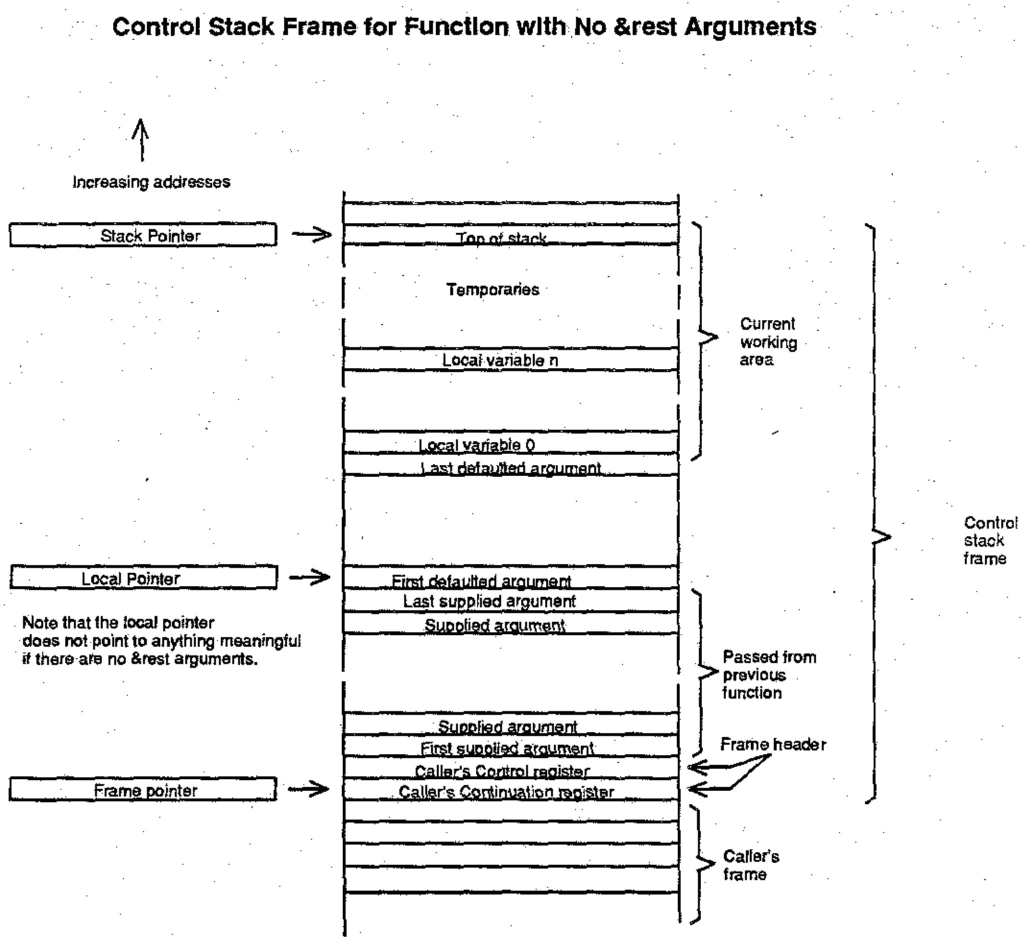


Figure 22. An I-machine control stack frame.

4.1.2 Binding Stack

Binding is the temporary replacement of a memory cell's contents. The Binding Stack saves the address and contents of memory cells that have been bound so the original contents can later be restored. Note that binding affects only the contents of a cell, not its cdr code.

Entries on the binding stack are two words long. The fields of an entry are as follows:

Word	*Position*	*Field*	*Comments*
0	<39>	--	Must be zero.
0	<38>	Binding-stack-chain-bit	=1 if the previous entry is
0			for the same frame.
0	<37:0>	Binding-stack-cell	Locative to the memory cell
0			that is bound.
1	<39:38>	--	Don't care. (Stack-group
0			switch may alter them.)
1	<37:0>	Binding-stack-contents	Saved contents of bound cell.

The binding-stack-cell field contains a **dtp-locative** pointer to the memory cell that is bound. This indicates which location has had its contents temporarily replaced. In the case of a *dynamic closure*, however, a new memory cell is created, and the old value cell is loaded with a **dtp-external-value-cell-pointer** to this new cell. The new cell is referenced by the closure.

The binding-stack-contents field contains the contents of the bound cell. Bindings do not persist across stack groups, and must be undone when control is transferred to another group. Binding-stack-contents contains the "former" contents of the cell when the binding stack belongs to the currently executing stack group; otherwise it contains the "current" contents of the cell. See the section "Stack-Group Switching."

The binding-stack-chain-bit is 1 if the previous entry on the binding stack is associated with the same function invocation as this entry. This bit is set by the **bind** instruction, and groups entries on the binding stack into frames associated with a function. Binding stack frames are removed at function return time.

The Binding Stack Pointer points to the top of the binding stack (word 1 of the topmost entry]) There is also a Binding Stack Limit register.

Bindings are performed by the **bind-locative** or **bind-locative-to-value** instruction. A bind instruction checks the Control register binding cleanup bit. If this bit is 0, then this binding is the first associated with the current frame. The instruction will set the binding cleanup bit in the Control register, and set the chain bit for the entry on the binding stack to 0. If the cleanup bit is 1, then there are already bindings associated with the current frame. The instruction will set the chain bit for the entry to 1.

Note that an unbind instruction (**unbind-n** or **%restore-binding-stack**) will clear the Control register cleanup bit if it removes an entry from the binding stack with the chain bit 0.

4.1.3 Data Stack

The purpose of the data stack is to provide an allocation area for temporary data whose lifetime is associated with a function's lifetime. This allows less expensive allocation/deallocation than the general mechanism.

This is implemented in software in the same manner as on the 3600.

4.2 Registers Important to Function Calling and Returning

The following processor registers are relevant to function calling and returning:

Program Counter (PC)
Address of the current instruction.
dtp-even-pc or **dtp-odd-pc**

Frame Pointer (FP)
Address of the current stack frame.
dtp-locative

Local Pointer (LP)
Address of the local-variable part of the current stack frame.
dtp-locative

Stack Pointer (SP)
Address of the highest in-use word in the stack.
dtp-locative

Continuation register (CONT)
Address of the first instruction to be executed after the next function call or return.
dtp-even-pc or **dtp-odd-pc**

Control register (CR)
A bunch of bits and fields to be described below.
dtp-fixnum

The program counter contains the address of the current instruction.

The frame pointer points to the first word of the control stack frame header. This register is used to locate the function's arguments, which start at a fixed offset (2) past FP. It can also be used to locate the function's locals if the function does not accept a **&rest** argument. When a function returns, the SP is set to FP-1 to remove the function's frame.

After a finish-call instruction, the local pointer points to the word after the spread arguments. Thus it points to the rest argument if there is one; otherwise it points to the first local variable. When there are optional arguments and no rest argument, LP points at the first optional argument not supplied by the caller, if there is one.

LP is used to locate local variables of the functions. FP cannot always be used for this since in general the number of arguments the function accepts is variable. LP may be adjusted by the **entry** and **locate-locals** instructions.

The stack pointer points to the highest word in the control stack. SP is incremented or decremented as execution proceeds and pushes or pops the stack.

The Continuation register contains the address of the instruction to be executed after the next finish-call or return instruction. Whether this is the return address in the caller, or the first instruction in a function about to be called, depends on context. It is the address of the function to call between the start-call and finish-call instructions, and the return address in the caller between the finish-call and return instructions.

The Control register contains a fixnum with several packed fields:

Position	*Size*	*Name*
<31:30>	2 bits	Trap-mode
<29>	1 bit	Instruction-trace
<28>	1 bit	Call-trace
<27>	1 bit	Trace-pending
<26:24>	3 bits	Cleanup-bits
<26>		cleanup-catch
<25>		cleanup-bindings
<24>		trap-on-exit
<23>	1 bit	Cleanup-in-progress
<22>	1 bit	Call-started
<21:20>	2 bits	Reserved
<19:18>	2 bits	Value-disposition
<17>	1 bit	Apply
<16:9>	8 bits	Frame-size-of-caller
<8>	1 bit	Extra-argument
<7:0>	8 bits	Arg-size

Trap-mode controls the handling of exception traps. The four modes, explained elsewhere (See the section "Trap Modes."), are:

0	Emulator
1	Extra Stack
2	High-Speed I/O
3	FEP

The trap-mode field is adjusted when a trap is taken. It is set to (max 1 current-trap-mode) by the **%allocate-list-block** or **%allocate-structure-block** instruction.

Instruction-trace when 1 at the beginning of an instruction, causes completion of the instruction to set trace-pending and causes a trap before the next instruction executes. If a post-trap occurs when instruction-trace is 1, trace-pending is set in the control register saved as part of taking the trap. This is not true of a pre-trap. If a return instruction restores a control register value with the instruction-trace bit set, the instruction returned to is executed before the trap occurs.

Call-trace when 1, causes the finish-call instructions to set trace-pending, which causes a trap before the first instruction of the called function executes. If stack overflow occurs simultaneously, trace-pending is set in the saved control register in the frame header of the stack overflow trap handler's frame. When the stack overflow handler returns, the trace trap occurs. Call-trace does not affect the implicit finish-call performed when a trap occurs, because call-trace gets cleared first. See the section "Revision 0 Implementation Function-Calling Features."

Trace-pending when 1, causes a trap to occur before the next instruction executes. Note that a sequence break can intervene before the trap actually goes off. There is only one trap vector location for trace-pending, regardless of the semantic significance of the trap to the software. See the section "Revision 0 Implementation Function-Calling Features." The interaction of trace-pending with the repeated returns caused by Value-disposition Return is not architecturally defined. See the section "Trace Traps."

Cleanup-bits specifies what actions need to be performed prior to removing the function's frame from the control stack. The actions are normally performed by a return instruction. In the case of abnormal termination, these actions are performed by the **throw** function (which uses a return instruction internally). All three bits are cleared by a finish-call instruction. The bits are:

Cleanup-catch — This bit indicates there are catch/unwind-protect blocks in the frame. The catch cleanup bit is set whenever a catch or unwind-protect block is created. The bit is cleared when the outermost catch/unwind-protect block in a frame is destroyed. See the section "Catch Instructions."

Cleanup-bindings — This bit indicates there is a non-empty binding-stack frame associated with this control-stack frame, in other words that this function has bound some special variables. This bit is set by the binding instructions (**bind-locative** and **bind-locative-to-value**) and can be cleared by the unbinding instructions (**unbind-n** and **%restore-binding-stack**). See the section "Binding Instructions."

Trap-on-exit — This bit causes a trap to software when the frame is exited. Used for bottom frame in stack, debugger c-X E command, phantom stacks, metering, and so forth. The software can use the cdr-code bits of the two header words in the frame, which are initially set to 11 by the hardware, to distinguish these cases. The trap-on-exit bit is set and cleared only by software, and only in copies of the Control register saved in memory, not in the live register.

For details: See the section "Frame Cleanup."

Cleanup-in-progress is set by an unwind-protect cleanup handler in accordance with the contents of the catch-block-previous word in the catch block to indicate that execution is occurring inside of an unwind-protect handler.

Call-started is set by start-call instructions and cleared by the finish-call instructions.

Reserved bits not allocated yet.

Value-disposition specifies what the caller wants done with the result(s) produced by the function. It is set by the finish-call instructions. The interpretation of value-disposition is:

0 Effect — The function has been called for effect. Discard any values the function may produce.

1 Value — Only a single value is desired by the caller. Push this on the control stack, discarding any extra values.

2 Return — The value(s) returned by the function are also the value(s) returned by the caller. Pass the value(s) along to this frame's caller.

3 Multiple — The caller wants multiple values returned. Push any number of values on the stack, followed by a fixnum specifying the number of values.

The requested disposition is performed by a return instruction. Returned results are pushed onto the stack after the function's frame has been removed from the stack. If a function terminates abnormally, it does not return a value so Value-disposition is ignored.

Apply, if 1, indicates that ~~a rest~~ an apply argument ~~list~~ has been supplied following the spread arguments and is stored in LP|0. This bit is set by the finish-call instructions, and is used to implement the Common Lisp **apply** function. This can be reset by the **entry** instruction doing a pull-apply-args operation.

Frame-size-of-caller contains the size of the caller's stack frame (callee's FP minus caller's FP). It is used by return instructions to locate the start of the caller's frame when the function returns. This field is set by the finish-call instructions.

Extra-argument is set to 1 to indicate an extra argument has been supplied to the function by a start-call instruction. This happens when calling a lexical closure, a generic function, an instance, or any interpreted function or illegal data type. See the section "Starting a Function Call." This bit is just used to transmit information from a start-call instruction to the corresponding finish-call instruction and then is no longer needed. It is cleared by a finish-call instruction.

Arg-size is the offset of LP from FP in the frame. It is used to restore the LP when the function resumes execution after calling another function. It is also used by the **entry** instruction to determine how many explicit arguments were supplied with the call. This field is set by the finish-call instructions (for the new frame). It is also adjusted by the **locate-locals** instruction.

4.3 Function Calling

A function call requires three different actions: specifying the function to call, pushing the arguments to the function, and finishing the call by building the new stack frame and entering the target function. The instructions that accomplish these actions are described below.

4.3.1 Starting a Function Call

A function call is begun by executing one of the start-call types of instructions, whose single argument is the function to be called. These instructions create the header of the callee's stack frame, possibly push an extra argument onto the stack, and set the continuation according to the type of function being called.

The most general start-call instruction, **start-call** itself, takes its argument from the top of stack or from a local variable. Several full-word instructions are also supplied; these contain an address that specifies the function and possibly its data type. In summary:

start-call Takes a general stack operand.

dtp-call-compiled-even and **dtp-call-compiled-odd**
Address a compiled-function directly, specifying whether to start

with the even or odd halfword instruction in the addressed location.

dtp-call-indirect Addresses a function cell and fetches its contents.

dtp-call-generic Addresses a generic function directly.

Each full-word start-call type of instruction comes in prefetching and nonprefetching versions. Semantically these are identical, but the prefetching version is a hint to the hardware that a finish-call instruction appears soon enough after the start-call instruction that it would be worthwhile to prefetch the first few instructions of the called function rather than continuing to fetch ahead instructions from the calling function. The decision of when to use the prefetching version is up to the compiler; it is probably appropriate when there are no nested function calls in the arguments and the number of instructions in the arguments is less than a certain constant (around half a dozen). Prefetching makes the ensuing finish-call operation run faster. The hardware does not necessarily actually prefetch when the prefetching version is executed; it depends on the particular instruction, on the data type of the function, and on how complex the hardware turns out to be. The prefetching versions of the indirect and generic calls are almost certainly not treated any differently from the normal versions by the hardware: they exist entirely for software reasons.

The start-call instructions push the Continuation and Control registers (in that order) onto the control stack with their cdr codes both set to 3; they will become the header of the callee's control stack frame. After the Control register is pushed, the control-register.call-started bit is set to 1.

Depending on the data type of the function being called, a start-call instruction may push a third word which is called the "extra argument." Its cdr code is set to 0. All data types other than **dtp-compiled-function** receive an extra argument. In the case of instance or generic function, the word pushed on the stack is just a placeholder for the real extra argument the function will be called with, since this cannot be computed until the first argument is known.[1] If the start-call instructions push an extra argument, they set the extra-argument bit in the Control register to 1; otherwise they clear the bit to 0. This information is saved for the finish-call instruction. The setting or clearing of this bit takes place after the Control register is saved on the stack.

After Continuation and Control registers are saved, the Continuation register is

[1] This extra argument mechanism is necessary because in general the data type of the function being called is not known until run time. Note that if a method is called directly, as from a combined method, or if a lexically internal function is called directly, as from its parent, the extra argument is passed instead as a normal argument. Any given function always receives its arguments in the same format, and does not need to know whether the first argument was supplied normally by the caller or was an "extra" argument.

set to a PC value pointing at the beginning of the function to be called (the argument of the start-call). Depending on the data type of the function, this continuation can be computed from the function itself or can be fetched from one of 64 trap-vector locations, indexed by the data-type of the function. The effect of the function's data type on a start-call is as follows:

compiled-function There is no extra argument. The continuation is set to **dtp-even-pc** with the address of the function.

symbol Fetch the contents of the symbol's function cell and try again. Take an error trap if the function cell contains **dtp-null.**

instance Push the instance as the extra argument. The continuation comes from the trap vector.

generic Push the generic function as the extra argument. The continuation comes from the trap vector.

lexical closure Fetch the enclosed function and the environment from memory. If the enclosed function is compiled, push the environment as the extra argument and set the continuation to **dtp-even-pc** and the function's address, producing a call to the enclosed function with the environment as its extra argument. If the enclosed function is not compiled, then push the lexical closure as the extra argument and take the continuation from the trap vector location for **dtp-lexical-closure.**

anything else Push the original function as the extra argument. Use the data type of the function as an index into the trap vector to fetch the appropriate interpreter function and set the continuation to that.

After a start-call instruction the continuation is guaranteed to be a PC pointing into a compiled function, assuming the trap-vector has been initialized correctly.

For the instance and generic function cases, the real function (the method) and the real extra argument (the mapping table) cannot be computed until the value of the first argument is known, so these have to be deferred until a finish-call instruction is executed and execution proceeds at the PC now in the Continuation Register.

Note that after doing a start-call, a program does not know the exact depth of the stack, because it does not know whether an extra argument was pushed. The compiler avoids using SP-relative addressing to access variables deeper in the stack than the incipient frame header.

Figure 23 shows how the stack looks at this point.

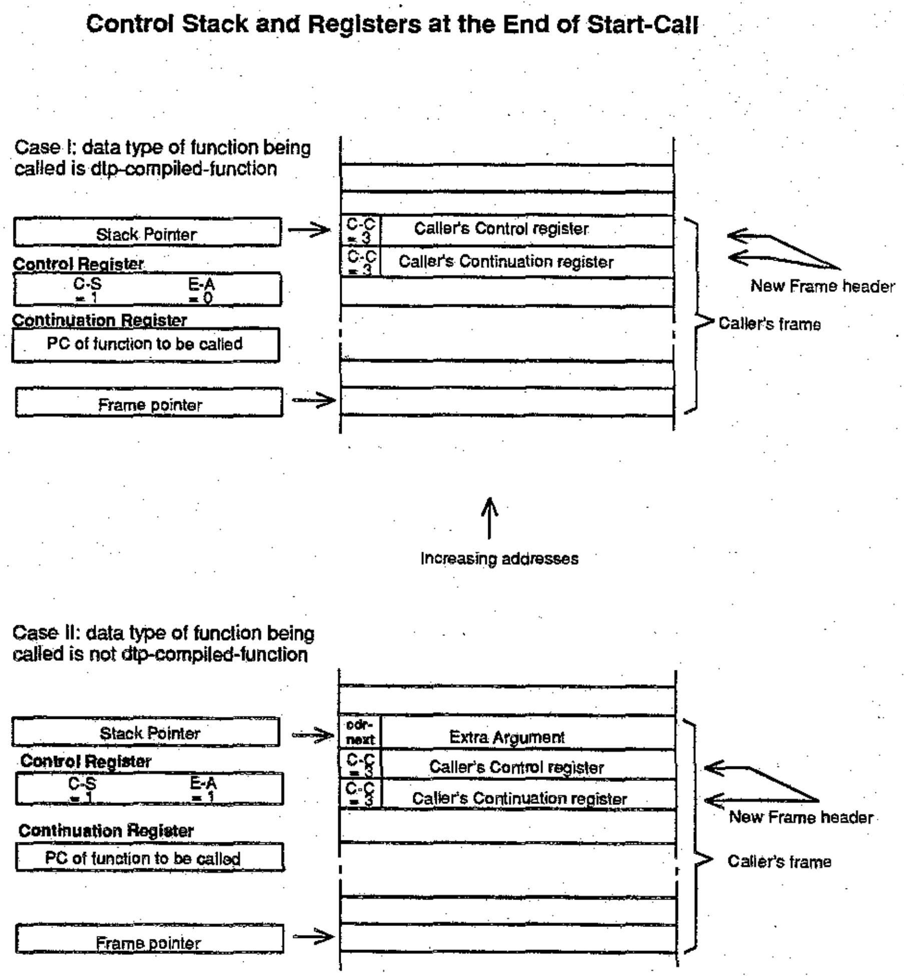

Control Stack and Registers at the End of Start-Call
Case I: data type of function being called is dtp-compiled-function
Stack Pointer
Control Register
C-S = 1
E-A = 0
Continuation Register
PC of function to be called
Frame pointer
C-C = 3
Caller's Control register
C-C = 3
Caller's Continuation register
New Frame header
Caller's frame
Increasing addresses
Case II: data type of function being called is not dtp-compiled-function
Stack Pointer
Control Register
C-S = 1
E-A = 1
Continuation Register
PC of function to be called
Frame pointer
cdr-next
Extra Argument
C-C = 3
Caller's Control register
C-C = 3
Caller's Continuation register
New Frame header
Caller's frame

Figure 23. The stack at the end of a start-call instruction

4.3.2 Pushing the Arguments

After starting a function call, the caller computes the arguments and pushes them onto the stack, in order. Results of instructions normally are **cdr-next**, to facilitate the linking of the arguments into a list to be ~~passed to~~ received by an **&rest** argument. The resetting of the final cdr code is performed by the **entry** instruction.

4.3.3 Finishing the Call

After starting a function call and pushing the arguments, the caller executes a finish-call instruction. This instruction builds the new stack frame, checks for control stack overflow, and enters the callee at the appropriate starting instruction.

Instructions at the beginning of the callee are in charge of checking the number of arguments and rearranging them to suit its needs, or signalling an error if the wrong number of arguments were supplied. Every compiled function should contain code to do this, but the linker (which places **dtp-call-compiled-even** or **dtp-call-compiled-odd** instructions into compiled callers) can optimize calls by bypassing those instructions and arranging for the called function to be entered directly at the right place.

There are two finish-call instructions, **finish-call-n**, and **finish-call-tos** which differ only in how they obtain their argument. **finish-call-n** takes its argument as an 8-bit field of a 10-bit immediate, and **finish-call-tos** pops its argument from the top of stack.

The operand, called *N-Args*, indicates the number of arguments explicitly supplied with the call, including the apply argument, if present. It does not include the extra-argument, if any. The finish-call-n instructions include an extra bias of +1 in the immediate argument count, to simplify the hardware. This bias is not in the operand to the finish-call-tos instructions.

There are a number of applications for calling a function with the number of arguments not known at compile time, where the arguments do not come from a list, including the **%finish-function-call** and **multiple-value-call** special forms and things built on them. These are handled by using the **finish-call-tos** instruction.

Three additional bits supplied with the instruction, I<9:8> of the 10-bit immediate field and one bit of the opcode, are used as follows.

Value-disposition A 2-bit field taken from the operand field that specifies what to do with the result(s) produced by the function being called:

0 Effect The function is being called for effect. Discard any values it may produce.

1 Value — Only a single return value is desired. Discard any additional values the function may produce.

2 Return — The value(s) returned by the function being called are also the value(s) returned by this function. Pass the value(s) along to this frame's caller. This is illegal in nested calls.

3 Multiple — Multiple values are desired. These should be returned along with a fixnum specifying the number of values returned.

Apply — A 1-bit field taken from the opcode, which is a 1 if the top word in the stack is a list of arguments. The list may be spread or packed by the entry instruction. This implements the Common Lisp **apply** function.

The operations of finish-call are described sequentially below, although in the actual hardware many of them happen in parallel.

The finish-call instruction next builds the new stack frame with the following procedure:

```
FP                For finish-call-n, finish-call-n-apply:
              <=  SP - N-Args - control-register.extra-argument

FP                For finish-call-tos, finish-call-tos-apply:
              <=  SP - N-Args - control-register.extra-argument - 1

LP            <=  SP + 1 - Apply           ;this could be past SP

Continuation  <=  the address of the next instruction after the
                  finish-call.

SP, Binding-stack-pointer, and Data-stack-pointer are unchanged.

Save the old contents of Continuation temporarily (see below).
```

The control register is adjusted as follows:

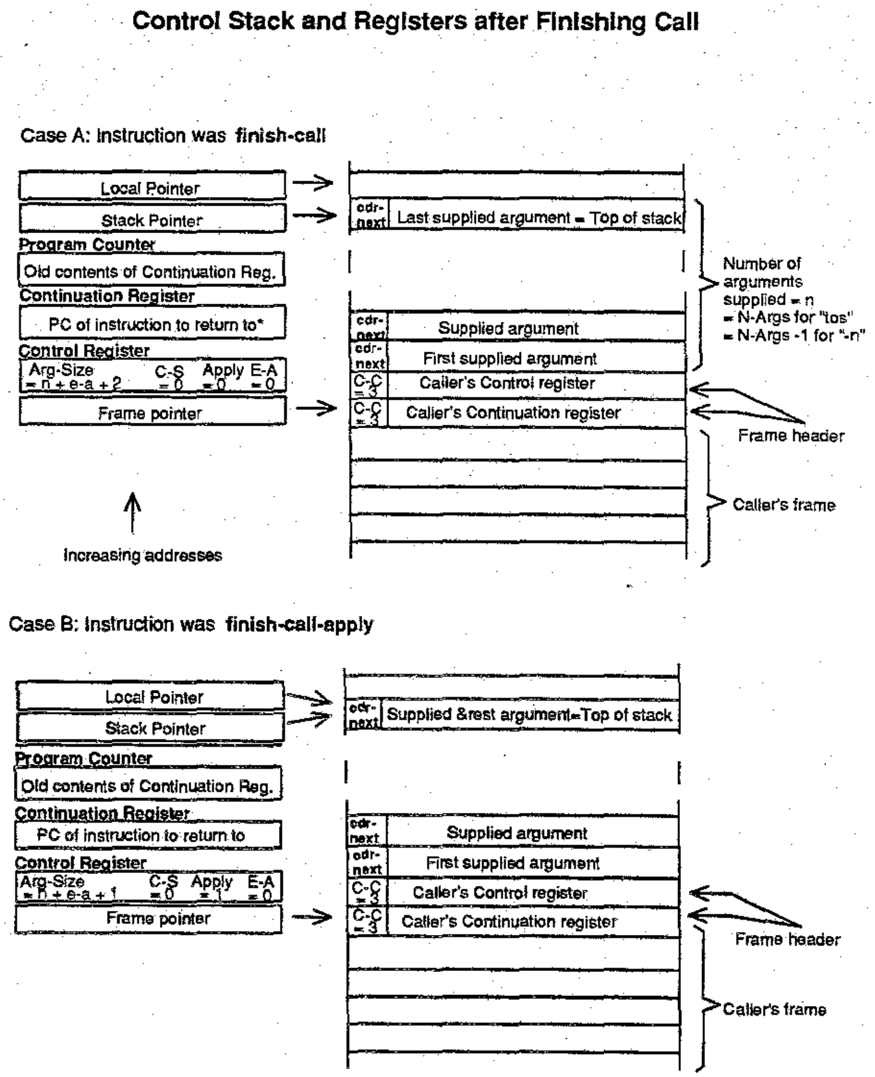
Arg-Size	<=	(new LP minus new FP) For **finish-call-n**, **finish-call-n-apply**: <= N-Args + control-register.extra-argument - apply + 1 For **finish-call-tos**, **finish-call-tos-apply**: <= N-Args + control-register.extra-argument - apply + 2
Apply	<=	Apply bit in the instruction
Value-Disposition	<=	Value Disposition bits in the instruction
Cleanup-Bits	<=	0
~~Instruction-State~~	~~<=~~	~~0~~
Trap-Mode	<=	*unchanged*
Extra-Argument	<=	0 ;actually this doesn't matter
Frame-size-of-caller	<=	new FP minus old FP
Call-started	<=	0

After building the new frame, finish-call checks for control stack overflow by calculating whether SP is greater than the stack limit. If the stack overflows in normal mode, a stack-overflow trap will be taken after the end of the finish-call instruction before executing the first instruction of the target function. If the stack overflows in extra stack mode, the machine halts with a fatal error. See the section "Processor Faults."

The finish-call instruction ORs control-register.call-trace into control-register.trace-pending, forcing a trace pre-trap upon execution of the next instruction if call-trace was 1.

Finally execution proceeds with the instruction at the halfword address specified in the Continuation register before it was set to the return address.

Figure 24 shows how the stack looks after completion of the finish-call instruction.


Control Stack and Registers after Finishing Call
Case A: Instruction was finish-call
Local Pointer
Stack Pointer
Program Counter
Old contents of Continuation Reg.
Continuation Register
PC of instruction to return to*
Control Register
Arg-Size = n + e-a + 2
C-S = 0
Apply = 0
E-A = 0
Frame pointer
cdr-next
Last supplied argument = Top of stack
cdr-next
Supplied argument
cdr-next
First supplied argument
C-C = 3
Caller's Control register
C-C = 3
Caller's Continuation register
Number of arguments supplied = n = N-Args for "tos" = N-Args -1 for "-n"
Frame header
Caller's frame
Increasing addresses
Case B: Instruction was finish-call-apply
Local Pointer
Stack Pointer
Program Counter
Old contents of Continuation Reg.
Continuation Register
PC of instruction to return to
Control Register
Arg-Size = n + e-a + 1
C-S = 0
Apply = 1
E-A = 0
Frame pointer
cdr-next
Supplied &rest argument=Top of stack
cdr-next
Supplied argument
cdr-next
First supplied argument
C-C = 3
Caller's Control register
C-C = 3
Caller's Continuation register
Frame header
Caller's frame

* This is the address of the next instruction after finish-call.

Note: e-a is the original value of control-register.extra-argument.

Figure 24. The stack after completion of the finish-call instruction

4.3.3.1 Trapping Out of Finish-call and Restarting

Traps in the finish-call instructions always occur after building the new frame and setting the Control register, the Continuation register, and the Program Counter to their new values. Thus any trap occurring in a finish-call instruction looks like a pre-trap in the first instruction of the called function. No special action is required to restart after such a trap.

4.3.3.2 Aborting Calls

It is sometimes necessary to abort a call that has been started, instead of finishing it with a finish-call instruction. Aborting a call consists of popping the stack back to the level before the call was started and restoring some of the Continuation and Control register values saved by the start-call instruction. This is performed by Lisp code.

4.4 Function Entry

A compiled function starts with a sequence of instructions that are involved in receiving the arguments. The first instruction is known as the entry instruction. It is followed by a possibly-empty sequence of instructions known as the entry vector. The function can be entered at the entry instruction, which will check the number of arguments and select the first instruction to be executed, either an element of the entry vector or the first instruction after the entry vector. Alternatively, this selection can be made by the linker when the number of arguments is known statically, and the function can be entered directly at an element of the entry vector or at the first instruction after the entry vector. In either case, execution proceeds from the selected instruction according to normal instruction sequencing, possibly executing additional instructions from the entry vector. After completing the entry vector, some additional argument-taking instructions may be executed, depending on the particular function. Thus a compiled function consists of:

Object header (2 words)
Entry instruction
Entry-vector instructions
Other argument-taking instructions
Body instructions

See the section "Representation of Compiled Functions."

Each entry-vector element is two half-word instructions long. For each **&optional**, there is an element of the entry vector and there is one for the **&rest** argument, if supplied. (This includes an automatically-generated **&rest** argument in a function with **&key** arguments.) The element of the entry vector corresponding to an argument contains instructions that are executed if that argument is not supplied by the caller. These instructions compute the default value (**nil** for a **&rest** argument) and push it on the stack. If this computation will not fit in an entry-vector element, the compiler inserts a branch to the rest of the code, which ends in a branch back. If the computation is smaller than the size of an entry-vector element, it ends with cdr-code sequencing that skips an instruction.

The entry instruction contains the following information:

Number of required arguments
Number of optional arguments
Number of rest arguments (zero or one)

An entry instruction performs an argument match-up process that either traps (for wrong number of arguments) or adjusts the stack and then branches to the appropriate instruction of the entry vector, or to the instruction after the entry vector. See the section "Entry-rest-accepted." The first entry-vector element follows immediately after the entry instruction. Adjusting the stack is done by performing one of two operations described later: *pull-apply-args* or *push-apply-args*.

The following conditions are computed by an entry instruction:

- Too few spread arguments (*N-Args+2* < *min-args+2*)
- Too many spread arguments (*N-Args+2* > *max-args+2*)
- Maximum spread arguments (*N-Args+2* = *max-args+2*)
- Rest argument wanted (rest-arg = 1)
- Rest argument supplied (control-register.apply = 1)

Note that the argument comparisons are all biased by plus 2. The value of control-register.arg-size is two greater than the actual number of arguments in the frame because it includes the two frame header words (this makes **return** faster). To simplify these entry comparisons, the arguments *min-args* and *max-args* in the entry instructions are correspondingly biased by two.

If "rest argument wanted" and "rest argument supplied" are both false, this is the simple case. If there are too few or too many arguments, take a Wrong-Number-of-Arguments (WNA) trap. Otherwise, enter the function at entry-vector element (*N-Args* - *min-args*); this skips over the default-initialization instructions for those optional arguments that had values supplied.

If "rest argument wanted" is false and "rest argument supplied" is true, then if there are less than the maximum number of arguments, do a pull-apply-args operation. Otherwise, take a wrong number of arguments trap because there are too many arguments.

The entry instruction contains the following information:

Number of required arguments
Number of optional arguments
Number of rest arguments (zero or one)

An entry instruction performs an argument match-up process that either traps (for wrong number of arguments) or adjusts the stack and then branches to the appropriate instruction of the entry vector, or to the instruction after the entry vector. See the section "Entry-rest-accepted." The first entry-vector element follows immediately after the entry instruction. Adjusting the stack is done by performing one of two operations described later: *pull-apply-args* or *push-apply-args*.

The following conditions are computed by an entry instruction:

- Too few spread arguments (*N-Args+2* < *min-args+2*)
- Too many spread arguments (*N-Args+2* > *max-args+2*)
- Maximum spread arguments (*N-Args+2* = *max-args+2*)
- Rest argument wanted (rest-arg = 1)
- Apply argument supplied (control-register.apply = 1)

Note that the argument comparisons are all biased by plus 2. The value of control-register.arg-size is two greater than the actual number of arguments in the frame because it includes the two frame header words (this makes **return** faster). To simplify these entry comparisons, the arguments *min-args* and *max-args* in the entry instructions are correspondingly biased by two.

- If "rest argument wanted" and "apply argument supplied" are both false, this is the common and simple case.
    - ° If the number of arguments is in range *(min-args <= N-Args <= max-args)* then enter the function at entry-vector element *(N-Args - min-args)*; this skips over the default initialization instructions for those optional arguments that had values supplied.
    - ° Otherwise there are too few or too many arguments; take a Wrong Number of Argument trap.
- If "rest argument wanted" is false and "apply argument supplied" is true, then the apply argument must be converted into spread arguments.
    - ° If there are less than the maximum number of arguments supplied

(N-Args < max-args) then do a pull-apply-args operation to pull *(max-args - N-Args)* arguments, which will normally turn off "apply argument supplied," and retry the argument matchup process.

  - If the maximum number of arguments is supplied *(N-Args = max-args)* and the apply argument is **nil**, pop the apply argument, clear control-register.apply, and enter at vector *(N-Args - min-args)*. Note: this is an optimization of the pull-apply-args operation pulling 0 arguments out of **nil** and retrying the argument matchup process.
  - Otherwise there are too many arguments; take a Wrong Number of Arguments trap.

- If "rest argument wanted" is true and "apply argument supplied" is false, then a rest argument may need to be made from some of the spread arguments.
  - If there are too few arguments *(N-Args < min-args)* then take a Wrong Number of Arguments trap.
  - If the number of spread args is in range *(min-args <= N-Args <= max-args)* then enter at entry-vector element *(N-Args - min-args)*; this skips over the default initialization for those optional arguments that had values supplied and the last element of the entry vector will push **nil** to default rest argument.
  - Otherwise *(N-Args > max-args)* some spread args must be made into the rest argument; do a push-apply-args operation of *(N-Args - max-args)* arguments and enter at entry vector *(max-args - min-args + 1)*.
- If "rest argument wanted" and "apply argument supplied" are both true, then the apply argument may be pushed, pulled or used as is.
  - If less than the maximum spread arguments were supplied *(N-Args < max-args)*, then convert some of the apply argument to spread arguments by doing a pull-apply-args operation to pull *(max-args - N-Args)* arguments, which may turn off "apply argument supplied," and retry the argument matchup process.
  - If exactly the maximum nu.nber of spread arguments was supplied *(N-Args = max-args)* then use the apply argument as the rest argument. Set the cdr-code of the top word of stack to **cdr-nil** and enter the function at entry-vector element *(max-args - min-args + 1)*. This skips over the default initialization for the optional arguments and for the rest argument.

function at entry-vector element *(max-args - min-args + 1)*. This skips over the default initialization for the optional arguments and for the rest argument.

- ° Otherwise more than the maximum number of spread arguments were supplied *(N-Args > max-args)*. Push some of the spread arguments into the apply argument by doing push-apply-args operation of *(N-Args - max-args)* arguments and enter at entry vector *(max-args - min-args + 1)*.

Figure 25 summarizes how the argument matchup operation is performed.

4.4.1 Push-apply-args

The push-apply-args operation is invoked when there are too many spread arguments and a rest argument is wanted. It pushes some spread arguments back into the apply argument, after which the function is started at its all-arguments-supplied entry point. This operation does not involve any memory references nor any possibility of trapping.

In detail, push-apply-args does the following:

- Set the cdr code of the last word in the stack to **cdr-nil.**

- If an apply argument was supplied, set the cdr code of the second to last word in the stack (the last spread argument) to **cdr-normal.**

  Since arguments are pushed with **cdr-next,** the stack now contains a list of all of the arguments.

- Make a rest argument out of the arguments after the max number of spread arguments wanted by the function by creating a **dtp-list** pointer to (frame-pointer + max-args + 2). Push this rest argument onto the stack.

- If apply=0, leave control-register.arg-size and control-register.apply alone. They describe the arguments preceding the rest argument that was just pushed, which is regarded as a local variable of the callee rather than an argument supplied by the caller.

- If apply=1, increment LP and control-register.arg-size, and leave control-register.apply alone. LP now points at the revised rest argument that was just pushed, instead of the original rest argument, which has been turned into the cdr word of a two-word cons.

The function is entered at entry vector element (max-args - min-args + 1) [past

Argument Match-Up Done by Entry Instruction

&rest argument wanted apply argument supplied	False	True
False	If: MIN <= NARGS <= MAX enter at vector (NARGS - MIN) Else: take WNA trap	If: NARGS < MIN take WNA trap If: MIN <= NARGS <= MAX enter at vector (NARGS - MIN) If: NARGS > MAX do a push-apply-args enter at vector (MAX - MIN + 1)
True	If: NARGS < MAX do a pull-apply-args If NARGS = MAX and apply-arg = NIL pop stack clear control-register.apply enter at vector (MAX - MIN) If NARGS = MAX and apply-arg not nil. or NARGS > MAX take a WNA trap	If: NARGS < MAX do a pull-apply-args If: NARGS = MAX set TOS cdr code to cdr-nil enter at vector (MAX - MIN + 1) If: NARGS > MAX do a push-apply-args enter at vector (MAX - MIN + 1)

In this figure, the variables used are

NARGS = cr.argument-size = N-Args + 2 + (cr.extra-argument - cr.apply)
-- that is the actual number of arguments supplied biased by 2,
but not including the apply argument.
MIN = min-args + 2 -- that is, the number in the required-arguments field
of the entry instruction, which is the actual number of required
arguments biased by 2.
MAX = max -args + 2 -- that is, the number in the required-plus-optional-arguments
field of the entry instruction, which is the actual maximum number of
arguments biased by 2.

Figure 25. The argument matchup algorithm

the **&rest** argument default]. Figure 26 illustrates the effect of the push-apply-args operation.

4.4.2 Pull-apply-args

The pull-apply-args operation is invoked when there are fewer than the maximum number of spread arguments and an apply argument was supplied. It pulls some additional spread arguments out of the apply argument.

In detail, pull-apply-args

- pops the list of arguments off the stack,
- extracts an argument from the list,
- pushes it onto the stack,
- pushes the tail of the list onto the stack,
- adjusts control-register.arg-size and the LP, and
- retries the argument match-up process.

Figure 27 illustrates the pull-apply-args operation.

If the apply argument is too short, the control-register.apply bit is turned off; the retry may then signal too few arguments or may simply default some optional arguments. The pull-apply-args operation occurs even if the callee did not want a **&rest** argument; if the desired number of arguments are pulled out of the apply argument and more arguments remain, a wrong number of arguments trap will occur when the argument match-up process is retried.

Following the entry vector, other instructions may appear that perform the operations described next.

In a function with both **&optional** and **&rest** arguments, it is necessary to adjust the LP register to make sure that the **&rest** argument is in LP|0. (If there is a **&rest** argument but not **&optional** arguments, LP will already contain the correct value.) Any function that takes a **&rest** argument may be called with an arbitrary number of spread arguments; push-apply-args will generate the correct **&rest** argument, but there remains an arbitrary distance between FP and SP at the time the function is entered and starts creating its local variables. This is the reason why the local pointer exists; it permits such functions to address their local variables. Functions without **&rest** arguments do not normally use the local pointer. The first instruction after the entry vector, when there are both **&optional** and **&rest** arguments, is a **locate-locals** instruction, which does the following:

register.apply alone. LP now points at the revised rest argument that was just pushed, instead of the original rest argument, which has been turned into the cdr word of a two-word cons.

The function is entered at entry vector element (max-args - min-args + 1) [past the **&rest** argument default]. Figure 26 illustrates the effect of the push-apply-args operation.

4.4.2 Pull-apply-args

The pull-apply-args operation is invoked when there are fewer than the maximum number of spread arguments and a **&rest** argument was supplied. It pulls some additional spread arguments out of the **&rest** argument.

In detail, pull-apply-args

- pops the list of arguments off the stack,
- extracts an argument from the list,
- pushes it onto the stack,
- pushes the tail of the list onto the stack,
- adjusts control-register.arg-size and the LP, and
- retries the argument match-up process.

Figure 27 illustrates the pull-apply-args operation.

If the **&rest** argument is too short, the control-register.apply bit is turned off; the retry may then signal too few arguments or may simply default some optional arguments. The pull-apply-args operation occurs even if the callee did not want a **&rest** argument; if the desired number of arguments are pulled out of the **&rest** argument and more arguments remain, a wrong number of arguments trap will occur when the argument match-up process is retried.

Following the entry vector, other instructions may appear that perform the operations described next.

In a function with both **&optional** and **&rest** arguments, it is necessary to adjust the LP register to make sure that the **&rest** argument is in LP|0. (If there is a **&rest** argument but not **&optional** arguments, LP will already contain the correct value.) Any function that takes a **&rest** argument may be called with an arbitrary number of spread arguments; push-apply-args will generate the correct **&rest** argument, but there remains an arbitrary distance between FP and SP at the time the function is entered and starts creating its local variables. This is the reason

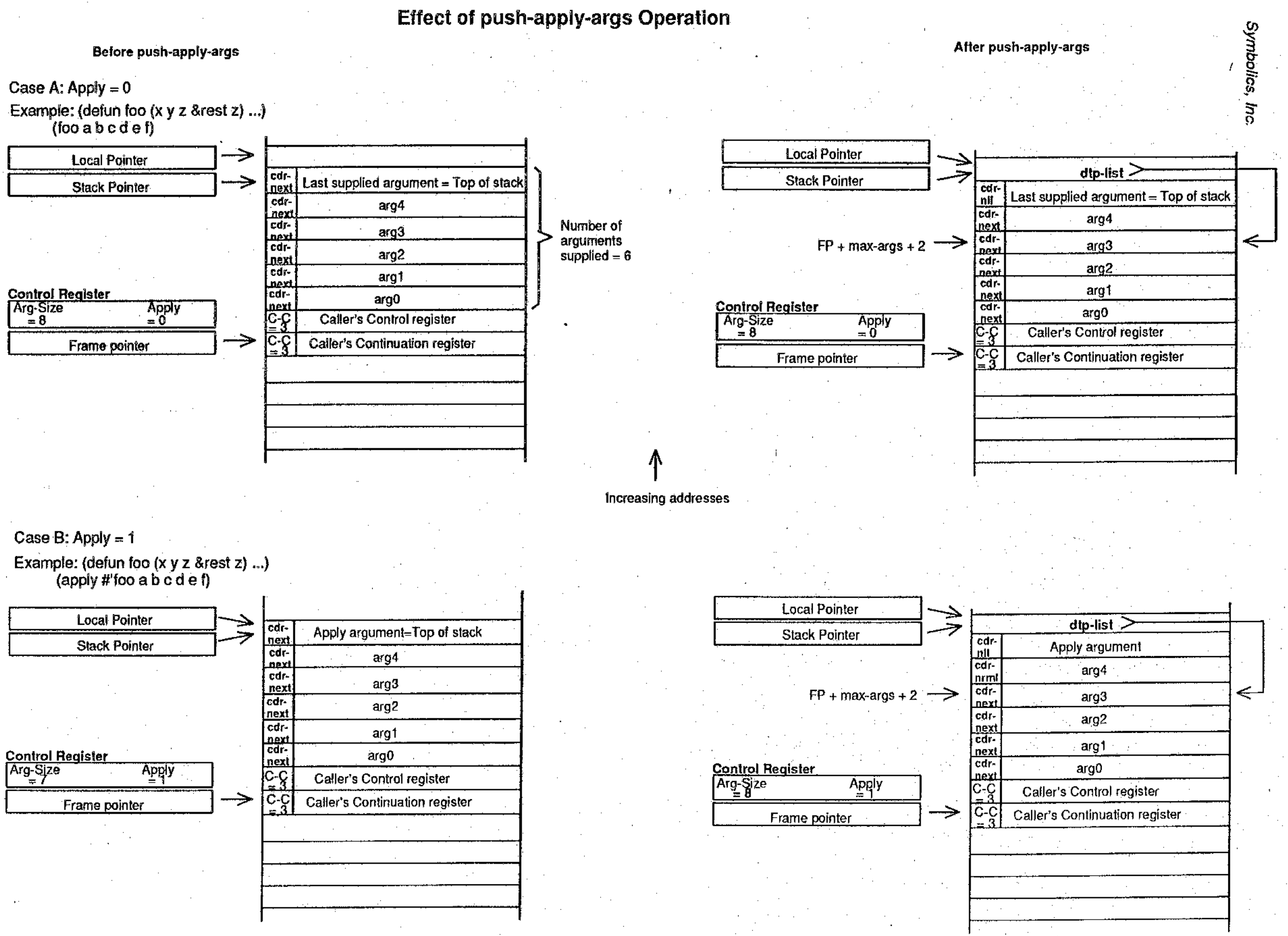


Figure 26. The push-apply-args operation

Effect of pull-apply-arguments Operation

Example: (defun foo (x y) ...)
(apply #'foo a b)

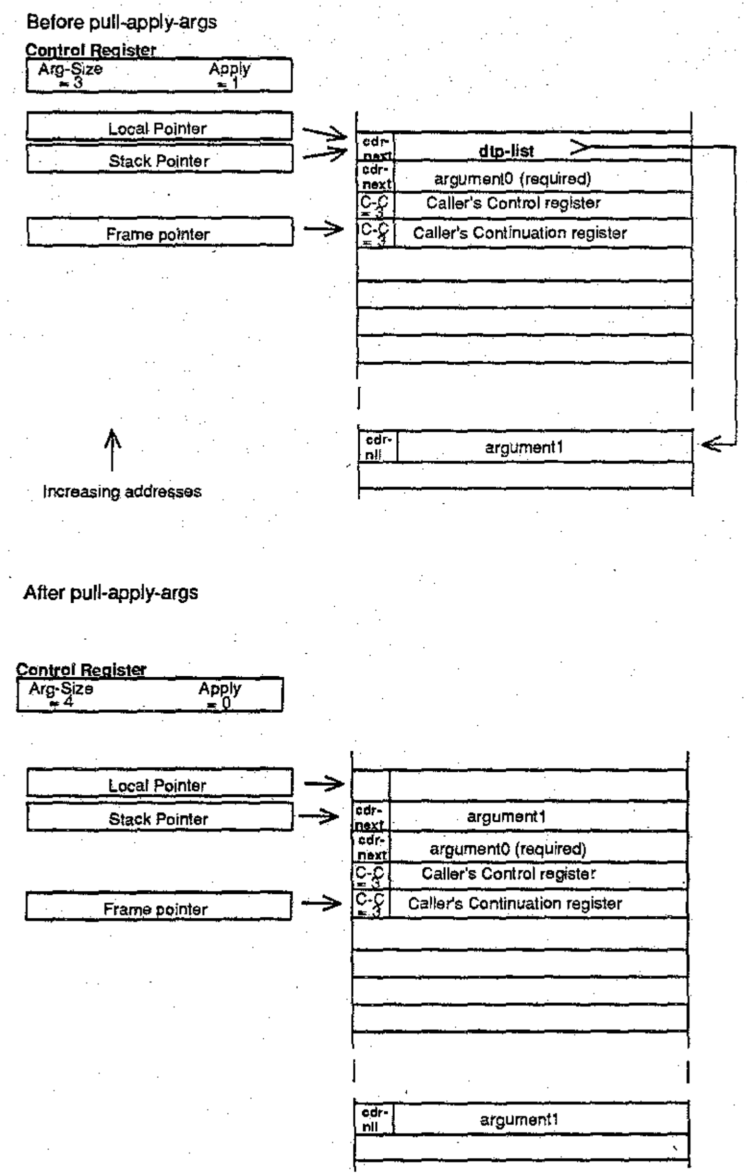


Figure 27. The pull-apply-args operation

why the local pointer exists; it permits such functions to address their local variables. Functions without **&rest** arguments do not normally use the local pointer. The first instruction after the entry vector, when there are both **&optional** and **&rest** arguments, is a **locate-locals** instruction, which does the following:

- Push (control-register.arg-size - 2) onto the stack, as a fixnum. This is the number of spread arguments that were supplied, which is less than the number of spread arguments now in the stack if some **&optional** arguments were defaulted. If the rest arg is not **nil**, this fixnum can be larger than the maximum number of spread arguments accepted.
- Set LP to (new-SP - 1). Thus LP|0 is the **&rest** argument and LP|1 is the argument count. *new-SP* here refers to the SP after the incrementation caused by the *locate-locals* instruction.
- Set control-register.arg-size to (LP - FP) as always.

Figure 28 shows how **locate-locals** works.

The next step is to create the auxiliary **supplied-p** variables for optional arguments. Each of these variables is stored as a local variable (after all the arguments) whose initial value is created by arithmetic comparison between the number of arguments supplied and an appropriate constant. The number of arguments supplied is control-register.arg-size - 2 except in functions with both **&optional** and **&rest** arguments, where it is LP|1. The computation can be performed with a sequence of existing instructions. The initialization of **supplied-p** variables recomputes information that was available while exectuting the entry vector, but there was no space in the stack to store that information then.

The next step takes care of any arguments that were declared special by binding the special variables to the values using the normal instructions for that purpose. If there are any non-special arguments after the special arguments, orphan words will be left in the stack since the values of the special arguments cannot be popped off.

If there are problematic dependencies among optional-argument default-value computations, special care is required. A problematic dependency occurs if the default value for an optional argument depends on a **supplied-p** variable of a previous optional argument or can be affected by a previous argument that is declared SPECIAL. The 3600 handles this with an alternate function entry sequence that the compiler generates if necessary. The I Machine will handle it by using **nil** as the default value in the entry vector and then generating code after the entry vector that tests whether the argument was supplied (just as if initializing a **supplied-p** variable) and if not computes the default value and pops

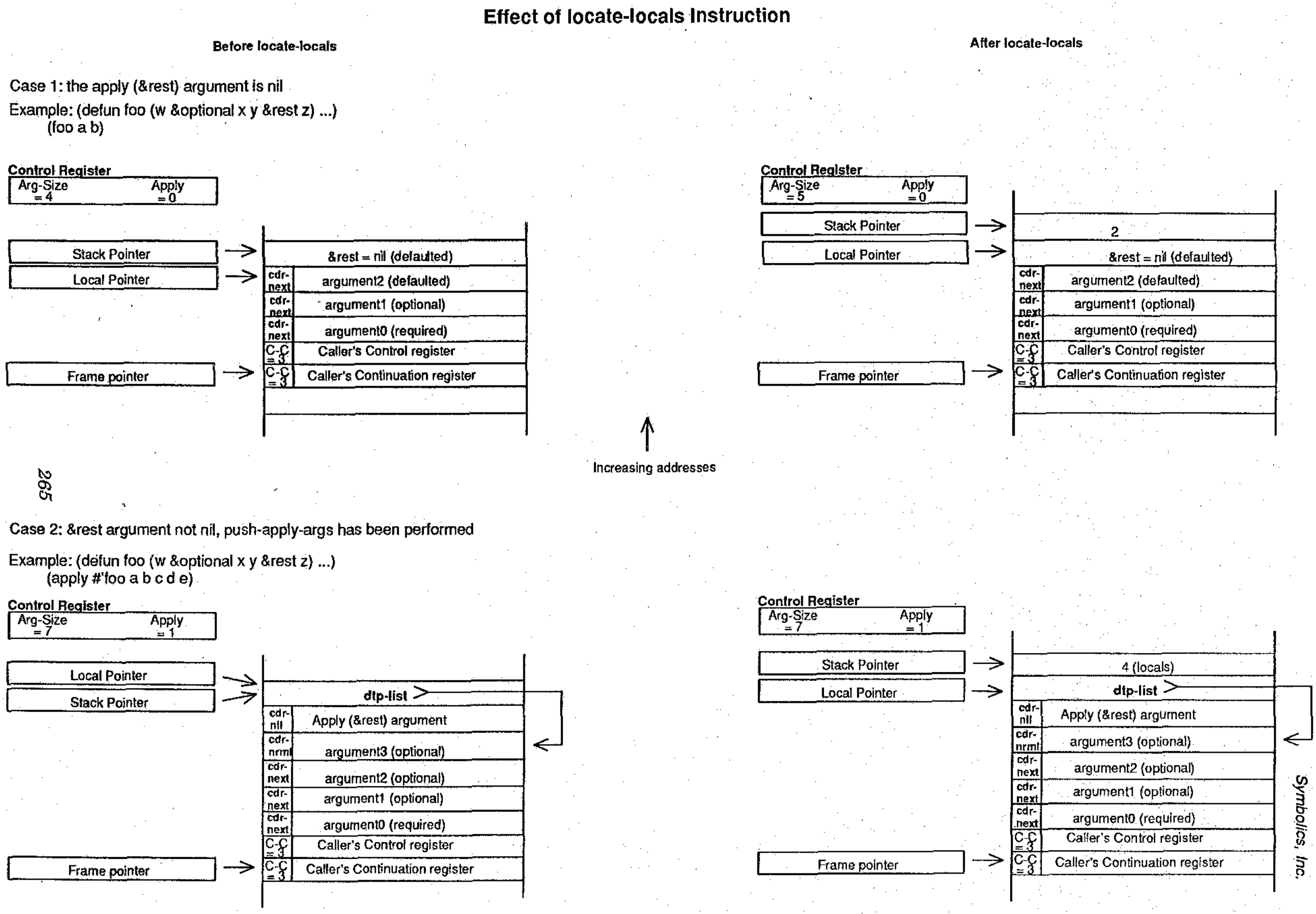


Figure 28. The effect of the **locate-locals** instruction

it into the argument's slot in the stack. This code is interleaved with the binding of special variables so that everything happens in the right order.

Note that if a **supplied-p** variable is used in a read-only way, the value can simply be computed where it is needed, rather than waiting until a stack slot is allocated for the variable, and the problematic case need not occur.

The next step is to compute the values of **&key** arguments and push them on the stack as local variables. This is done with code that looks at the rest argument, just as on the 3600.

This completes the function entry sequence. If the body of the function creates local variables (or **&aux** variables) pushing the initial value of the variable on the stack allocates a stack slot, just as on the 3600. These stack slots can be addressed from the top of the stack frame (relative to SP) or can be addressed from the bottom of the stack frame (relative to FP if the function does not take a **&rest** argument or relative to LP if it does).

4.4.3 Trapping Out of Entry and Restarting

Traps can occur in an entry instruction. Error traps such as wrong number of arguments are handled in the ordinary way.

The pull-apply-args operation references memory, so it is possible for it to trap. Usually, however, the **&rest** argument will be a cdr-coded list in the stack and no trap will occur; these cases are handled quickly by microcode. It is implementation-dependent whether the pull-apply-args microcode handles the full generality of **car** and **cdr**, including non-cdr-coded lists and invisible pointers. Cases it does not handle make the stack frame self-consistent and then call a special trap handler that performs the rest of the pull-apply-args operation and then returns to the **entry** instruction, which will not need a pull-apply-args this time. See the section "Pull-apply-args Exception." If the pull-apply-args microcode handles apply arguments in memory, the usual memory traps such as page faults can occur, and are handled by making the state of the stack frame consistent and then calling the usual trap handler. After the reason for the trap has been rectified, the trap handler returns to the **entry** instruction, which will go back into pull-apply-args and should make further progress this time.

4.5 Function Returning

4.5.1 Function Return Instructions

A function returns to its caller by executing one of the return instructions. These instructions specify the value(s) to be returned, remove the returning function's frames from the various stacks, restore the state of the caller, and resume execution of the caller with the returned values on the stack in the form desired by the caller.

The value(s) to be returned can be constant or can be some number of words at the top of the stack; the number of words can be either fixed or variable.

The form of values desired by the caller can be to throw all the values away, to push the first value on the stack, or to push on the stack all the values and a fixnum which is the number of values excluding itself. The caller uses the value-disposition field of the Control register to specify the desired form of values. Note that any form of values supplied to the return instruction can be converted to any form of values desired by the caller. In addition to this format conversion, the return instruction must move the values from one place in the stack to another, from the callee's frame to the caller's frame.

The return instructions are:

return-single	Return a single value.
return-multiple	Return multiple values (zero or more).
return-kludge	Return multiple values in a non-standard form.

return-single has an immediate operand that addresses an internal register that supplies the value to be returned. The values that can be returned include **nil**, **t**, and the top-of-stack. **return-single** does not do anything that cannot be done with **return-multiple** (accompanied by a **push** in some cases), but it is likely that **return-single** can be implemented to be much faster than the corresponding **return-multiple,** which will speed up important common cases.

return-multiple has a standard operand that specifies the number of values to be returned. The values themselves are on the top of the stack. The operand must be a non-negative fixnum. If there is an implementation dependent upper limit on the number of values, it must be at least 16. Although **return-multiple** takes a standard operand, only immediate and sp-pop operands are legal. (The reason for this is discussed below.)

return-kludge takes the same argument as **return-multiple,** but it returns the values in a different way. **return-kludge** ignores the value disposition and simply places the values at the top of the caller's stack, without pushing the number of values. **return-kludge** is used for certain internal stack-manipulating subroutines and all trap handlers. Note that because **return-kludge** does not return values according to the standard calling sequence, it can only be used in subroutines that are specially known by the compiler, and in certain trap handlers.

Note that the description of return values in the instructions above is from the callee's perspective. In other words, this represents what the function would normally return upon completion. The value-disposition field in the Control register, set by the caller, specifies what should actually be done with the return value(s) (that is, they could be discarded).

Before return can remove the frame from the stack, it may have to perform other cleanup actions. These are specified by the Cleanup Bits in the Control register being nonzero. The actions include popping the binding stack, popping the catch stack (a list threaded through the control stack), executing unwind-protect instructions (which may pop the data stack), and escaping to arbitrary software. See the section "Frame Cleanup."

Once these cleanups have been taken care of, the return instruction restores the state of the caller using the information saved in the frame header of the frame being abandoned, according to this procedure:

```
PC                    <=  Continuation register (unless vd is return)
Continuation register <=  FP|0
temp                  <=  FP|1
SP                    <=  FP - 1
FP                    <=  FP - control-register.frame-size-of-caller
Control Register      <=  temp
LP                    <=  FP + control-register.arg-size
```

At this point the function's frame has been removed from the control stack. The stack cache now is either empty or contains part or all of the caller's frame. Since the frame that was just removed from the stack was entirely in the stack cache, the lowest word in the stack cache is less than or equal to SP+1; if equal, the stack cache is empty. The return instruction does not worry about refilling the stack cache at this stage.

The return instruction now places the values being returned at the top of the control stack, according to the value disposition field in the old Control Register and the particular type of return instruction being executed. The **return-single** instruction can simply push its argument, but the **return-multiple** and **return-kludge** instructions may have to transfer a block of values. The source and destination locations of this block can overlap, both in virtual memory and in stack-cache memory, so care must be taken when copying the block of values to its new location.

The specific handling of the value disposition is as follows:

Effect — Leave the stack alone. This leaves the TOS register invalid.

Value — Push the first value being returned onto the stack. If no values were being returned, use **nil** as the first value.

Multiple — Copy the values down from the old top of the stack to the new top of the stack, and form them into a multiple group by appending a count.

Return — Copy the arguments to the Return instruction down to the new top of the stack and then re-execute the instruction. If the instruction was **return-multiple** and its operand was sp-pop, the count of values must be pushed back on the stack.

The final thing the return instruction does is to make sure that the frame being returned to is contained in the stack cache. If necessary, words in the frame are fetched from main memory. If a trap or interrupt occurs during this process, PC points at the instruction in the caller being returned to, not at the return instruction, so that the return instruction is not retried (which would return from an extra level of call). When the trap/interrupt handler returns, its return instruction will continue loading the frame into the stack cache. Note that the stack cache must be refilled in *decreasing* order of addresses, so that if a trap occurs the range of addresses validly contained in the stack cache will be contiguous. See the section "Revision 0 Implementation Function-Calling Features."

When the value disposition is Return, the stack cache is refilled if necessary and then the return instruction is re-executed, causing the value(s) to be returned from the caller. This process can be repeated any number of times.

If the callee returns more values than will fit in the caller's frame, the hardware takes an error trap out of the callee's **return** instruction, before the stack becomes illegal.

In order to allow smooth trapping out of the middle of a **return,** it is required that all **return** instructions keep their state, if any, at the top of the stack. This means that we cannot have a **return-local** instruction that returns the value of a local variable; you have to first push the value on the stack and then return it from there with **return-single.** Similarly, the number-of-values operand of a **return-multiple** instruction cannot be addressed with FP-relative addressing; only immediate and sp-pop operands are allowed. This restriction eliminates any need to play around with special macro-PCs; any trap out of a **return** leaves the PC pointing at the original **return** instruction and the stack set up so that the instruction can be retried.

Returning from a call that had Value-disposition equal to Effect does not restore the TOS register from the top of the stack. This is because there is no time to do it: three reads from the stack cache would be required in this case, whereas when the Value-disposition equals Value, two reads from the stack cache plus one write are required and the **return-single** instruction executes in only two cycles. This is normally not a problem, since the compiler can compensate, just as it does on the 3600 for other instructions that leave TOS invalid. The compiler simply knows that a finish-call instruction with a value disposition of Effect has the smashes-stack attribute.

4.5.2 Frame Cleanup

The Cleanup Bits in the Control register specify actions necessary before the frame can be exited. Traps, such as page faults, can occur while cleaning up. After handling the trap, the return instruction is retried. The state of the stack while cleaning up is always self-consistent.

The bits and the cleanup actions they cause are as follows, listed in the order that they are processed:

Catch

This bit indicates there are catch/unwind-protect blocks to be unthreaded. Unthreading a block examines the words in the stack addressed by the catch-block-pointer register. If the catch block is for an unwind-protect (that is, if bit 38 = 1 in the binding-stack-pointer word of the catch block), the following actions are performed:

- Restore stack-pointer to its original value, if it was popped by an sp-pop operand.
- If the catch-block-binding-stack-pointer is less than the binding-stack-pointer, unbind special variables until the two pointers are equal. Note that this can clear the Bindings cleanup bit.
- Push the current PC with the current value of control-register.cleanup-in-progress in bit 38 and 1 in bit 39 onto the stack.
- Set the PC to the catch-block-PC, which is the address of the cleanup handler.
- Set the cleanup-in-progress bit in the Control register.
- Set control-register.cleanup-catch in accordance with the cdr code of catch-block-previous and at the same time restore the control-register.extra-argument bit.
- Set the catch-block-pointer register to the catch-block-previous, which is the address of the previous catch block or nil if there is none.
- Transfer control to the first instruction of the cleanup handler. When the cleanup handler exits the return instruction will be retried.

If the catch block is for a catch (that is, bit 38 = 0 in the binding-stack-pointer word of the catch block), only the catch block need be removed (bindings will be undone by cleanup because the bindings cleanup bit will be set for the frame). The following actions are taken:

- Set control-register.cleanup-catch in accordance with the cdr code of catch-block-previous. The hardware is permitted, but not required, to restore control-register.extra-argument.
- Set the catch-block-pointer register to the catch-block-previous, which is the address of the previous catch block or **nil** if there is none.
- Check the cleanup bits again.

Bindings

This bit indicates there is a non-empty binding-stack frame associated with this control-stack frame, in other words that this function has bound some special variables. Pop the binding stack and undo bindings until a binding stack entry whose binding-stack-chain-bit is zero is encountered. Then clear control-register.cleanup-bindings and check the cleanup bits again.

Trap-on-Exit

Take a trap. If the trap handler clears the Trap-on-Exit bit and returns, the return instruction can proceed.

4.5.3 Value Matchup

When Value-disposition is Multiple, the instruction after a finish-call instruction will usually be a **take-values** instruction. As on the 3600, this converts the multiple group left on the stack by **return** into the desired number of values, popping extra values or pushing **nil** as a default for missing values.

4.6 Catch, Throw and Unwind-Protect

A catch block is a sequence of words in the control stack that describes an active catch or unwind-protect operation. All catch blocks in any given stack are linked together, each block containing the address of the next outer block. They are linked in decreasing order of addresses. An internal register named catch-block-pointer contains the address of the innermost catch block, as a dtp-locative, or contains nil if there are no active catch blocks. The address of a catch block is the address of its catch-block-pc word.

The format of a catch block for a catch operation is as follows:

Word Name	*Bit 39*	*Bit 38*	*Contents*
catch-block-tag	0	invalid flag	any object reference
catch-block-pc	0	0	catch exit address
catch-block-binding-stack-pointer	0	0	binding stack level
catch-block-previous	extra-arg	cleanup-catch	previous catch block
catch-block-continuation	value-disposition		continuation

The format of a catch block for the unwind-protect operation is:

Word Name	*Bit 39*	*Bit 38*	*Contents*
catch-block-pc	0	0	cleanup handler
catch-block-binding-stack-pointer	0	1	binding stack level
catch-block-previous	extra-arg	cleanup-catch	previous catch block

The *catch-block-tag* word refers to an object that identifies the particular catch operation, that is, the first argument of **catch-open** or **catch-close**. The catch-block-invalid-flag bit in this word is initialized to 0, and is set to 1 by the **throw** function when it is no longer valid to throw to this catch block; this addresses a problem with aborting out of the middle of a throw and throwing again. This word is not used by an unwind-protect operation and is only known about by the **throw** function, not by hardware.

The *catch-block-pc* word has data type **dtp-even-pc** or **dtp-odd-pc.** For a catch operation, it contains the address to which **throw** should transfer control. For an unwind-protect operation, it contains the address of the first instruction of the cleanup handler. The cdr code of this word is set to zero (**cdr-next**) and not used. For a catch operation with a value disposition of Return, the catch-block-pc word contains **nil.**

The *catch-block-binding-stack-pointer* word contains the value of the binding-stack-pointer hardware register at the time the catch or unwind-protect was established. When undoing the catch or unwind-protect, special-variable bindings are undone until the binding-stack-pointer again has this value. The cdr-code field of this word uses bit 38 to distinguish between catch and unwind-protect; bit 39 is set to zero and not used.

The *catch-block-previous* word contains a dtp-locative pointer to the catch-block-pc word of the previous catch block, or else contains **nil.** The cdr-code field of this word saves two bits of the control register that need to be restored.

The *catch-block-continuation* word saves the Continuation hardware register so that **throw** can restore it. The cdr-code field of this word saves the value disposition of a catch; this tells the **throw** function where to put the values thrown. This word is not used by unwind-protect.

An unwind-protect cleanup handler terminates with a **%jump** instruction. This instruction checks that the data type of the top word on the stack is **dtp-even-pc** or **dtp-odd-pc,** jumps to that address, and pops the stack. In addition, if the bit 39 of the top word on the stack is 1, it stores bit 38 of that word into control-register.cleanup-in-progress. If bit 39 is 0, it leaves the control register alone.

The compilation of the **catch** special form is approximately as follows:

Code to push the catch tag on the stack.
Push a constant PC, the address of the first instruction after the catch.
A **catch-open** instruction.
The body of the catch.
A **catch-close** instruction.
Code to move the values of the body to where they are wanted;
  this usually includes removing the 5 words of the catch block
  from the stack.

The compilation of the **unwind-protect** special form is approximately as follows:

Push a constant PC, the address of the cleanup handler.
A **catch-open** instruction.
The body of the unwind-protect.
A **catch-close** instruction.
Code to move the values of the body to where they are wanted; this
  usually includes removing the 3 words of the catch block from
  the stack.

Somewhere later in the compiled function:

The body of the cleanup handler.
A **%jump** instruction.

Each active catch or unwind-protect operation has an associated catch-block stored in the control stack and linked onto a list whose root is a processor register, named **%catch-block-list,** that is saved in the stack group by context switch.

All the frames between the current frame and the destination of the throw are "unwound" individually, and the data stack is taken care of by this. Each frame that uses the data-stack has an unwind-protect to clean it up. The binding stack is also taken care of by this; the only reason for the binding SP in the catch block is because bindings can happen at any point in the function, and only those that happened after the catch should be undone (the binding stack itself only says with which frame the bindings are associated, not where in the frame).

The implementation of **throw** is somewhat similar to the way it is done on the 3600, but simpler and with less special kludgery. A **throw** special form

```
(throw <tag> <values>)
```

is compiled as

```
(multiple-value-call #'%THROW (VALUES <tag>) <values>)
```

which calls **%throw** with the value of <tag> as its first argument and the values of <values> as its remaining arguments. **%throw** starts by searching the list of catch blocks for one with the correct tag. If it doesn't find one, or if the catch-block-invalid bit is set in the block it finds, it signals an error. Having located the destination catch block, **%throw** prepares to discard all intervening stack frames and catch blocks; this requires invoking any unwind-protect cleanup handlers that are present, each in its proper stack frame and special-variable binding environment. **%throw** changes the value disposition of each intervening stack frame to Return, and sets the catch-block-invalid bit in each intervening catch block. Next, **%throw** examines the restart PC and value disposition of the destination catch block, and modifies the return PC and value disposition of the next frame in the stack, the one that was called by the frame containing the catch block. There are two cases:

If the catch value disposition is Return, **%throw** sets the frame value disposition to Return and returns the values to be thrown. These values are passed back through all the intervening frames, since their value dispositions are Return, and eventually arrive at the desired destination.

Otherwise, **%throw** sets the frame value disposition to Multiple, sets the frame return PC to the address of a hand-crafted helping routine, pushes the following values on the stack, and executes a **return-multiple** instruction that returns these values through all of the intervening frames. The values pushed are:

- the words to be left in the stack when control reaches the catch's restart PC. This depends on the catch's value disposition and could be nothing, one word, or a multiple group. These are derived from the values to be thrown passed to %THROW as its arguments.

- The catch's restart PC.

- The number of catch blocks to be closed in the destination frame. This is at least 1, and will be more if there are other catches inside the destination catch in the same frame.

- The number of special variable bindings to be undone. This is always zero in this context, but the same helping routine is used for other purposes.

- A count of the total number of values, to make this a valid multiple group.

The hand-crafted helping routine proceeds as follows:

- Loop executing **catch-close** instructions the specified number of times.
- Loop executing **unbind** instructions the specified number of times.
- Pop the top three words off the stack.
- Do a **%jump** instruction, which jumps to the catch's restart PC and leaves the values thrown in the stack.

Note that the return PC and value disposition that need to be modified are actually stored in the frame header of the frame two frames up in the stack from the frame containing the destination catch block. The frame containing the destination catch block could be the same one that called **%throw.** In order to avoid having to modify the internal processor registers (Return PC and Control register), **%throw** calls itself recursively in this case.

The purpose of the catch-block-invalid bit is to detect the case where a **throw** begins, is interrupted part way through, and the interrupt handler does another **throw** to a catch that is inside the original catch. This can also happen if an unwind-protect cleanup handler gets an error and a **throw** occurs from the Debugger. Since the stack has already been clobbered by changing the value disposition of the frame containing this new catch, the program would operate incorrectly if the second **throw** was permitted to occur. The 3600 deals with this differently; it doesn't modify the value disposition of each frame until it is just about to return from it. This still has a possibility of the same bug, since there could be a catch in the frame being returned from, but the timing window is open for a much smaller time. The 3600's method is more difficult to do on the IMach because of the Control register.

catch-block-invalid catches nonlocal, but lexical, **gos** and **returns** too, since they are compiled as **throw** to a special tag. It does not catch local **gos** and **returns** out of unwind-protect cleanup handlers, but those are thoroughly illegal!

4.7 Generic Functions and Message Passing

The flavor system deals with flavors, instances, instance variables, generic functions, and message passing. A flavor describes the behavior of a family of similar instances. An instance is an object whose behavior is described by a flavor. An instance variable is a variable that has a separate value associated with each instance. A generic function is a function whose implementation dispatches on the flavor of its first argument and selects a method that gets called as the body of the generic function. In message passing, an instance is called as a function; its first argument, known as the message name, is a symbol that is dispatched upon to select a method that gets called. Message passing is the pre-Release-7 reason for generic functions; we plan to phase it out eventually (over several years).

4.7.1 Flavor

A flavor is a structure that contains information shared by all its instances. The header of each instance points into the middle of the structure, at three words known by hardware. Other portions of the flavor are architecturally defined, but not known by hardware. Still other portions of the flavor are known only by the internals of the flavor system.

The data-representation chapter lists the architecturally defined fields of a flavor. See the section "Flavor Instances."

4.7.2 Handler Table

A handler table is a hash table that maps from a generic function or a message to the method to be invoked and a parameter used by that method to access instance variables. The details concerning the contents of a handler table are presented elsewhere. See the section "Flavor Instances."

The hashing function used to search the handler table is designed to maximize speed and simplify hardware implementation, not to maximize density. It is optimized assuming that the search succeeds on the first or second probe of the hash table. It operates as follows:

- **logand** the generic function or message name with the hash mask from the flavor.
- Multiply the result by 3 (this is just a shift and an add).
- Add the product to the handler hash table address from the flavor and initiate a block read of sequential locations starting at that address.
- For each block of three words, if the first word does not match the generic function or message name, and is not **nil**, skip the next two words and go on to the next block.
- When a block is found whose key matches or is **nil**, accept the method and the parameter and terminate the search.

Note that when a mismatch occurs, the hash search proceeds through consecutive addresses; it does not rehash. It also does not wrap around when it gets to the end of the table. Consequently the software must allocate sufficient room at the end of the table, after the highest address defined by the hash mask, to accomodate overflow from the end of the table and a final entry with a key of **nil** that is guaranteed to terminate the search.

The hash mask is normally a power of 2 minus 1.

Methods are **dtp-even-pc** or **dtp-odd-pc**. An interpreted method invokes a special entry point to the Lisp interpreter; this is implemented by storing the interpreter (the PC that points to its first instruction) as the method and storing the actual method as the parameter.

4.7.3 Calling a Generic Function

A call to a generic function can be started by **dtp-call-generic, dtp-call-generic-prefetch, dtp-call-indirect, dtp-call-indirect-prefetch** that finds a **dtp-generic-function,** or a **start-call** instruction whose operand is a **dtp-generic-function.** In any case, the generic function is pushed as the extra-argument to the call and the continuation is set to the trap-vector element for calling a **dtp-generic-function.** When the call is finished, control transfers to the continuation, which is always a function that consists of nothing but a **%generic-dispatch** instruction (there is no entry vector).

The **%generic-dispatch** instruction sees the following on the stack:

```
FP|0,1        the usual function-call save area
FP|2          the generic function
FP|3          the instance
FP|4,5,...    additional arguments, if any
```

%generic-dispatch operates as follows:

- Make sure that the number of "spread arguments" is at least 2. This ensures that FP|2 and FP|3 are valid. If necessary, perform a pull-lexpr-args operation. If that fails to produce two arguments, signal a "too few arguments" error.

- Get the address of the interesting part of the flavor, which specifies the size and address of the handler hash table. This is done by checking whether the data type of FP|3 is one of the instance data types. If it is, fetch its header following forwarding pointers (header-read). If it is not, use the data type to index a 64-element table in the trap vector that points to the hash-mask fields of the flavor descriptions.

- Fetch two words from the flavor, the hash mask and hash-table address, and perform the handler hash table search described above. If the parameter is not **nil**, store it into FP|2, otherwise leave the generic function in FP|2 (the default handler needs it). If the method is **dtp-even pc** or **dtp-odd-pc**, jump to its entry instruction. If the method is anything else, trap (this is an error).

4.7.4 Sending a Message

Sending a message occurs when **dtp-call-indirect, dtp-call-indirect-prefetch** or a **start-call** instruction finds an instance data type as the function. It pushes the instance as the extra-argument to the call and sets the continuation to the trap-vector element for calling that data type. When the call is finished, control transfers to the continuation, which is a function that dispatches to the appropriate method.

At this point, the stack contains the following:

FP|0,1	the usual function-call save area
FP|2	the instance
FP|3	the message
FP|4,5,...	additional arguments, if any

This is almost like the generic function case except that FP|2 and FP|3 have been exchanged. The distinction between a message and a generic function is unimportant at this level; they are both used only as keys for searching the handler hash table.

The **%message-dispatch** instruction, whose description is similar to that of **%generic-dispatch** except that the arguments are interchanged, accomplishes the dispatch by effecting results equivalent to the following sequence of instructions:

```
ENTRY       MIN ARGS = 2, MAX ARGS = ∞
PUSH FP|2
PUSH FP|3
POP FP|2
POP FP|3
%GENERIC-DISPATCH
```

Note that an **entry** instruction cannot actually be used in this manner, so the **%message-dispatch** instruction must exist.

4.7.5 Accessing Instance Variables

Instructions exist to read, write, and locate instance variables.

- Read: fetch the value of the variable, trapping if it is **dtp-null,** and push the value on the stack.

- Write: pop a value off the stack and store it into the instance variable, preserving the cdr code of the location and checking for invisible pointers and **dtp-monitor-forward** (the same as when writing a special variable).

- Locate: compute the address of the instance variable's value cell and push it on the stack with **dtp-locative.** If the value cell contains an invisible pointer, **dtp-null,** or **dtp-monitor-forward,** that has no effect on the result of this instruction.

These instructions are parameterized by the instance in question and the offset within that instance of the instance-variable slot. There are three groups of instructions:

- Access an arbitrary instance, typified by **%instance-ref**: The instruction receives the instance and the offset as ordinary arguments.

- Access **self** unmapped, typified by **push-instance-variable-ordered**: The instruction finds the instance in FP|3 (the first argument in the current stack frame, after the extra-argument) and receives the offset as an immediate operand.

- Access **self** mapped, typified by **push-instance-variable**: The instruction finds the instance in FP|3 (the first argument in the current stack frame, after the extra-argument), receives an instance variable number as an immediate operand, and finds a mapping table in FP|2 (the extra-argument or "environment"). The mapping table is always a simple, short-prefix ART-Q array. The instance variable number is used as a subscript into the mapping table to get the offset. [Note to those who understand the format of mapping tables used in Release 6 on the 3600: some slots in mapping tables are used for instance variable offsets as described here; other slots are used for other purposes such as subsidiary mapping tables for combined methods. The slots are allocated dynamically by the flavor system as they are required and in general the two types of slots will be interspersed. This eliminates the complexity and slowness of using array-leaders and art-16b arrays.]

If an instance has been **structure-forward**ed to another instance, the value of **self** (FP|3) in a method is the original instance. This means that the instructions to access instance variables must check the header of the instance for a **dtp-header-forward,** just as the array referencing instructions do, before adding the offset to the address of the header to get the address of the instance variable.

4.8 Stack-Group Switching

The major steps of a stack-group switch are:

1. Inhibit preemption
2. Check the state of the new stack group for resumability
3. Set argument, resumer of new stack group
4. Save internal processor and coprocessor registers
5. Swap out special-variable bindings of the current stack group
6. Make sure the new stack group is prepared for execution
7. Dump the stack cache
8. Switch to the new stack and load the stack cache
9. Restore internal processor and coprocessor registers
10. Swap in special-variable bindings of the new stack group
11. Enable preemption and return

Saving internal processor and coprocessor registers is done by using **%read-internal-register** instructions to read the registers into local variables in the stack. When the switch to the new stack group is done, the new current stack frame will be one whose local variables contain the register values for the new stack group.

To restore internal processor and coprocessor registers, use **%write-internal-register** instructions to pop the local variables off the stack and put them back in the registers.

Swapping special-variable bindings in and out is the same except that swapping *in* traverses the binding stack in *ascending* address order and swapping *out* traverses it in *descending* address order. All memory reads are done with block-read or **%memory-read** instructions, since those contain magic bits to select special memory operand reference types.

The basic procedure to swap one binding, assuming that *P* points to a pair of words in the binding stack, is:

```
loc ← data_read(P)                    ;Get address of bound cell
old ← bind_read_no_monitor(P+1)       ;Get old contents of
                                      ;that cell
new ← bind_read_no_monitor(loc)       ;Get new contents of
                                      ;that cell
                                      ;If an invisible pointer is
                                      ;followed, update loc
mem(loc) ← merge_cdr(old,new)         ;Store back old contents
                                      ;preserve cdr
mem(P+1) ← new                        ;Store new contents into
                                      ;binding stack
```

P and *loc* are block-address registers (BARs), *old* and *new* are locations in the stack, *data_read* and *bind_read_no_monitor* are memory read operations described in section "Operand References."

In assembly language, the procedure is as follows. Assume *P* is BAR-1, *loc* is BAR-2, and these BARs can be used for both reading and writing. (The order of these instructions might be rearranged to cut down on memory interference and to put the two block-1-reads adjacent, but that is a secondary consideration.)

```
block-1-read                          data_read(P)
write-internal-register bar-2         loc ←
block-1-read last_word,               old ← bind_read_no_monitor(P+1)
  bind_read_no_monitor,no_increment
block-2-read last_word,               new ← bind_read_no_monitor(loc)
  bind_read_no_monitor,preserve_cdr,no_increment
merge-cdr-nopop sp|-1                 cdr(old) ← cdr(new)
block-1-write sp-pop                  mem(P+1) ← new
block-2-write sp-pop                  mem(loc) ← old
```

To make sure the new stack group is prepared for execution, it is necessary to call a subroutine in the paging system to wire down appropriate pages of the stack, and to run the GC scavenger over those pages if necessary. This also determines the appropriate values for the stack limit registers. The paging system maintains enough state so that this operation is very fast if the stack group has been run recently. Doing this before actually switching to that stack ensures that no traps (page faults or transport traps) can occur during the actual act of switching, when things are inconsistent, and ensures that the new stack group has enough space for the extra-stack.

To dump the stack cache, use a loop that does block-read and block-write at identical addresses. The architecture requires that writes to memory locations in the stack cache write through to main memory.

To switch to the new stack and load the stack cache, initialize the registers that control the stack cache to suitable values and then do block-reads to fill it. In detail:

1. Save the SP into the current stack group.
2. Get the SP value of the new stack group. The FP value is at a known offset from this. These bracket a stack frame which is in the same format as the current stack frame, but contains the register values of the other stack group.
3. Go into extra-stack mode so no traps/interrupts can occur.
4. Store the FP value into the hardware FP and into a BAR.
5. Set the stack cache lower bound register to the SP value +1, so that the following block-reads will neither read from the stack cache nor cause it to overflow.
6. Store the FP value minus 1 into the hardware SP. Do this last, since it renders the old stack frame inaccessible.
7. Execute a sequence of block reads that fetch the new stack frame into the stack cache and increment the SP to its appropriate value.
8. Set the stack cache lower bound register to FP. The stack cache is now consistent.
9. Set the stack limit registers to the values for the new stack group.

Restoring the internal processor registers will turn off extra-stack mode by restoring the control register. **return** will restore extra-stack-mode.

4.9 Appendix: Comparison of 3600-Family and I-Machine Function-Calling

To be supplied in the next revision of this specification.

5. Exception Handling

****
This file is confidential. Don't show it to anybody, don't hand it out to people, don't give it to customers, don't hardcopy and leave it lying around, don't talk about it on airplanes, don't use it as sales material, don't give it as background to TSSEs, don't show it off as an example of our (erodable) technical lead, and don't let our competition, potential competition, or even friends learn all about it. Yes, this means you. This notice is to be replaced by the real notice when someone defines what the real notice is.
****

5.1 Traps in General

It is occasionally necessary to escape from a situation that the hardware/microcode cannot handle and give control to some Lisp code. This escape action is known as a trap, and the Lisp code invoked is known as the trap handler. The trap handler rectifies the situation and returns to the interrupted program, which never knows that the trap occurred. Applications for traps include page faults, stack overflows, arithmetic overflows, arithmetic instructions applied to types of numbers that are not built into the hardware, I/O interrupts, execution of instructions that are not implemented by the hardware, and several others.

All trap handlers are functions called in the ordinary way; when an exception occurs the hardware forces a function call to a function found in a "trap vector," with arguments describing the exception and a return PC pointing to the appropriate instruction. Trap handlers written directly as instructions that execute in the stack frame of the function that trapped, as on the 3600, are never used. All trap handlers are Lisp functions.

There are two major categories of traps: pre-traps and post-traps. A pre-trap is used when the trap handler will rectify some condition, such as a non-resident page, and then the trapped instruction is to be retried. A post-trap is used when the trap handler will emulate the desired effect of the trapped instruction and then return to the next instruction in sequence. Most out-and-out errors are pre-traps, simply for the convenience of the hardware and the debugger; in this case the trap handler will never return.

The value disposition for the values produced by a trap handler is undefined. All traps must return their values via **return-kludge**.

Trap handlers are stored in the trap vector. See the section "Trap Vector", page

286. The trap vector is wired to avoid recursive page faults. All trap handlers receive the trap vector index and the PC of the trapped instruction as the first two arguments.

The sequence of events for a pre-trap is as follows:

1. Restore the stack to its condition at the start of the instruction.

2. Push the continuation and control registers onto the stack with cdr code set to 3, set continuation to the contents of the trap vector entry, clear the control-register.extra-argument bit, set the control-register.trace-bits to 0, and set the control-register.trap-mode field to the maximum of the cdr code of the trap vector entry and the current trap mode.

3. Push the trap vector index.

4. Push the PC of the trapped instruction.

5. Push the trap arguments.

6. Do a finish-call operation to invoke the trap handler, using the current PC as the return address. The value disposition is undefined.

The sequence of events for a post-trap is as follows:

1. Save the arguments to the trapped instruction and pop them off the stack.

2. Push the continuation and control registers onto the stack setting the cdr code to 3, set continuation to the contents of the trap vector entry, clear the control-register.extra-argument bit and control-register.trace-bits, and set the control-register.trap-mode field to the maximum of the cdr code of the trap vector entry and the current trap-mode.

3. Push the trap vector index.

4. Push the PC of the trapped instruction.

5. Push the arguments to the trapped instruction.

6. Do a finish-call operation to invoke the trap handler, using the incremented PC as the return address. The value disposition is undefined.

5.2 The Extra Stack

Certain traps, such as page faults and disk-wakeup sequence breaks, have to be handled on a stack that is guaranteed to be in main memory and guaranteed to be large enough. These traps cannot tolerate another trap, such as a page fault on the stack, occurring during their handling. Such traps are handled on the user stack and the architecture and storage system are designed to treat stack pages specially so that no fault can occur while a trap is being handled. This has the advantage that there is no need for special hardware to deal with multiple stacks and context switching.

Two stack-limit registers are provided, one for normal execution and the other for trap handlers. When the second stack-limit register is being used, the machine is said to be "executing on the extra stack." This is not a different stack from the normal control stack, but just extra space reserved at the end of the normal stack for use only by trap handlers. Only the control stack needs extra space; the binding and data stacks are not used by page-fault processing.

The extra space is not actually used by a trap handler unless the stack happened to be close to overflowing at the time of the trap. The trap handler just uses the space starting at the current top of the stack in the user program. If a normal program attempts to use the extra space, it takes a stack overflow trap and software grows the stack before allowing the program to proceed. (The initial handling of the stack overflow trap occurs on the extra stack.) If a trap handler overflows the extra stack, the machine halts. This fatal error indicates either a bug in the trap handler or failure to allocate enough extra space when building the stack-group.

The stack-limit register in use is specified by the processor trap mode.

5.3 Trap Modes

There are four *interrupt levels* or *modes* the processor can be in. The mode the processor is in specifies what can interrupt it, what control stack limit to use, and in one case, how traps work. The current mode is specified by the trap-mode field in the control register.

Level 0, Emulator This is where most code gets run. Low-priority sequence break requests, high-priority sequence break requests, and preempt pending will interrupt the processor.

Level 1, Extra-Stack

This is where the paging system runs, clock sequence breaks, other low-speed I/O, and certain critical routines (such as just after a **%allocate-xyz-block**). Only high-priority sequence break requests can interrupt this.

Level 2, High-Speed I/O

This is where time-critical device service is done. Nothing can interrupt it.

Level 3, FEP mode

FEP code runs in here. Nothing can interrupt it. Additionally, when a trap occurs, it goes through a single trap vector. See the section "FEP-mode Traps", page 293.

Unless the processor is in the emulator mode (the trap mode is nonzero), the machine is allowed to use the extra stack. (Level 1 is called "extra-stack," but levels 2 and 3 also imply the use of the extra stack.)

The trap mode is set to the maximum of the current trap mode and the cdr-code field of the trap-vector entry when a trap is taken. This allows the processor to change mode atomically when entering trap handlers. Restoration of the control register on completion of the trap handler will restore the trap mode to its pre-exception state.

The trap mode is set to 3 by INIT. The trap vector entry for RESET should specify level 3. Note that RESET is not inhibited by the trap mode, in that respect it could be called Non-Maskable-Interrupt.

5.4 Trap Vector

The trap vector is a table whose elements specify the functions to be called when various exceptional conditions occur. Each entry is a PC (**dtp-even-pc/dtp-odd-pc**) that points to the first instruction of the trap handling function. Byte <39:38> (the cdr-code) of the entry is the minimum initial trap mode for the handler. This table is stored at physical addresses 1000000 through 1007777; the trap vector index always supplied as the first argument to a trap handler is relative to the base of this table.

See the section "Trap Vector Layout", page 294.

5.5 Exceptions

5.5.1 Error Traps

When an instruction receives illegal operands, references memory and receives a bad data type, or encounters an instruction-specific error condition, it takes an error pre-trap. The error trap handler takes two arguments (in addition to the trap index and PC): a micro-state, and a VMA. The micro-state is a unique identifier that is looked up in a table to determine the cause of the error. If appropriate, the second argument is the contents of the BAR that caused the error, otherwise it is ignored.

5.5.2 Instruction Exceptions

An instruction exception occurs when an instruction needs to perform some operation that is not an error, but is not directly supported by the hardware (taking the **car** of a list instance, for example). Instruction exceptions are post-traps, called with whatever arguments the instruction takes. The contract of the trap handler is to emulate the behavior of the particular instruction. Occasionally exceptional conditions will arise during emulation, such as the need to redecode an array register or refill a cons cache.

The instruction exception trap handlers are contained in the instruction exception vector, which is indexed by the opcode of the faulting instruction. Note, though, that some instructions are emulated by dispatching through the arithmetic dispatch vector. See the section "Arithmetic Traps", page 287.

A special case of instruction exception occurs when the processor attempts to execute an undefined instruction. In this case, a post-trap is taken, using the trap handler obtained by indexing into the instruction exception vector with the opcode. However, since the number of arguments is not known, the trap microcode presumes that the instruction takes zero arguments, and the trap handler must compensate.

%halt (opcode 377) is guaranteed to be an undefined instruction and will always take an exception.

5.5.3 Arithmetic Traps

To improve the efficiency of simple arithmetic on non-fixnum numbers, instruction exceptions for a number of instructions fetch the trap handler from the arithmetic dispatch vector instead of from the instruction-exception vector. The particular handler fetched depends on the types of the arguments. This reduces the overhead of dispatching on the types of the arguments by moving it into microcode.

All of the instructions that use the arithmetic dispatch vector accept numeric arguments only; if any argument is non-numeric, an error trap will occur. (**eql** is a slight exception to this rule -- it accepts nonnumeric arguments, but will only trap out for numeric arguments). The normal instruction exception vector for these instructions is not used in any circumstances.

There are two different categories of arithmetic traps. Traps in the first category occur when an arithmetic instruction is applied to operands that are numeric types which the hardware does not support for the particular instruction. (Hardware support for certain types may depend on the presence of a coprocessor.) Traps in the second category occur when an exceptional condition (such as arithmetic overflow) results from attempting to perform the arithmetic operation.

In general, information about why a particular arithmetic trap was taken is not available -- the trap handler is expected to check the operands, emulate the operation, check the results for exceptional conditions, and return. In certain circumstances more specific processing is allowed. For example, the only possible exception that can occur while adding two fixnums is an integer overflow, and the trap handler for **add** of fixnum arguments may take advantage of this.

The arithmetic dispatch vector contains sixty-four trap handlers (eight numeric types for up to two arguments) for each instruction that uses it. These trap handlers are invoked via post-traps, in the same manner as normal instruction exceptions. The dispatching trap computes a trap-vector index from bits out of the opcode field of the instruction and bits out of the data types of the arguments. Specifically, for a binary arithmetic trap, the index into the arithmetic dispatch vector is

```
        OPCODE<4:0> | ARG1<34:32> | ARG2<34:32>
```

For a unary instruction, the dispatch acts as though *arg2* were a fixnum; that is, the low three bits of the index will always be zero.

When the two operands are not of the same type, the trap handler may be a shared "coercion function" that simply coerces one of the operands to be compatible with the other, then jumps into the correct trap handler to perform the desired operation for the given type of (coerced) operands. The coercion function does not have to know what the operation is; the appropriate trap handler is fetched from the trap vector indexed by the original trap vector index plus a constant that accounts for the coercion that was performed. It is also possible to have a special-case function for a mixed-type operation (fixnum times bignum is always popular) just by filling in the trap vector asymmetrically.

The following instructions post trap through the arithmetic dispatch vector:

eql (263), **eql-no-pop** (267)
equal-number (260), **equal-number-no-pop** (264)
greaterp (262), **greaterp-no-pop** (266)
lessp (261), **lessp-no-pop** (265)
plusp (36), **minusp** (35), **zerop** (34)
add (300), **sub** (301), **unary-minus** (114)
multiply (202), **quotient** (203), **remainder** (210), **rational-quotient** (211)
ceiling (204), **floor** (205), **truncate** (206), **round** (207)
max (213), **min** (212)

logand (215), **logior** (217), **logxor** (216),
    logtest (273), **logtest-no-pop** (277)
ash (232)

5.5.4 Memory Exceptions

Memory exceptions occur when referencing the contents of a given location in memory. There are three classes of memory exceptions:

- The memory operation could not be performed due to some property of the location. For example, the page might not be resident in main memory.
- The memory operation was performed, but further processing is required due to some property of the contents of the location. For example, the contents might be a pointer to a condemned object.
- A hardware error occurred during the memory operation.

Correctable memory errors are not fatal. They are corrected by the memory interface. The occurrence of a correctable error will be recorded by a flag, and the address and syndrome of the cell in error will be stored in a register. Software should periodically poll this register and log any errors.

An uncorrectable memory error is more serious. It causes an uncorrectable memory error page fault. The trap handler can do whatever is appropriate after the error. It is possible to recover from some uncorrectable errors, and others are fatal.

Memory exceptions are pre-traps that take one argument, the address of the referenced location, in addition to the usual trap-vector-index and fault-pc arguments. The argument type can be either locative (a virtual address), or **dtp-physical-address** (a physical address, not always meaningful). The memory exceptions are:

- Page not resident -- PHT search failed.
- Page fault request -- PHT search succeeded but pht.fault-request is set. See the section "Revision 0 Implementation Memory Features", page 297.
- Write protect violation -- attempted to write into a page with pht.write-protect set.
- Transport trap -- read pointer to oldspace from a page with pht.transport-trap set.
- Uncorrectable ECC error -- location contains an uncorrectable error. See the section "Revision 0 Memory Exceptions", page 300.

- Bus error -- processor received a negative acknowledgement of a read. See the section "Revision 0 Memory Exceptions", page 300.

- Monitor trap -- read a reference of type **dtp-monitor-forward**.

5.5.5 Stack Overflow

Control stack overflow occurs when the finish-call instruction (or the equivalent operation when a trap is taken) detects the frame pointer is greater than stack limit. The limit register used depends on the trap mode of the processor. The stack limit is set lower than the real limit by the maximum size of a stack frame plus the amount of extra space needed to process the stack-overflow trap.

Control stack overflow invokes a special trap handler found in a dedicated trap vector. The trap handler takes no arguments other than the trap-vector index and the fault PC.

Binding stack overflow occurs when a **bind-locative** instruction tries to advance the binding-stack-pointer beyond the binding-stack-limit. Binding stack overflow signals an error trap. The error trap handler must be careful not to bind anything until it has considered the possibility that the error is a binding stack overflow.

The return instructions that return multiple values check for stack frame overflow. If (+ cr.frame-size-of-caller values-being-returned) is greater than stack-frame-maximum-size (an internal register), an error trap is taken.

5.5.6 Sequence Breaks

A sequence break is an asynchronous interruption of the currently executing program. A sequence break causes control to be transferred to one of two PCs found in the trap vector. (Most other computers call this an interrupt, but we cannot use that word without confusion because of the without-interrupts special form in Zetalisp, which only prevents preemption, not sequence breaks.) Sequence breaks are requested by the high-priority and low-priority sequence break request pins on the processor.

A high-priority sequence break trap will be taken at the completion of any macroinstruction where the high-priority sequence break request pin is asserted and the trap mode is either 0 or 1. A low-priority sequence break trap will be taken at the completion of any macroinstruction where the low-priority sequence break request pin is asserted and the trap mode is 0. See the section "Revision 0 Sequence Breaks", page 301.

Like other traps, the sequence-break handling functions execute in the context of the interrupted process. They are essentially pre-traps, called with no arguments (other than the standard ones). These interruptions are intended to be transparent to normal Lisp programs, and therefore the handling functions must be careful what they do.

There are two sources of external sequence breaks: low-speed I/O (for example, disk completion) and high-speed I/O (for example, 56Kb serial line). Low-speed I/O routines may spend a moderately long time executing, if needed. High-speed I/O must by programmer design spend a very small amount of time executing, especially if there is more than one device that needs service.

Programs may synchronize with sequence-break handling functions either by raising the trap mode, or using the **store-conditional** instruction.

All indefinite-duration microcode loops are interruptible by sequence breaks, causing the instruction to be aborted. This includes invisible pointer following, method table searching, indirection through symbols in **start-call,** and **rgetf/member/assoc**. An indefinite-duration microcode loop will of course only be interrupted by a sequence break if the current trap mode permits sequence breaks.

5.5.7 Preemption

Preemption is switching from the current process to the scheduler. This is a software operation, which has hardware support for its initiation.

Preempt-request and preempt-pending are bits in a global register, not in the Control register. These bits are set at the same time by software, such as a clock sequence break handler, that wants to preempt the current process. If preempt-pending is set, and the processor is in emulator mode, then a preempt-request trap occurs after the current instruction completes. The trap handler clears the preempt-pending bit and then checks whether the process can be preempted. If so, it clears preempt-request and passes control to the scheduler. If not, it leaves preempt-request set and returns.

The priority of preempt-pending relative to other traps is:

High	reset
	stack-overflow (in finish-call)
	high-priority-sequence-break
	low-priority-sequence-break and emulator-mode
	preempt-pending and emulator-mode
Low	trace-pending

The check-preempt-request operation sets the preempt-pending flag if the preempt-request flag is set. This causes a trap at the end of the current instruction if the processor is in emulator mode, otherwise the trap is taken as soon as the processor returns to emulator mode.

Anything that unbinds a special variable (whether the **unbind** instruction or an implicit unbind caused by the **return** instruction encountering a cleanup bit) does a check-preempt-request operation. This is the reason why preempt-request is a hardware flag instead of just being a software variable. See the section "Revision 0 Unbinding", page 301.

The **%check-preempt-request** instruction (called **%check-preempt-pending** on the 3600) does a check-preempt-request operation. Those extra-stack trap handlers that wish to check for a pending preempt when they return to the user must do a **%check-preempt-request** instruction; if this sets preempt-pending the trap will go off when the trap handler returns. The **%check-preempt-request** instruction is also used in a couple of places in the garbage collector. This could be open-coded using **%read-internal-register** and **%write-internal-register,** rather than being a real instruction, but is probably easy to implement as an instruction since the logic has to be present already for **unbind.**

Note that function return does not do a check-preempt-request operation unless it unbinds special variables, and instructions that change the processor trap mode do not do a check-preempt-request operation, but may provoke a trap if preempt-pending is already set.

Details on stack-group switching can be found in the function calling chapter. See the section "Stack-Group Switching", page 280.

5.5.8 Trace Traps

Instruction-trace, call-trace, and trace-pending are three bits in the Control register, set and cleared by software in saved copies of the Control register in memory. Trace-pending can also be set by hardware. Reset and Init clear all three of these bits. The hardware clears all three of these bits whenever a trap occurs, after saving the Control register on the stack.

If trace-pending is 1, a trap occurs before executing the next instruction. Note that a sequence break can intervene before the trap actually goes off. There is only one trap vector location for trace-pending, regardless of the semantic significance of the trap to the software. If a return instruction restores a Control-register value with the trace-pending bit set, the trap occurs after completion of the return instruction and before execution of the instruction returned to.

When a **return** instruction is executed repeatedly because of Value-disposition Return, and trace-pending is set by restoring a Control-register value, the trap either occurs immediately or after the repeated Return operations finish; the architecture doesn't specify which. The trace-pending values in the several Control register values that are restored are effectively ORed together, so the trap is not lost.

If instruction-trace is 1 at the beginning of an instruction, completion of the instruction sets trace-pending and causes a trap before the next instruction executes. If a post-trap occurs when instruction-trace is 1, trace-pending is set in the Control register saved as part of taking the trap. This is not true of a pre-trap. If a return instruction restores a Control register value with the instruction-trace bit set, the instruction returned to is executed before the trap occurs.

If call-trace is 1, the finish-call instruction sets trace-pending and causes a trap before the first instruction of the called function executes. If stack overflow occurs simultaneously, trace-pending is set but the stack overflow trap occurs first. When the stack overflow handler returns, the trace trap occurs. Call-trace does not effect the implicit finish-call performed when a trap occurs, because call-trace gets cleared first.

5.5.9 PULL-APPLY-ARGS Exception

See the section "Pull-apply-args", page 261.

A pull-apply-args pre-trap is taken from a function entry instruction to extract additional arguments from an apply argument that the microcode is not capable of doing. The trap handler takes two arguments, the number of arguments to pull, and the apply argument, which is popped off the stack before the trap is taken. The trap handler extracts the arguments, updates the saved Control register to reflect the new state of the previous frame, and return-kludges the extracted arguments and the remaining apply argument, if any, directly into the correct place in the previous frame.

5.5.10 FEP-mode Traps

With few exceptions, traps are not supposed to happen while the FEP code is running. To give the FEP a chance to examine each trap and decide whether or not it is meaningful, all traps while in FEP mode go through a single trap vector. Any given trap will be taken in exactly the same manner, with the same arguments and the same continuation, whether or not the processor is in FEP mode; the only difference is where the trap handler PC comes from.

5.5.11 Processor Faults

A processor fault occurs when the processor encounters a situation from which it cannot proceed. The occurrence of a processor fault halts the processor and indicates the error on an external pin. The causes of a processor fault are:

- Stack overflow while using extra stack.

- Other than **dtp-even-pc/dtp-odd-pc** in the trap vector.

- Uncorrectable ECC error when reading trap vector.

- Recursive uncorrectable ECC error.

- Page fault while dumping stack cache.

The processor will not respond to anything other than reset and init when halted. See the section "Revision 0 Traps for Processor Faults", page 301.

5.6 Trap Vector Layout

The trap vector is stored at physical addresses 1000000 through 1007777, and is basically partitioned as follows:

0000..3777	Arithmetic dispatch vector
4000..4377	Instruction exception vector
4400..4477	Interpreter function table
4500..4777	Reserved
5000..5077	Generic dispatch table
5100..5177	Miscellaneous exceptions
5200..7777	(Reserved for future expansion)

The arithmetic dispatch vector contains the exception handlers for those instructions defined to use the arithmetic dispatch. See the section "Arithmetic Traps", page 287.

The instruction exception vector contains the exception handlers for instructions that do not use the arithmetic dispatch vector. See the section "Instruction Exceptions", page 287.

The interpreter function table contains one entry per data type. When a **start-call** is given a data type not directly understood by the hardware, the contents of this table, indexed by the data type, are placed in the continuation register. See the section "Starting a Function Call", page 249.

The generic dispatch table contains one entry per data type. When the "instance" argument to **%message-dispatch** or **%generic-dispatch** is not an instance, the address of the flavor hash mask needed to do the method search is found by indexing into this table. See the section "Calling a Generic Function", page 277.

The miscellaneous exceptions are assigned as follows:

5100	Error trap
5101	Reset
5102	pull-apply-args
5103	Stack overflow
5104	Trace trap
5105	Preempt request
5106	Transport trap
5107	FEP-mode trap
5110	Low priority sequence break
5111	High priority sequence break
5112	Monitor trap
5113	Reserved for future use

5114	Generic-dispatch instruction
5115	Reserved for a fence word
5116	Message-dispatch instruction
5117	Reserved for a fence word
5120	Page not resident
5121	Page fault request
5122	Page write fault
5123	Uncorrectable memory error
5124	Bus error
5125-5177	Reserved for future use

5.7 Reset and Init

Reset and Init are exceptions invoked by pins of the same names on the processor chip. Reset is similar to a sequence break, and is used to return the processor to the FEP. Init is a no-holds-barred initialization of the machine, usually performed after power on.

Reset forces the processor to take an exception to fetch a new PC from the trap vector. It is up to software to save the machine state if it is desired to resume execution at the point the reset occurred.

Init initializes the processor hardware, and may abort outstanding memory accesses without completion, and so on. The PC is set to a fixed vma=pma address, 77400100, from which execution proceeds. See the section "Revision 0 Init PC", page 301.

5.8 Appendix: Comparison of 3600-Family and I-Machine Exception Handling

To be supplied with the next revision of this document.

Appendix A
Revision 0 Implementation Features

A.0.1 Revision 0 Implementation Memory Features

Revision 0 of the Ivory chip implements the following fields for ephemeral address:

Position	*Meaning*
<31:27>	00000 => ephemeral, otherwise non-ephemeral
<26:22>	ephemeral level number
<26:25>	ephemeral level group number
<21>	which half of the ephemeral level
<20:0>	word address within an ephemeral level

The comparison used Revision 0 in the inner loop of the PHT search is

```
(and (= (ldb %%pht0-vpn entry) vpn)
(= (ldb %%pht0-fault-request entry) 0))
```

A page-fault-request trap will not be taken if the %%pht0-fault-request bit is 1; this simply causes the entry not to match, eventually resulting in a page-not-resident trap instead.

The Revision 0 implementation always traps when executing an instruction from a page with transport-trap=1. Later implementations may be able to do an actual oldspace check in this case.

The Revision 0 implementation of Ivory does update a PHT entry by ORing the new bits in, but it does not use an interlocked bus read/write cycle.

Revision 0 of the Ivory chip sets the PC to VMA=PMA address 0 on receiving INIT.

The actual pht lookup algorithm for Revision 0 of the Ivory chip is:

```
(defun pht-lookup (vpn)
  (flet ((search-bucket (pht-offset)
           (loop repeat 4
               initially (setf (%block-address) (+ pht-base pht-offset))
               for entry = (%block-read)
               do (if (and (= (ldb %%pht0-vpn entry) vpn)
                           (= (ldb %%pht0-fault-request entry) 0))
                      (return-from pht-lookup
                                   (values entry (%block-read)))
```

```
                  (%block-read))
           finally
               ;; If at end of collision chain, fail.
               (when (= (ldb %%pht0-collision-chain entry) 1)
                 (return-from pht-lookup ()))))))
(search-bucket (logand (pht-hash vpn) pht-mask))
(loop for state = (pht-next vpn) then (pht-next state)
      do (search-bucket (logand (lsh state 3) pht-mask))))
```

A.0.2 Revision 0 Implementation Instruction Features

The following text describes characteristics of the Revision 0 implementation of the I-machine architecture.

Revision 0 %Allocate-list-block

Takes an instruction exception if *arg1* is **nil**.

Revision 0 %Allocate-structure-block

Takes an instruction exception if *arg1* is **nil**.

Revision 0 Aset-1

-- does not check for the high-order 24 (16) bits of **dtp-character** arguments being 0 when storing such arguments into 8(16)-bit arrays.

Revision 0 Binding Instructions

%restore-binding-stack, **unbind-n**, and the return instructions performing unbindings when the cleanup bit is set do not check for binding-stack underflow in Revision 0.

Revision 0 %Block-n-read-alu

-- performs overflow detection for exceptions but does not allow the memory operand to be in the stack cache. User programs must be sure that that the operands are not in the stack cache to insure proper operation. Does not take the shift mask specification from the DP OP register.

Revision 0 %Block-n-read-shift

-- will not work with ECC errors.

Revision 0 %Block-n-read-test

-- only implements the **eq** condition and the true sense. The alu-op field of the DP Op register must be loaded with "subtract" in order to use the instruction. Needs to have the oldspace condition added.

Revision 0 Branch and Loop Instructions

Revision 0 conditional branch instructions take an exception if the bottom eight bits of the offset are all 0 (as opposed to all 10 bits being 0). This effectively limits the branch distance to plus or minus 128, but the compiler could be smart about this. **branch** to the next instruction is the fastest no-op, except for skipping an instruction entirely via cdr-code sequencing.

Revision 0 Entry-rest-accepted

-- and **entry-rest-not-accepted** do not perform correctly when doing a *pull-apply-args* operation when the rest argument (or tail) is an item of type **dtp-list-instance** in the stack cache.

Revision 0 Fast-aset-1

-- does not check for the high-order 24 (16) bits of **dtp-character** arguments being 0 when storing such arguments into 8(16)-bit arrays.

Revision 0 Loop-decrement-tos

-- does not check overflow conditions. Uses **zerop** for the check rather than **plusp**.

Revision 0 Loop-increment-tos-less-than

-- does not check overflow conditions.

Revision 0 Opcode 57

-- jumps to a totally random address after pushing D.PC on the stack.

Revision 0 Numeric Operations

There will be no floating-point support in the Revision 0 chip.

Revision 0 Return-single

When the value disposition is "for value," the cdr code is the cdr code of the top of stack or it is **cdr-next** for **t** or **nil**.

Revision 0 Return-kludge

Sets the cdr codes of all values to **cdr-next**.

Revision 0 Stack-blt

-- sets the cdr code of the operand to **cdr-next**.

Revision 0 Stack-blt-address

-- sets the cdr code of the operand to **cdr-next**.

Revision 0 Unbind-n

Revision 0 of the Ivory chip, when unbinding, only checks the preempt-request bit when the trap mode is zero.

A.0.3 Revision 0 Implementation Function-Calling Features

The Ivory chip uses a scratchpad register to hold the value of %stack-frame-maximum size, which, with stack-cache-size being 128, is currently 119.

In Revision 0, if control-register.trace-pending would be set upon normal completion of an instruction, but the instruction pre-traps instead, control-register.trace-pending will incorrectly be set in the saved control-register image of the pre-trap handler.

In Revision 0, if control-register.instruction-trace is 1 at the beginning of a return instruction, and the return instruction restores a control-register with the trace-pending bit 0, control-register.trace-pending (and the correspond'ng trace-trap) may or may not be set at the completion of the return instruction.

In Revision 0, when a return instruction with value disposition return restores a control-register with trace-pending set, the trace-trap will be lost. Only the trace-pending bit of the last control-register restored is significant.

Revision 0 of the Ivory chip cannot handle faults during stack cache refill.

A.0.4 Revision 0 Implementation Exception Handling Features

The following text describes characteristics of the Revision 0 implementation of the I-machine architecture.

Revision 0 Memory Exceptions

The revision 0 implementation of the Ivory chip takes an uncorrectable-memory-error trap when it should take a bus-error trap.

Revision 0 Sequence Breaks

Sequence breaks vector through only one place in microcode. That microcode then examines a register to decide whether the sequence break is high-priority or low-priority. This is only visible to the user if a high-priority sequence break is requested and, before the microcode can execute the start of the sequence break microcode, the high-priority sequence break goes away. In this case, because of the order of polling in the microcode, the chip will take a low-priority sequence break (although the low-priority bit might not be set in the preempt register).

Revision 0 Traps for Processor Faults

The Revision 0 chip only halts for a stack overflow while using extra-stack mode. It does not halt for any of the other reasons listed in the Processor Faults section in the Exceptions chapter. The architectural issues of processor-fault handling have not yet been resolved.

The Revision 0 chip does not respond to reset when halted.

Page faults currently do not work, since it is not possible for an instruction to look like it has finished without actually transferring control to the next instruction's microcode. This means that either the entire control stack must be wired down and scavenged, or that the trap-on-exit bit must be used to cause a trap when more stack must be wired down. At the same time that the stack is wired down, it must be transported for proper operation.

Revision 0 Unbinding

See the section "Revision 0 Unbind-n".

Revision 0 Init PC

Revision 0 Init sets the PC to vma=pma 0.

Appendix B
Summary of Omitted 3600 Instructions

;;; These are supported only if the floating point chip supports them
%CONVERT-SINGLE-TO-FIXNUM
%DOUBLE-FLOATING-ABS, %DOUBLE-FLOATING-ADD, %DOUBLE-FLOATING-COMPARE,
%DOUBLE-FLOATING-DIVIDE, %DOUBLE-FLOATING-MINUS,
%DOUBLE-FLOATING-MULTIPLY, %DOUBLE-FLOATING-SCALE, %DOUBLE-FLOATING-SUB
FLOAT-OPERATING-MODE, FLOAT-OPERATION-STATUS,
SET-FLOAT-OPERATING-MODE, SET-FLOAT-OPERATION-STATUS

Will not be implemented:

FOLLOW-CELL-FORWARDING **%memory-read-address data-read** or **bind-read**
FOLLOW-STRUCTURE-FORWARDING **%memory-read-address struct-offset**
LOCATION-BOUNDP (/= (**%data-type** (**%memory-read** bind-read) **dtp-null**)
%P-STRUCTURE-OFFSET **%memory-read-address** followed by **%pointer-plus**
%P-CONTENTS-AS-LOCATIVE **%memory-read-address** followed by **%set-tag**
%P-CONTENTS-OFFSET (**cdr** (**%p-structure-offset** ...)

Unclassified:
NOT - Implemented by **type-member**
LONG-BRANCH-IMMED
PUSH-MICROCODE-ESCAPE-CONSTANT
%DRAW-STRING-STEP, %BITBLT-DECODE-ARRAYS, %BITBLT-LONG,
%BITBLT-LONG-ROW, %BITBLT-LONG-ROW-BACKWARDS, %BITBLT-SHORT,
%BITBLT-SHORT-ROW, %DRAW-LINE-LOOP, %DRAW-STRING-LOOP,
%DRAW-TRIANGLE-SEGMENT, SOFT-MATTE-DECODE-ARRAYS, SOFT-MATTE-INTERNAL

Lisp Instructions:
%SAVE-BINDING-STACK-LEVEL
  - Implemented as an internal register
CONS
NCONS

Function-Calling instructions:
TAKE-ARG, TAKE-M-REQUIRED-N-OPTIONAL-ARGS,
TAKE-M-REQUIRED-N-OPTIONAL-ARGS-REST, TAKE-N-ARGS, TAKE-N-ARGS-REST,
TAKE-N-OPTIONAL-ARGS, TAKE-N-OPTIONAL-ARGS-REST, TAKE-REST-ARG,

Fortran Array Instructions: (This might make it in, but I think we will be too tight on "B" memory locations and microcode)
FTN-ALOC-1, FTN-AREF-1, FTN-ASET-1
FTN-DOUBLE-ALOC-1, FTN-DOUBLE-AREF-1, FTN-DOUBLE-ASET-1
FTN-LOAD-ARRAY-REGISTER

Low-Level Hardware:
%AUDIO-START, %CHECK-PREEMPT-PENDING, %CLEAR-CACHES,
%CLEAR-INSTRUCTION-CACHE, %DISK-START,
%RESUME-MAIN-STACK-BUFFER, %FUNCALL-IN-AUXILIARY-STACK-BUFFER,
%FEP-DOORBELL,
%FIXNUM, %FLONUM
%GC-MAP-WRITE, %GC-TAG-READ, %GC-TAG-WRITE,
%MAP-CACHE-WRITE
%METER-OFF, %METER-ON
%MICROSECOND-CLOCK
%NET-WAKEUP
%NUMERIC-DISPATCH-INDEX
%PHTC-READ, %PHTC-SETUP, %PHTC-WRITE
%PHYSICAL-ADDRESS-CACHE,
%REFERENCE-TAG-READ, %REFERENCE-TAG-WRITE
%SCAN-FOR-ECC-ERROR, %SCAN-FOR-EPHEMERAL-SPACE, %SCAN-FOR-OLDSPACE
%SCAN-GC-TAGS, %SCAN-REFERENCE-TAGS, %SET-PREEMPT-PENDING,
%TAPE-WAKEUP, %UNSYNCHRONIZED-DEVICE-READ

;;; Replaced by Bars
%BLOCK-GC-COPY, %BLOCK-TRANSPORT
%BLOCK-STORE-CDR-AND-CONTENTS, %BLOCK-STORE-TAG-AND-POINTER

;;; Open coded
POP-MULTIPLE, POP-MULTIPLE-SAVE-1, POP-MULTIPLE-SAVE-N,
POP-MULTIPLE-SAVE-MULTIPLE
POP-N-SAVE-MULTIPLE, APPEND-MULTIPLE-GROUPS, PUSH-FROM-BEYOND-MULTIPLE
FIXUP-TOS (MOVEM SP|0)
%MAKE-POINTER-IMMED-OFFSET

Appendix C
Notes on I-Machine Architecture History

Data-Types Chapter -- Representations of Arrays

A prototype of the precise algorithm to be used when accessing an indirect array, using the 3600 array format instead of this array format, can be found in the file V:>Moon>IMach>3600>array.lisp. This was translated from the existing, working 3600 microcode.

Some static analysis of arrays, in a system 311 world that been used for a week:

> 99.65% of all arrays are one-dimensional.
>
> 2-dimensional and 3-dimensional arrays exist; no higher-dimensional or 0-dim arrays.
>
> The average size of an array is 38 words.
>
> There is no category of arrays whose average size is larger than will fit in 15 bits; unfortunately I didn't measure the size distribution of arrays directly, so I don't know the percentage of arrays whose size will not fit in 15 bits, but it must be very small.
>
> All array types are used at least once.
>
> The maximum leader length seen is 38 elements.
>
> Unfortunately I didn't measure what fraction of arrays are displaced.
>
> --Moon

The longest array-leader observed was 38 elements, so a maximum limit of 255 elements should not be restrictive. The maximum on the 3600 is 1023.

The leader header uses **dtp-header-p** rather than **dtp-header-i** because there were more spare header-type codes available for that type of header.

More information from Rel 6.1

99.54% of the arrays are one-dimensional, of which 99.54 are direct (not displaced). Totals: 453049 arrays, 450961 one-dim, 448900 direct one-dim, 2061 indirect one-dim. The distribution of the LOG2(LENGTH) is as follows:

```
0:    12493;
1:    16181;
2:    35701;
3:   130120;
4:    93601;
5:    85780;
6:    44053;
```

```
  7:    26594;
  8:     2447;
  9:      873;
 10:      615;
 11:      324;
 12:       48;
 13:       25;
 14:       20;
 15:       12;
 16:        6;
 17:        3;
 19:        1;
 20:        3.

20.33% of those arrays have a leader
The distribution of the LOG2(LEADER-LENGTH) is as follows:
  0:        2;
  1:     7962;
  2:     5870;
  3:     3428;
  4:    73835;
  5:      181;
  6:        1.
```

Data-Types Chapter -- Representations of Compiled Functions

Not only does this (using the cdr code 1 as a fence) avoid loading the instruction cache with extraneous words from functions other than the one being executed, but more importantly it avoids a subtle bug involving fetchahead past the free-pointer for allocation of compiled code, after a sequence of timing coincidences has left words there containing valid data types for instructions, The bug is that obsolete data could get into the instruction cache and not get cleared out when a new function was created at the same address.

Note that the design is intended to put the function cell and the entry instruction both on the same page and in the same cache line, minimizing the cost of indirecting through a function cell. The loader may want to insert extra words to keep compiled functions aligned on appropriate boundaries so that the function cell and entry instruction always fall into the same cache line, if we have a cache.

Data-Types Chapter -- Instruction Representation

This scheme, different from the 3600, is designed to eliminate the constants/external-reference table in a compiled function and thereby to enable prefetching of such data through the normal instruction pipeline. This saves time and simplifies the hardware by eliminating an addressing mode. H says the average number of references per constant is small enough that this actually saves space, compared to the 3600. In cases where there are many calls to the same function or references to the same constant, the compiler can attempt to encache it in a local variable.

Data-Types Chapter -- Representation of Physical Addresses:

BARs need to store 33 bits, the 33rd bit being the dtp-physical-address'ness of the pointer field.

The BAR incrementer only increments the pointer field; it leaves the dtp-physical-address'ness alone.

The FAST-AREF/ASET-1 ucode/hardware adds the offset to the pointer field of the array register base address slot and preserves the dtp-physical-address'ness on the way to the BAR.

The input to the map cache now has three possibilities instead of two:

mapped virtual address
vma=pma virtual address
dtp-physical-address physical address

The possibilities for cache-control output lines/signals/meanings for each of the above should be discussed separately. The thing I think we all agree on is that data referenced with dtp-physical-addresses are never cached.

Memory Chapter -- Wired Addresses:

The 3600-family feature where some portion of virtual address space defined by a control register (%wired-virtual-address-high) is mapped to a contiguous portion of unmapped address space defined by another control register(%wired-physical-address-low) is eliminated, to simplify the hardware. This reduces configuration flexibility by requiring that some portion of unmapped address space starting at a fixed physical address, presumably 0, must always contain working memory; this is not a problem if that memory is packaged right on the CPU board. The permanently-wired programs and data that on the 3600 are stored in virtual address space below %wired-virtual-address-high will instead be stored at physical addresses.

Memory Chapter -- Pages:

There has been a lot of discussion about increasing the page size. There are a lot of variables involved, including:

- Page tables (PHT and MMPT, not SMPT or ESRT) can be smaller for larger page sizes. This isn't that large an effect -- 256 word pages yield about 1.9% overhead, 512 yields 1.0%, and 1024 yields 0.4%. All of these figures are tolerable, and well below a number of comparable (sic) systems.

- Overhead of managing page tables is lower with larger page sizes. The only place where this is significant is in creating new pages. However, reorganization of the code and algorithms can compensate. In fact, there is very significant progress to be made here before the effect of page size gets out of the noise.
- Larger page sizes mean there are more untranslated bits available (presumably the processor can spit these out 1/2 cycle earlier) for data caches and clever dynamic ram organizations to take advantage of.
- A given size map cache describes more storage if the page size is larger. However, according to the literature, the primary contributors to map cache performance are number of entries, associativity, and replacement algorithm (in roughly that order), with page size a distant fourth.
- The instruction prefetcher faults crossing page boundaries.

That's about it for the pros. On the con side are:

- Larger page sizes reduce primary memory utilization.
- Larger page sizes reduce the ability of the EGC to isolate ephemeral references. The EGC keeps track of ephemeral references on a per-page basis -- any page thought to contain such references needs to be scanned. If there is poor locality of such references, the amount of scanning required per garbage collection will increase proportional to the page size. This is an important effect, since as main memory sizes increase the amount of EGC scanning increases, but the memory bandwidth and processor speed will stay relatively const ınt.

Some perfunctory analysis indicated that the reduced primary memory utilization of larger page sizes was a very significant effect, and on that evidence (and conservatism in general) the page size was left at 256 words.

In Release 6.0, the function with the largest number of required+optional arguments is TV:DRAW-TRIANGLE-SETUP, which takes 15 arguments.

Appendix D
Hints for Software Developers

Data-Types Chapter

Double-precision Floating-point Representation: Similar to the 3600, except that a cons is used instead of a structure to eliminate the overhead of a header word.

Note that the two halves of the number are being stored in arguably the wrong order, since the least-significant bits of the fraction should be first. This is consistent with the 3600. The real basis for deciding should be the order that data are fed into the double-precision floating-point processor chip, if there is one.

Memory Chapter

The system must ensure, or arrange, that there are never any safeguarded objects in about-to-be oldspace. The 3600 solves this by simply not flipping that region, but that might not be easy on this machine, especially ephemeral space.

Memory Chapter

The fields in an MMPT entry are: (these fields aren't known to hardware)

```
- bits 31:8 -- VPN -- the virtual page number now in this physical
                page, -1 if invalid.
- bit 7 -- FLUSHING -- 1 => VPN will change when disk write completes.
- bit 6 -- WRITE LOCK -- 1 => don't reassign the page (being written
                to disk?).
- bit 5 -- STACK -- 1 => this page is held in main memory because
                it's a stack.
- bit 4 -- spare
- bits 3:0 -- status code, defined by software (3600 uses 10 codes)
```

Explanation of PHT.PENDING: Hardware does not look at the PHT.PENDING bit. If it is set, PHT.FAULT-REQUEST is also set, by software convention. PENDING is set when a page is being read in from disk, but first another page has to be written out from the memory page frame the new page is going to occupy. In this situation, there are two PHT entries pointing to the same physical page. Each of them has FAULT REQUEST set, and one of them also has PENDING set. The MMPT entry for that physical page contains the information needed by the page-fault trap-handling software to figure out what is going on.

Compromises: AGE bits would really rather be in the MMPT. Setting of EPHEMERAL REFERENCE bits would really rather be in parallel with memory access. I don't think either of these will have a significant effect on performance, in practice.

Instruction Chapter -- rgetf

Additional instructions can be used together with **rgetf** to implement the **zl:get**, **zl:putprop**, **cl:get**, and **cl:getf** functions and to implement **&key** arguments. **rgetf** is often followed by either an instruction to pop the second value or a branch instruction that tests the second value and if it is **nil** pops both values and goes to code to substitute a default value.

```
(get loc arg2)
```

should be

```
(getf (location-contents
...) arg2)
```

cl:get is

```
push arg2
push symbol
type-member-n-no-pop                        ;symbolp
branch-false .
%pointer-plus 4
%memory-read data-read
rgetf sp-pop
set-sp-to-address SP|-1
...
```

rgetf stands for "reverse getf" because the argument order is reversed from **cl:getf**.

Instruction Chapter -- logtest

logtest is commutative, so that if there is a small integer, **logtest** should commute it to the second argument. -- DCP The hardware has no idea about commutativity. Software probably has to do this. -- BEE

Instruction Chapter -- pop

The file V:>moon>imach>pop.text has more information about stack-popping instructions, including stack-blt.

D.0.1 Stack Groups on the I Machine

A stack group is the object of computation. It contains the memory image of a process. This includes many things, all of which eventually need to be enumerated. For now, the list includes the following:

- Control Stack

Instruction Chapter -- Mapped Access to Self

The instructions for mapped accesses to self check that the argument *I* is within the bounds of the mapping table. If it is not, a trap occurs. The bounds check is performed by fetching the array header of the mapping table, assuming it is a short-prefix array, and comparing *I* against the array-short-length field.

Implementation note: it is useful to cache the array header to avoid making a memory reference to get it every time. For an example of how to do this using two scratchpad locations and one cycle of overhead, see the 3600 microcode.

Instruction Chapter -- rgetf

Additional instructions can be used together with **rgetf** to implement the **zl:get**, **zl:putprop**, **cl:get**, and **cl:getf** functions and to implement **&key** arguments. **rgetf** is often followed by either an instruction to pop the second value or a branch instruction that tests the second value and if it is **nil** pops both values and goes to code to substitute a default value.

```
    (get loc arg2)
```

should be

```
    (getf (location-contents
    ...) arg2)
```

cl:get is

```
     push arg2
     push symbol
     type-member-n-no-pop                      ;symbolp
     branch-false .
     %pointer-plus 4
     %memory-read data-read
     rgetf sp-pop
     set-sp-to-address SP|-1
     ...
```

rgetf stands for "reverse getf" because the argument order is reversed from **cl:getf.**

Instruction Chapter -- logtest

logtest is commutative, so that if there is a small integer, **logtest** should commute it to the second argument. -- DCP The hardware has no idea about commutativity. Software probably has to do this. -- BEE

Instruction Chapter -- pop

The file V:>moon>imach>pop.text has more information about stack-popping instructions, including stack-blt.

  - Control Stack Base
  - Control Stack Pointer
  - Control Stack Limit
  - Control Stack Extra Limit
  - Control Stack Wired Low
- Frame Pointer
- Local Pointer
- Binding Stack
  - Binding Stack Base
  - Binding Stack Pointer
  - Binding Stack Limit
- Data Stack
  - Data Stack Base
  - Data Stack Pointer
  - Data Stack Limit
- Catch Block Pointer
- PC
- Control Register
- Continuation Register
- Floating Point State
  - Mode - rounding, underflow-to-0, ...
  - Status - sticky-over/underflow, ...

Several of the stack group registers are not hardware registers, just software slots in the stack group.

Function-Calling Chapter

Because the handling of Multiple and Return value dispositions is similar, the **return-single** and **return-multiple** instructions can be implemented by starting with a four-way dispatch to these cases:

1. Cleanup Bits non-zero – Perform the cleanup and then retry the instruction.
2. Value Disposition = Effect – Just return without worrying about the values.
3. Value Disposition = Value – Just return the first value.
4. Value Disposition = Multiple or Return – Take complex actions.

(But KHS doesn't believe it is actually implemented that way.)

Function-Calling Chapter -- Stack-Group Switching

Existing instructions have the following capabilities:

- ability to do appropriate special memory references, using block-read/write
- ability to do necessary cdr-code hacking
- ability to dump the entire stack cache into memory
- ability to load a new stack into an empty stack cache
- ability to read and write all internal processor and coprocessor registers
- that are part of the stack group context
- ability to inhibit all traps and interrupts while the stack cache control registers are in an inconsistent state
- ability to inhibit process preemption during the whole operation this is done by setting a software flag respected by the preempt

Other instruction assumptions:

- bind_read_no_monitor bit in block-read instruction
- no_increment bit in block-read instruction prevents incrementing BAR
- preserve_cdr bit in block-read instruction inhibits setting cdr of result to 0 (this is already in the rev -2 spec)
- when block-read follows an invisible pointer, it updates the BAR
- merge-cdr-nopop instruction: cdr(operand) ← cdr(top-of-stack), no change to

SP this could be done with **%p-tag-ldb** and **%p-tag-dpb** but it would be much slower.

Note that it is possible for the cdr code of the bound location to change while it is bound, which is why the merge-cdr-nopop instruction is required instead of simply rewriting all 40 bits with the value saved in the binding stack.

Alternatively, to the assumption that memory locations in the stack write through to main memory, a specific instruction could be provided to dump the entire stack cache, since the processor already knows how to dump parts of the stack cache when it fills up.

Exceptions Chapter

Floating exceptions need to be covered as well. Floating overflow and underflow always trap. Floating inexact needs a software writable enable to stop it from trapping, since it occurs so frequently. Floating divide by 0 always traps. Floating invalid operation always traps. The trap handlers can maintain sticky bits for all these exceptions.

Be especially careful about non-commutative instructions with pop-stack address mode [for traps].

Exceptions Chapter

There are four kinds of recursive traps to fear:

Page fault on a stack page. This is avoided by requiring that all pages of the control stack of a stack group, up to the extra-stack limit, be resident in main memory before control can enter the stack group. The Revision 0 chip does not do this, so stack get completely wired. Other implementations do this, so page faults on running stacks cause the pages to get wired, but are otherwise pretty normal. (Just the structure defining the stack group would be stored, not the actual stacks). The paging system has to be careful about evicting stack pages or clearing their write-permission bits. It may not do this to the current stack group, and if it does it to another stack group it must set a bit in the SG that will cause a trap if control attempts to enter it. Note that the stack-limit register values in a stack group can be set to less than their maximum values, to save on main memory. Then if the limited stack available overflows the stack overflow handler can wire down additional pages and increase the stack limits.

Virtual address translation failure on a resident stack page. On the 3600 if a virtual reference is satisfied by neither the map cache nor the PHTC, it traps to macrocode. Occasionally a trap to macrocode will occur for a resident page, merely to translate its virtual address. This cannot be tolerated on the IMach, so the page translation tables must be designed so that the hardware and microcode can always find the physical address of a resident page. This has the additional advantage that spilling of a cache into main memory can never cause a page fault (assuming of course that when a page is evicted its contents are first removed from any caches that may still contain them.)

Page fault while dumping the contents of a cache to make room for new data. This cannot happen as explained just above.

If uncorrectable ECC error is a trap, then this can also be recursive.

It is not actually necessary to wire down an entire stack, just the top part of the stack that is being used. When control unwinds to earlier frames in the stack, a page fault will occur while trying to reload the stack cache from virtual memory. It should be easy to arrange for this page fault to be handled in the part of the stack that is still resident. The control-register.trap-on-exit bit could also be set in the bottommost frame in the resident portion of the stack, so that the trap would occur before the page fault. In this way main memory would be used in the same way that A-memory is used for a stack buffer in the 3600.

Appendix E
Notes on Future Implementations of the Ivory Chip

Data-Types Chapter -- Array Representations

Non-word-aligned array registers can be optimized by an additional 5-bit adder and a special carry input to the main adder.

Instruction Chapter -- minusp

Small ratios might also be handled by microcode since they can be compared on the same basis as fixnums -- if ratios are canonicalized to have the sign bit in the numerator. Same test for all of them, bitwise? except for floating point, not-a-numbers. -0.0 is not minusp, so bit test fails.

Instruction Chapter -- Instance Variable Accessors

All of the instance-variable accessing instructions could take an sp-pop argument as an alternative to an immediate. This issue needs to be reviewed when the microcode is written. **%instance-loc, %instance-ref, %instance-set** could be flushed. Removing them would slow the specific kinds of instance-variable accesses that use these instructions by a factor of 2 or 3. Most instance-variable accesses use the mapped or ordered instruction described earlier.

Function-Calling Chapter

This is not done in Revision 0 of the chip, but might ought to be:

"The first thing finish-call does is to check for Apply = 1 but the top word on the stack is **nil** (an empty list). In this case it pops the stack and clears its copy of the Apply bit, turning into a normal call. This canonicalization simplifies the argument match-up procedure described later."

Function-Calling Chapter -- Calling a Generic Function

A reasonable optimization would be to avoid the memory references to fetch the trap-vector element and to fetch the **%generic-dispatch** instruction, since calling of generic functions is so common. (It would save 2 memory references out of 5, and avoid perturbing the I cache.) The **%generic-dispatch** instruction could be fed magically into the instruction pipeline, and the PC could be set to a constant value that is architecturally required to be the address of a memory location containing a **%generic-dispatch** instruction; this location will be referenced if the **%generic-dispatch** traps (for example, for a page fault) and has to be retried.

Future hardware might contain a special-purpose cache used by the generic-dispatch instruction to speed repeated lookups with the same generic function and instance.

Appendix F
Instruction Classifications for Packed Instructions

F.1 Formats

The two major classifications of packed instructions are operand-from-stack format and 10-bit-immediate format. These are further broken down into various subclasses. Additional information in the opcode field is indicated with a "*". Instructions in the operand-from-stack format always have an operand-specifier in their lower 10 bits. Instructions in the 10-bit-immediate format have different uses for their lower 10 bits. Fields in the 10-bit immediate are indicated by a "-".

F.2 Operand-from-stack Instructions

* Unary Instruction (otherwise ≥2 args)
* Signed Immediate (otherwise unsigned)

unary/signed (14 opcodes)
car, cdr, endp, plusp[2], minusp[1], zerop[1], setup-1d-array, setup-force-1d-array, start-call, bind-locative, %restore-binding-stack, %ephemeralp, %tag, %jump

unary/unsigned (12 opcodes)
unary-minus[1], push, push-n-nils, push-address-sp-relative, return-multiple, return-kludge, take-values[3], unbind-n[2], push-instance-variable[2], push-address-instance-variable[2], push-instance-variable-ordered[2], push-address-instance-variable-ordered[2]

unary/address (11 opcodes)
set-to-car, set-to-cdr, set-to-cdr-push-car, increment, decrement, push-address[4], set-sp-to-address[3], set-sp-to-address-save-tos[3], %pointer-increment, %set-cdr-code-1, %set-cdr-code-2

[2] Arithmetic dispatching.

[3] Instructions which are only defined for an immediate argument could be in either operand-from-stack or 10-bit-immediate format.

[4] Not all address-operand instructions modify their argument.

not-unary/signed (31 opcodes)

rplaca, rplacd, rgetf, member, assoc, multiply[1], quotient[1], ceiling[1], floor[1], truncate[1], round[1], remainder[1], rational-quotient[1], max[1], min[1], logand[1], logior[1], logxor[1], ash[1], rot, lsh, %multiply-double, %lshc-bignum-step, stack-blt, bind-locative-to-value, %pointer-plus, %pointer-difference, store-conditional, %memory-write, %p-store-contents

not-unary/unsigned (24 opcodes)

add[1], sub[1], %32-bit-plus, %32-bit-difference, %add-bignum-step, %sub-bignum-step, %multiply-bignum-step, %divide-bignum-step, aref-1, aset-1, aloc-1, array-leader, store-array-leader, aloc-leader, pop-instance-variable[2], movem-instance-variable[2], pop-instance-variable-ordered[2], movem-instance-variable-ordered[2], %instance-ref, %instance-set, %instance-loc, %allocate-list-block, %allocate-structure-block, %set-tag

not-unary/address (6 opcodes)

pop, movem, stack-blt-address[3], fast-aref-1[3], fast-aset-1[3], %merge-cdr-no-pop

Binary-Predicate Subformat
* no-pop arg1

not-unary/signed (12 opcodes)

eq, eq-no-pop, eql[1], eql-no-pop[1], equal-number[1], equal-number-no-pop[1], greaterp[1], greaterp-no-pop[1], lessp[1], lessp-no-pop[1], logtest[1], logtest-no-pop[1]

not-unary/unsigned (2 opcodes)

%unsigned-lessp, %unsigned-lessp-no-pop

BAR Subformat
* BAR number

unary/signed (4 opcodes)

%block-n-write

unary/address (4 opcodes)

%block-n-read-alu

Lexical Subformat
* variable number

unary/signed (8 opcodes)

push-lexical-var-n

not-unary/signed (16 opcodes)
pop-lexical-var-n, movem-lexical-var-n

F.3 10-bit-immediate Instructions

Type-member Subformat
* pop arg
* field number (2 bits)
- field number (2 bits) <9:8>
- type set <7:0>

unary (8 opcodes)
type-member-n, type-member-n-no-pop

Branch Subformat
* condition false
* no-pop condition
* and extra pop
* else extra pop
- branch offset <9:0>

(16 opcodes) branch-true, branch-false, branch-true-no-pop, branch-false-no-pop, branch-true-else-no-pop, branch-false-else-no-pop, branch-true-and-no-pop, branch-false-and-no-pop, branch-true-and-extra-pop, branch-false-and-extra-pop, branch-true-else-extra-pop, branch-false-else-extra-pop, branch-true-and-no-pop-else-nopop, branch-false-and-no-pop-else-nopop, branch-true-extra-pop, branch-false-extra-pop

Loop Subformat
- branch offset <9:0>

(3 opcodes) branch, loop-decrement-tos, loop-increment-tos-less-than

Byte-field Subformat
- field width <9:5>
- field starting position <4:0>

unary (4 opcodes)
ldb, char-ldb, %p-ldb, %p-tag-ldb

not-unary (4 opcodes)
dpb, char-dpb, %p-dpb, %p-tag-dpb

BAR Subformat
- * BAR number
- memory cycle type <9:6>
- fixnum only <5>
- set cdr-next or invert test <4>
- last word <3>
- no increment <2>
- test select <1:0>

(12 opcodes) %block-n-read, %block-n-read-shift, %block-n-read-test

Finish-call Subformat
- * apply
- value disposition <9:8>
- number of arguments <7:0>

(4 opcodes) finish-call-n, finish-call-n-apply, finish-call-tos, finish-call-tos-apply

Entry Subformat
- * rest accepted
- min args <7:0>
- max args <25:18>

(2 opcodes) entry-rest-not-accepted, entry-rest-accepted

Return Subformat
- return value select <1:0>

(1 opcode) return-single

Catch-open Subformat
- value disposition <7:6>
- catch/unwind-protect <0>

(1 opcode) catch-open

Memory-read Subformat
- memory cycle type <9:6>
- fixnum only <5>
- set cdr-next <4>

(2 opcodes) %memory-read, %memory-read-address

Internal-Register Subformat

- internal register address <9:0>

  (2 opcodes) %read-internal-register, %write-internal-register

Coprocessor Subformat (2 opcodes)
- coprocessor address <9:0>

  (2 opcodes) %coprocessor-read, %coprocessor-write

Unused-Immediate Subformat (6 opcodes)
locate-locals, catch-close, %generic-dispatch, %message-dispatch,
%check-preempt-request, no-op, %halt

F.4 Encodings

unary	00 0__	00	car	10	start-call
signed		01	cdr	11	%jump
		02	endp	12	%tag
		03	setup-1d-array	13	
		04	setup-force-1d-array	14	
		05	bind-locative	15	
		06	%restore-binding-stack	16	
		07	%ephemeral-p	17	
		20	push-lexical-var-0	30	%block-0-write
		21	push-lexical-var-1	31	%block-1-write
		22	push-lexical-var-2	32	%block-2-write
		23	push-lexical-var-3	33	%block-3-write
		24	push-lexical-var-4	34	zerop*
		25	push-lexical-var-5	35	minusp*
		26	push-lexical-var-6	36	plusp*
		27	push-lexical-var-7	37	
	00 1__	00	type-member-0	10	locate-locals
		01	type-member-1	11	catch-close
		02	type-member-2	12	%generic-dispatch
		03	type-member-3	13	%message-dispatch
		04	type-member-0-no-pop	14	%check-preempt-request
		05	type-member-1-no-pop	15	
		06	type-member-2-no-pop	16	no-op
		07	type-member-3-no-pop	17	%halt
		20	branch-true	30	branch-false

	21	branch-true-else-extra-pop	31	branch-false-else-extra-pop
	22	branch-true-and-extra-pop	32	branch-false-and-extra-pop
	23	branch-true-extra-pop	33	branch-false-extra-pop
	24	branch-true-no-pop	34	branch-false-no-pop
	25	branch-true-and-no-pop	35	branch-false-and-no-pop
	26	branch-true-else-no-pop	36	branch-false-else-no-pop
	27	branch-true-no-pop-extra-pop	37	branch-false-no-pop-extra-pop
unary 01 0__	00	push	10	push-instance-variable
unsigned	01	push-n-nils	11	push-address-instance-var
	02	push-address-sp-relative	12	push-instance-var-ordered
	03		13	push-address-instance-var-or
	04	return-multiple	14	unary-minus*
	05	return-kludge	15	return-single
	06	take-values	16	%memory-read
	07	unbind-n	17	%memory-read-address
	20	%block-0-read	30	%block-0-read-test
	21	%block-1-read	31	%block-1-read-test
	22	%block-2-read	32	%block-2-read-test
	23	%block-3-read	33	%block-3-read-test
	24	%block-0-read-shift	34	finish-call-n
	25	%block-1-read-shift	35	finish-call-n-apply
	26	%block-2-read-shift	36	finish-call-tos
	27	%block-3-read-shift	37	finish-call-tos-apply
unary 01 1__	00	set-to-car	10	push-address
address	01	set-to-cdr	11	set-sp-to-address
	02	set-to-cdr-push-car	12	set-sp-to-address-save-tos
	03	increment	13	
	04	decrement	14	%read-internal-register
	05	%pointer-increment	15	%write-internal-register
	06	%set-cdr-code-1	16	%coprocessor-read
	07	%set-cdr-code-2	17	%coprocessor-write
	20	%block-0-read-alu	30	ldb
	21	%block-1-read-alu	31	char-ldb
	22	%block-2-read-alu	32	%p-ldb
	23	%block-3-read-alu	33	%p-tag-ldb
	24		34	branch
	25		35	loop-decrement-tos
	26		36	entry-rest-accepted
	27		37	entry-rest-not-accepted

not-	10 0__	00	rplaca	10	remainder*
unary signed		01	rplacd	11	rational-quotient*
		02	multiply*	12	min*
		03	quotient*	13	max*
		04	ceiling*	14	
		05	floor*	15	logand*
		06	truncate*	16	logxor*
		07	round*	17	logior*
		20	rot	30	%pointer-plus
		21	lsh	31	%pointer-difference
		22	%multiply-double	32	ash*
		23	%lshc-bignum-step	33	store-conditional
		24	stack-blt	34	%memory-write
		25	rgetf	35	%p-store-contents
		26	member	36	bind-locative-to-value
		27	assoc	37	
	10 1__	00	pop-lexical-var-0	10	movem-lexical-var-0
		01	pop-lexical-var-1	11	movem-lexical-var-1
		02	pop-lexical-var-2	12	movem-lexical-var-2
		03	pop-lexical-var-3	13	movem-lexical-var-3
		04	pop-lexical-var-4	14	movem-lexical-var-4
		05	pop-lexical-var-5	15	movem-lexical-var-5
		06	pop-lexical-var-6	16	movem-lexical-var-6
		07	pop-lexical-var-7	17	movem-lexical-var-7
		20	equal-number*	30	eq
		21	lessp*	31	
		22	greaterp*	32	
		23	eql*	33	logtest*
		24	equal-number-no-pop*	34	eq-no-pop
		25	lessp-no-pop*	35	
		26	greaterp-no-pop*	36	
		27	eql-no-pop*	37	logtest-no-pop*
not-	11 0__	00	add*	10	aset-1
unary unsigned		01	sub*	11	%allocate-list-block
		02	%32-bit-plus	12	aref-1
		03	%32-bit-difference	13	aloc-1
		04	%add-bignum-step	14	store-array-leader
		05	%sub-bignum-step	15	%allocate-structure-block

		06	%multiply-bignum-step	16	array-leader
		07	%divide-bignum-step	17	aloc-leader
		20	pop-instance-variable	30	
		21	movem-instance-variable	31	%unsigned-lessp
		22	pop-instance-var-ordered	32	
		23	movem-instance-var-ordered	33	
		24	%instance-ref	34	
		25	%instance-set	35	%unsigned-lessp-no-pop
		26	%instance-loc	36	
		27	%set-tag	37	
not-	11 1__	00	pop	10	fast-aref-1
unary	address	01	movem	11	fast-aset-1
		02	%merge-cdr-no-pop	12	stack-blt-address
		03		13	
		04		14	
		05		15	
		06		16	
		07		17	
		20		30	dpb
		21		31	char-dpb
		22		32	%p-dpb
		23		33	%p-tag-dpb
		24		34	
		25		35	loop-increment-tos-<
		26		36	catch-open
		27		37	

Index