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.

****

Introduction

This chapter defines all the instructions executed by the I machine. The instructions are grouped according to their function. The end matter of this manual contains indexes to the instructions organized alphabetically, by opcode, and by instruction format. An appendix to the manual 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 formats, including control stack addressing modes, instruction sequencing, internal registers, types of memory references, and top-of-stack register effects.

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".

The following table summarizes I-machine instruction formats and lists the data types in each class.

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
	DTP-LEXICAL-CLOSURE	32
	DTP-DYNAMIC-CLOSURE	33
	DTP-COMPILED-FUNCTION	34
	DTP-GENERIC-FUNCTION	35
	DTP-SPARE-OBJECT-1	36
	DTP-SPARE-OBJECT-2	37
	DTP-SPARE-OBJECT-3	40
	DTP-SPARE-OBJECT-4	41
	DTP-SPARE-OBJECT-5	42
	DTP-CHARACTER	43
	DTP-SPARE-OBJECT-6	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

The following paragraphs describe these formats.

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 and address field as shown in Figure INSTRUCTION-FORMATS. This kind of instruction starts a function call.

[Figure caption: 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 INSTRUCTION-FORMATS, is

Bits	*Meaning*
<39:38>	Sequencing code = "add 2 to PC"
<37:36>	**dtp-packed-instruction**
<35:28>	Opcode of second half word, may be unused
<27:26>	Addressing mode of second half word, may be 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.

****

Notes: When a rest arg is not wanted and no rest arg has been supplied, entry will take 2 clocks if all the optionals (which may of course be 0) are defaulted and 4 clocks if only some are defaulted. The cases for a rest arg wanted are more involved. When a rest arg has been supplied, entering at N-args minus min-args should take 3/5 clocks for all/some defaulted.

The additional hardware support for entry is the ability to read CR.argument-size as a fixnum and E.instruction<25:18> as a fixnum. The too-few/many calculations are done in the main data path, and the PC adjustment in the PC adder (which also already does offset=0 detection). No bypassing of CR.apply or CR.argument-size is required. I tried to come up with a way to use the I-stage to shift #required+optional args into the proper place but was unsuccessful.

It may be useful to have pull-apply-args set the PC to the second half word when faulting, in which case the second opcode would need to be defined. [There is no sequencing code that will do what I think you want in that case, i.e., +2 for the even halfword and +1 for the odd halfword. --Moon]
****

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**. This is the case for any object that is pushed onto the control stack, unless otherwise specified.

*
****

Notes:

Note that 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.
****

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.

*
****

Notes:

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.
****

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 INSTRUCTION-FORMATS. 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 operand to the function, though). This special case is called the "sp-pop addressing mode." For example, the following sequence is 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 "addess-operand addressing mode." For instructions that employ the address-operand mode, the immediate and sp-pop modes are illegal.

Note that it is always only the *last* argument of an instruction that is specified by an 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".

*

****

Notes:

Note in the hardware, the stack cache address is just the bottom 8 bits of SP + offset + 1 (which uses the carry input to the adder).

****

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 eliminate 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.

*

****

Notes: The second word of the prefix cdr code could also be 1 to indicate that it is not a valid instruction.

****

Internal Registers

The following implementation independent registers are defined:

- In the scratchpad (an implementation-dependent assumption):
  - array-register-event-count
  - binding-stack-pointer
  - binding-stack-limit
  - %catch-block-list
  - control-stack-extra-limit
  - control-stack-limit
  - %count-map-reloads
  - list-cache-address
  - list-cache-area
  - list-cache-length
  - pht-address (=? pht-base)
  - pht-mask
  - processor-fault-reason
  - reset-pc
  - structure-cache-address
  - structure-cache-area
  - structure-cache-length
  - top-of-stack

- Probably not in scratchpad memory (read-only?):
  - **t**
  - **nil**
- Not in scratchpad memory:
  - alu-op-register
  - block-address-0,1,2,3
  - byte-rotate
  - byte-size
  - control-register
  - continuation
  - ephemeral-oldspace
  - frame-pointer
  - local-pointer
  - memory-error-status
  - stack-pointer
  - rotate-latch
  - stack-cache-lower-bound (smallest vitual address now in stack cache)
  - zone-oldspace
- Single bits:
  - fep-mode
  - floating-inexact-trap-enable
  - floating-inexact-status
  - preempt-request
  - sequence-break-request
  - trap-mode (write-only)
- Other control bits:
  - invalidate-map-cache

- ° invalidate-map-cache-entry
- ° instruction-cache-enable
- ° invalidate-instruction-cache
- ° map-cache-enable
- ° mapping-enable

*

****

Notes: 1. Whether local-pointer is a separate register may be implementation-dependent.

2. alu-op-register, byte-rotate, and byte-size could be a single register.

3. Should floating-inexact-trap-enable be in the Control register?

4. Are there any other floating-point modes or status?

5. There might be separate sequence-break flags for I/O, fast, and slow clocks.

6. The "other control bits" are from Efland's file, v:>imach>doc>mem-layout.mss. They may need to be discussed and verified. Or maybe they belong in the implementation document.

****

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 a pre-trap. The only spare data type that numeric instructions accept is **dtp-spare-number**, which will cause a post-trap. Non-numeric instructions (that is, instructions that do not require their arguments to have numeric data types) accept any spare data type and always take a post-trap on encountering one, unless otherwise noted.

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

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".

The following table 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.

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

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.

Data-Read Operations

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
0	`data-read`	-	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-monitor-forward, dtp-gc-forward**) is encountered, the error causes the instruction performing the data read to fault. If a data location that is read contains an address in oldspace, a transport trap handler is invoked to scavenge the page and then the data-read is resumed. See the section "I-machine Garbage Collection".

Data-Write Operations

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
1	`data-write`	-	-	`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** or **dtp-monitor-forward** -- but not **dtp-null** -- is encountered in any location, the error causes the instruction doing the data write to trap out. 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 the data is stored in the last location, preserving the cdr code of the location into which it stores.

CDR-Read Operations

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
9	`cdr`	-	-	`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 fault.

Bind-Read Operations

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

The binding instructions (**unbind-n** and **bind-locative**) 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

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
5	**bind-write**	-	-	trap	ind	ind	ind	-	trap	mtrp	-
3	**bind-w-mon**	-	-	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". 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

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Monitor*	*Pointer*
6	**header-rd**	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 fault. If the data location that is read (without a trap) contains an address in oldspace, a transport trap handler is invoked to scavenge the page and then the header-read is resumed. Refer to the chapter on garbage collection. See the section "I-machine Garbage Collection".

Structure-Offset Operations

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
7	**struc-offset**	-	-	-	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

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
8	**scavenge**	-	-	-	-	-	-	-	trap	-	trnspt
10	**gc-copy**	-	-	-	-	-	-	-	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

Code	*Cycle Type*	*Data*	*Null*	*Header*	*HFWD*	*EFWD*	*1FWD*	*EVCP*	*GC*	*Mon-itor*	*Point-er*
11	**raw-read**	-	-	-	-	-	-	-	-	-	-

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 ephemeral-reference bits in the PHT just as other write operations do.

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.

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 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**.
- An instruction such as **movem** or **increment** that stores into its stack operand preserves the cdr code.

In the following instruction descriptions, 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.

The Instructions

The I-machine implements 211 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

28 subprimitive

instructions.

List-Function Operations

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

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

doc:%allocate-list-block
doc:%allocate-structure-block

car *Instruction*

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

Argument(s) 1:
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.

Post Trap
Type of *arg* is **dtp-list-instance**.

Memory Reference Data-read

TOS Register Effects Valid after

cdr *Instruction*

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

Argument(s)/Operand Address(es) 1:
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 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.

Post Trap
Type of arg is dtp-list-instance.

Memory reference Cdr-read, then data-read

TOS Register Effects Valid after

doc:set-to-car *Instruction*
No documentation available for "Set To Car" as a Instruction.

doc:set-to-cdr *Instruction*
No documentation available for "Set To Cdr" as a Instruction.

doc:set-to-cdr-push-car *Instruction*
No documentation available for "Set To Cdr Push Car" as a Instruction.

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

Immediate Argument Type Signed

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

Post Trap
Type of arg1 is dtp-list-instance.

Memory Reference Data-write

TOS Register Effects 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

Immediate Argument Type **Signed**

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

Post Trap
Type of arg1 is dtp-list-instance or if the type of *arg1* **is dtp-list and its cdr code is cdr-next or cdr-nil.**

Memory Reference **Cdr-read, then data-write**

TOS Register Effects **Valid before, invalid after**

Interruptible Instructions

No documentation available for "Interruptible Instructions" as a Section.

****

Notes:

set-to-car may get flushed.

****

Predicate Instructions

eq, eql, zl-user:equal-number, zl:greaterp, zl:lessp, zl-user:endp, plusp, minusp, zerop, zl-user:logtest, zl-user:type-member-n, and the **no-pop** versions of those instructions that take more than one argument.

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

eq *Instruction*

eq-no-pop

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

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

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.)

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

eql *Instruction*
No documentation available for EQL as a Instruction.

zl-user:equal-number *Instruction*
No documentation available for ZL-USER:EQUAL-NUMBER as a Instruction.

zl:greaterp *Instruction*
No documentation available for ZL:GREATERP as a Instruction.

zl:lessp *Instruction*
No documentation available for ZL:LESSP as a Instruction.

zl-user:endp *Instruction*
No documentation available for ZL-USER:ENDP as a Instruction.

plusp *Instruction*
No documentation available for PLUSP as a Instruction.

minusp *Instruction*
No documentation available for MINUSP as a Instruction.

NIL

zerop *Instruction*
No documentation available for ZEROP as a Instruction.

zl-user:logtest *Instruction*
No documentation available for ZL-USER:LOGTEST as a Instruction.

NIL

zl-user:type-member-n *Instruction*
No documentation available for ZL-USER:TYPE-MEMBER-N as a Instruction.

Numeric Operations

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

Refer also to the following:

> **zl-user:equal-number**
> **zl:greaterp**
> **zl:lessp**
> **plusp**
> **minusp**
> **zerop**

If either argument to a numeric instruction is a non-number, then the instruction will pre-trap. Otherwise, if both arguents 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 post-trap 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 a post trap on encountering operands of type **dtp-single-float**. This post trap is in addition to any mentioned in the instruction definitions.

zl-user:add *Instruction*

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

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

Immediate Argument Type Unsigned

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

Post Traps
Type of *arg1* or *arg2* is other than dtp-fixnum or dtp-single-float.
Integer overflow.
Floating-point over- or underflow.

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:sub

Instruction

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

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

Immediate Argument Type Unsigned

Description
Subtracts
arg2 from *arg1*, and pushes the result on the stack.

Post Traps
Type of *arg1* or *arg2* is other than dtp-fixnum or dtp-single-float.
Integer overflow.
Floating-point over- or underflow.

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:unary-minus

Instruction

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

Argument(s) 1:
arg any numeric data type

Immediate Argument Type Unsigned

Description
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.

Post Traps
Type of *arg* is other than dtp-fixnum or dtp-single-float.
Integer overflow.

Memory Reference None

TOS Register Effects Valid after

zl-user:increment

Instruction

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

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

Immediate Argument Type Not applicable

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

Post Trap
Type of *arg* is other than **dtp-fixnum** or
dtp-single-float.
Integer overflow.

Memory Reference None

TOS Register Effects Unchanged

zl-user:decrement

Instruction

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

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

Description
Subtracts 1 from *arg* and stores the result back into the
operand.

Post Trap
Type of *arg* is other than **dtp-fixnum** or **dtp-single-float**.
Integer overflow.

Memory Reference None

TOS Register Effects Unchanged

zl-user:multiply

Instruction

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

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

Immediate Argument Type Signed

Description
Computes *arg1*arg2* and pushes the result on the stack.

Post Traps
Type of *arg1* or *arg2* is other than **dtp-fixnum** or **dtp-single-float**.
Integer overflow.
Floating-point over- or underflow.

Memory Reference None

TOS Register Effects Valid before, valid after

zl:quotient *Instruction*

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

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type, must not be zero

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.

Post Traps
Type of *arg1* or *arg2* is other than **dtp-fixnum** or **dtp-single-float**.
Integer overflow.

Memory Reference None

TOS Register Effects Valid before, valid after

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
arg2 any numeric data type, 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 positive infinity, and the remainder (*TOS*) is such that *arg1* = *arg2* * *NOS* + *TOS*.
See the section "Division Operations That Return Two Values".

Post Traps
Data type of either argument is other than dtp-fixnum.

Memory Reference None

TOS Register Effects Valid before, valid after

floor

Instruction

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

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

Post Traps
Data type of either argument is other than dtp-fixnum.

Memory Reference None

TOS Register Effects Valid before, valid after

truncate

Instruction

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

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type, 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 zero, and the remainder (*TOS*) is such that *arg1 = arg2 * NOS + TOS.*

Post Traps
Data type of either argument is other than dtp-fixnum.

Memory Reference None

TOS Register Effects Valid before, valid after

round *Instruction*

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

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

Post Traps
Data type of either argument is other than dtp-fixnum.

Memory Reference None

TOS Register Effects Valid before, valid after

zl:remainder *Instruction*

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

Argument(s)
arg1 any numeric data type

arg2 any numeric data type, must not be zero

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.

Post Traps
Data type of either argument is other than dtp-fixnum.
Integer overflow.

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:rational-quotient *Instruction*

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

Argument(s) 2:
arg1 any numeric data type
arg2 any numeric data type, must not be zero

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*. 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.
(This instruction implements the CL:/ function.)

Post Traps
Data type of either argument is other than dtp-fixnum or dtp-single-float.
Integer overflow.

Memory Reference None

TOS Register Effects 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

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 post-traps.)

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.

Post Trap
Type of *arg1* or *arg2* is other than **dtp-fixnum** or **dtp-single-float**.

Memory Reference None

TOS Register Effects 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

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 post-traps.)

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.

Post Trap
Type of *arg1* or *arg2* is other than **dtp-fixnum** or **dtp-single-float**.

Memory Reference None

TOS Register Effects Valid before, valid after

zl:logand *Instruction*

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

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

Immediate Argument Type Signed

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

Post Trap
Type of *arg1* or *arg2* is not **dtp-fixnum**

Memory Reference None

TOS Register Effects Valid before, valid after

zl:logior *Instruction*

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

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

Immediate Argument Type Signed

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

Post Trap
Type of *arg1* or *arg2* is not **dtp-fixnum**

Memory Reference None

TOS Register Effects Valid before, valid after

zl:logxor *Instruction*

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

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

Immediate Argument Type Signed

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

Post Trap
Type of *arg1* or *arg2* is not dtp-fixnum

Memory Reference None

TOS Register Effects 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

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.

Post Trap
Type of *arg1* or *arg2* is not dtp-fixnum.
Integer overflow.

Memory Reference None

TOS Register Effects Valid before, valid after

rot

Instruction

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

lsh

Instruction

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

sys:%32-bit-plus

Instruction

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

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

Immediate Argument Type Unsigned

Description
Pushes *arg1* + *arg2* on the stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

sys:%32-bit-difference *Instruction*

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

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

Immediate Argument Type Unsigned

Description
Pushes *arg1* - *arg2* on the stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%multiply-double *Instruction*

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

*
****
Notes:

This instruction could be eliminated, if space gets tight. DCP would like to see this placed near the middle of the delete list.
****

zl-user:%add-bignum-step

Instruction

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%sub-bignum-step

Instruction

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%multiply-bignum-step *Instruction*

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

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%divide-bignum-step *Instruction*

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

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

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. Overflow is not checked, so 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*|).

Post Trap
To Lisp to handle division by zero.

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%lshc-bignum-step *Instruction*

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

Argument(s) 3:
arg1 dtp-fixnum
arg2 dtp-fixnum
arg3 dtp-fixnum, must be between 0 and 32. inclusive

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

*
****
Notes: Flushed **lognot**. DCP wonders about the other 13 Boolean instructions.

What should be done about specifying the slots for all cases of data types? Put in an introductory paragraph to the numeric instruction section?
****

Data-Movement Instructions

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

zl:push *Instruction*

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

Argument(s) 1:
arg any data type

Immediate Argument Type Unsigned

Description
Pushes *arg* on stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl:pop *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:movem *Instruction*

Format Operand from stack, *Value(s) Returned* 1
address-operand mode (immediate and sp-pop addressing modes illegal)

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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:push-n-nils *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* I

Argument(s) 1:
I dtp-fixnum

Immediate Argument Type Unsigned

Description
Pushes *I* nils on the stack, where *I* is the immediate argument.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:push-address *Instruction*

Format Operand from stack, *Value(s) Returned* 1
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg address operand

Immediate Argument Type Not applicable

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

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:set-sp-to-address *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg is address operand

Immediate Argument Type Not applicable

Description
Sets the stack pointer to the address of *arg*.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:set-sp-to-address-save-tos *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg is address operand

Immediate Argument Type Not applicable

Description
Sets the stack pointer to the address of *arg*. The new top of
stack is set to the value that was previously on the top of stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:push-address-sp-relative *Instruction*

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

Argument(s) 1:
arg dtp-fixnum

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

****
Notes:

Refer to the file V:>MOON>IMACH>POP.TEXT for more information about this.
****

zl-user:stack-blt *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s)
arg1 dtp-locative pointing to a location in the current stack frame
arg2 dtp-locative pointing to a location in the current stack frame
arg1 less than or equal to *arg2*

Immediate Argument Type Signed

Description
With the value of *arg1* being *TO* and the value of *arg2* being *FROM*, moves 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. This instruction is not interruptible.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:stack-blt-address *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s)
arg1 dtp-locative, pointing to a location in the current stack frame
arg2 is an address operand
arg1 less than or equal to the address of arg2

Immediate Argument Type Not applicable

Description
With the value of *arg1* being *TO* and *arg2* being *FROM-ADDR*, moves 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.

The instruction

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

is equivalent to the instruction sequence

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

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

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

Field-Extraction Instructions

ldb, dpb, zl-user:char-ldb, zl-user:char-dpb, sys:%p-ldb, sys:%p-dpb, zl-user:%p-tag-ldb, zl-user:%p-tag-dpb

The following instructions are used to extract and deposit fields from different data types. 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. See the section "Format for Field Extraction".

ldb *BB FS* *Instruction*

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

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

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".

Post Trap
Type of *arg1* **is dtp-bignum.**

Memory Reference **None**

TOS Register Effects **Valid after**

dpb *BB FS* *Instruction*

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

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

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".

Post Trap
Type of *arg1* **or** *arg2* **is dtp-bignum.**

Memory Reference **None**

TOS Register Effects **Valid before, valid after**

zl-user:char-ldb *BB FS* *Instruction*

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

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

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

Post Traps **None**

Memory Reference **None**

TOS Register Effects **Valid after**

zl-user:char-dpb *BB FS* *Instruction*

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

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

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".

Post Traps **None**

Memory Reference **None**

TOS Register Effects **Valid before, valid after**

* ***

Notes BEE doesn't think that arg1 being an integer is legitimate.

DCP can live with arg1 causing an error if it's a bignum.
****

sys:%p-ldb *BB FS* *Instruction*

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

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

Description

Extracts the field specified by *BB* and *FS* from the bottom 32 bits of the word at the address contained in *arg*, then pushes the extracted field on the stack. It is illegal, though not checked, to specify a field with bits outside the bottom 32 bits.
See the section "Format for Field Extraction".

Post Traps None

Memory Reference Raw-read

TOS Register Effects Valid after

*
****
Notes: 3600 %p-ldb-immed

%P-LDB: The comment about illegality of fields outside the bottom 32 bits applies to all field-extraction instructions and should be repeated in the section at the front "Format for Field Extraction". Actually I occasionally found it useful to exploit the strange thing it does on the 3600 (see strange-ldb in the 3600 microcode manual), I suppose we could define the I Machine to do the same strange thing rather than making it strictly illegal.

I plan that the operations be defined, but I could figure out how to explain what the weird cases do. Certainly it is an easy way to get the ROT (for fixnums) instruction for free.
****

sys:%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 dtp-fixnum (10-bit immediate)

Description
Deposits the value *arg1* into the field in the contents of the location addressed by *arg2* specified by *BB* and *FS*.
It is illegal, though not checked, to specify a field with bits outside the bottom 32 bits. See the section "Format for Field Extraction".

Post Traps None

Memory Reference Raw-read followed by raw-write

TOS Register Effects Valid before, invalid after

zl-user:%p-tag-ldb *BB FS* *Instruction*

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

Argument(s) 2:
arg1 any Lisp data type

BB and FS **dtp-fixnum** (10-bit immediate)

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. It is illegal, though not checked, to specify a field with bits outside the top 8 bits.
See the section "Format for Field Extraction".

Post Traps None

Memory Reference Raw-read

TOS Register Effects Valid after

zl-user:%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 **dtp-fixnum** (10-bit immediate)

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. No data types are checked.
See the section "Format for Field Extraction".

Post Traps None

Memory Reference Raw-read followed by raw-write

TOS Register Effects Valid before invalid after

Array Operations

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

See the section "I-Machine Array Registers".

Instructions for Accessing One-Dimensional Arrays

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

zl-user:aref-1 *Instruction*

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

Argument(s)
arg1 is either **dtp-array**, **dtp-array-instance**, **dtp-string**, or **dtp-string-instance**
arg2 **dtp-fixnum**

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.

Post Trap Type of *arg1* is **dtp-array-instance** or **dtp-string-instance** or if the **array-long-prefix** bit is set to 1.

Memory Reference Header-read, data-read

TOS Register Effects Valid before, valid after

zl-user:aset-1 *Instruction*

Format Operand from stack *Value(s) Returned* 0

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

Immediate Argument Type Unsigned

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

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**, checks the data type of the argument. When the array element-type is **dtp-character** and the array byte-packing is 8-bit bytes, the instruction traps if bits < 31:8> of the character are nonzero. It does not check that fixed numbers are within range.

Post Trap Type of *arg2* is **dtp-array-instance** or **dtp-string-instance** or if the **array-long-prefix-bit** is set to 1.

Memory Reference Header-read, data-write

TOS Register Effects Valid before, invalid after

****

Notes:

BEE thinks that it will be hard/inconvenient to implement the checking of the top bits of **dtp-character** 8-bit arrays.

****

zl-user:aloc-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** (array must contain full-word Lisp references and be one-dimensional);
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.

Post Trap Type of *arg1* is **dtp-array-instance** or **dtp-string-instance** or if the **array-long-prefix-bit** is set to 1.

Memory Reference Header-read

TOS Register Effects **Valid before, valid after**

Instructions for Creating Array Registers

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

zl-user:setup-1d-array *Instruction*

Format **Operand from stack** *Value(s) Returned* **4**

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

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".

Post Trap **Type of *arg* is dtp-array-instance or dtp-string-instance or if the array-long-prefix-bit is set to 1.**

Memory Reference **Header-read**

TOS Register Effects **Valid after**

zl-user:setup-force-1d-array *Instruction*

Format **Operand from stack** *Value(s) Returned* **4**

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

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".

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.

Post Trap Type of *arg* is dtp-array-instance or
dtp-string-instance or if the array-long-prefix-bit is be set to
1.

Memory Reference Header-read

TOS Register Effects Valid after

Instructions for Fast Access of Arrays

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

zl-user:fast-aref-1 *Instruction*

Format Operand from stack, *Value(s) Returned* 1
address-operand mode (immediate and sp-pop addressing modes illegal)

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

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.

This instruction takes a pre-trap if the current event count does not equal the array-register event count.
See the section "I-Machine Array Registers".

Post Trap None

Memory Reference Data-read

TOS Register Effects Valid before, valid after

zl-user:fast-aset-1 *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s)
arg1 any Lisp data type
arg2 dtp-fixnum
arg3 the address operand (address of an array register)

Immediate Argument Type Not applicable

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

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 traps if bits <31:8> of the character are nonzero.

This instruction takes a pre-trap if the current event count does not equal the array-register event count.
See the section "I-Machine Array Registers".

Post Trap None

Memory Reference Data-write

TOS Register Effects Valid before, invalid after

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 is either dtp-array, dtp-array-instance, dtp-string, or dtp-string-instance
arg2 dtp-fixnum

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.

Post Trap Type of *arg1* is dtp-array-instance or dtp-string-instance.

Memory Reference Header-read, data-read

TOS Register Effects Valid before, valid after

zl:store-array-leader *Instruction*

Format Operand from stack *Value(s) Returned* 0

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

Immediate Argument Type Unsigned

Description
Stores *arg1* into the element specified by *arg3* of the leader of *arg2*. Returns no values.

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.

Post Trap Type of *arg2* is dtp-array-instance or dtp-string-instance.

Memory Reference Header-read, data-write

TOS Register Effects Valid before, invalid after

zl-user:aloc-leader *Instruction*

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

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

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.

Post Trap Type of *arg2* is dtp-array-instance or dtp-string-instance.

Memory Reference Header-read

TOS Register Effects Valid before, valid after

Branch and Loop Instructions

zl-user:branch, zl-user:branch-true{-else}{-and}{-no-pop}{-extra-pop}, Branch-false{-else}{-and}{-no-pop}{-extra-pop}, zl-user:loop-decrement-tos, zl-user: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 branch instruction with an offset of zero is executed and the branch would be taken, the instruction traps instead. This does not apply to 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.

zl-user:branch *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 0

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

Immediate Argument Type Not applicable

Description
Continues execution at the location offset *I* halfwords from the current program counter (PC). Traps if the offset is zero.

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user: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:
I is **dtp-fixnum**

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

nopop 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 traps if 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.

branch-true	Always pop once, whether or not branch taken.
branch-false	Always pop once, whether or not branch taken.
branch-true-no-pop	Do not pop, whether or not branch taken.
branch-false-no-pop	Do not pop, whether or not branch taken.
branch-true-else-no-pop	No pop if no branch, pop once if branch.
branch-false-else-no-pop	No pop if no branch, pop once if branch.
branch-true-and-no-pop	No pop if branch taken, pop if no branch.
branch-false-and-no-pop	No pop if branch taken, pop if no branch.
branch-true-and-extra-pop	Pop twice if branch, pop once if no branch.
branch-false-and-extra-pop	Pop twice if branch, pop once if no branch.
branch-true-else-extra-pop	Pop once if branch, pop twice if no branch.
branch-false-else-extra-pop	Pop once if branch, pop twice if no branch.
branch-true-extra-pop	Always pop twice, whether or not branch taken.
branch-false-extra-pop	Always pop twice, whether or not branch taken.

Not generated:

branch-true-and-no-pop-else-nopop-extra-pop	Same as **branch-true**
branch-false-and-no-pop-else-nopop-extra-pop	Same as **branch-false**

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:loop-decrement-tos *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 0

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

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.

Post Trap
Type of *arg1* other than dtp-fixnum.

Memory Reference None

TOS Register Effects Valid after

zl-user:loop-increment-tos-less-than *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 0

Argument(s) 3:
arg1 any numeric data type
arg2 any numeric data type
I dtp-fixnum

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.

Post Trap Type of *arg1* or *arg2* not either dtp-fixnum or dtp-single-float.

Memory Reference None

TOS Register Effects Valid before, valid after

*
****

Notes:

loop-increment-tos-less-than could be flushed.

LONG-BRANCH - This is a proposed instruction. It probably will not exist in the IVORY processor, but might be implemented in other I series processors. This instruction takes a 8 bit branch offset in bits 24 through 31, an 8 bit signed immediate in bits 16 through 23, a predicate specifier in bits 10 through 15, and regular operand specifier in bits 0 through 9. This allows branches of the form: BR-GREATERP FP|0 8. Another motivation for this instruction is for type branches, where the immediate is a type mask, the predicate is TYPE-MEMBER, and the regular operand specifier is the operand. If the offset of a branch is 0 *or some other specified offset* and the branch condition is true, then an error is signalled.

Note that having this instruction on some processors implies that worlds will not be transportable.
****

Block Instructions

zl-user:%block-n-read, zl-user:%block-n-read-shift, zl-user:%block-n-read-alu, zl-user:%block-n-read-test, zl-user:%block-n-write

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 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) 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.
invert-test <4>	(1 bit) For **%block-n-read-test**: invert the sense of the test. This is the same bit as set-cdr-next.
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.
test <1:0>	(2 bits) Select one of four tests (**%block-n-read-test** only).

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 byte-r, byte-s, and the rotate-latch registers. These are also internal registers that must be loaded by means of **%write-internal-register** instructions before the **%block-n-read-shift** instruction is executed.

zl-user:%block-n-read *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 1:
I **dtp-fixnum** (a 10-bit mask)

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 as a side effect.

Post Trap None

Memory Reference Cycle-type specified

TOS Register Effects Valid after

zl-user:%block-n-read-shift *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 1:
I **dtp-fixnum** (10-bit mask)

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 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 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 as a side effect.

Post Trap None

Memory Reference Cycle-type specified

TOS Register Effects Valid after

zl-user:%block-n-read-alu *Instruction*

Format Operand from stack, *Value(s) Returned* 1
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg is any numeric data type

Immediate Argument Type Not applicable

Description
Performs the ALU operation specified in the alu-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 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
TEST -- not applicable
INVERT TEST -- not applicable

Post Trap Traps according to the generic-arithmetic traps associated with the specified ALU operation

Memory Reference Data-read

TOS Register Effects Unchanged

****

Note:

BEE thinks that the generic arithmetic traps will be difficult/expensive/inconvenient to implement.

****

zl-user:%block-n-read-test *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 1 or 2:
arg(s) can be any Lisp data type, except for when logtest, which requires dtp-fixnum, is selected

Immediate Argument Type Not applicable

Description
Performs the test selected by the 2 test bits of the 10-bit immediate argument with the sense determined by the invert-test bit of the same. These tests 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.)

If the test succeeds, transfers control to the program counter in SP|-1.

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".
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.

Post Trap None

Memory Reference Cycle-type specified.

TOS Register Effects Valid for 2-operand tests, unchanged

zl-user:%block-n-write *Instruction*

Format Operand from stack *Value(s) Returned* 0

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

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.

Post Trap None

Memory Reference Raw-write

TOS Register Effects Unchanged

Function-Calling Instructions

zl-user:dtp-call-compiled-even, zl-user:dtp-call-compiled-odd, zl-user:dtp-call-indirect, zl-user:dtp-call-generic, and the **-prefetch** versions of these last four, **zl-user:start-call, zl-user:finish-call-n, finish-call-apply-n, zl-user:finish-call-tos, finish-call-apply-tos, zl-user:entry-rest-accepted, entry-rest-not-accepted, zl-user:locate-locals, zl-user:return-single, zl-user:return-multiple, zl-user:return-kludge, zl-user:take-values**

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".

zl-user:dtp-call-compiled-even *Instruction*

dtp-call-compiled-even-prefetch

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1:
Included in the instruction is *addr*, the address of the first instruction 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".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:dtp-call-compiled-odd *Instruction*

dtp-call-compiled-odd-prefetch

Format Full-word instruction *Value(s) Returned* Not applicable

Argument(s) 1:
Included in the instruction is *addr*, the address of the first

instruction 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".

Post Trap **None**

Memory Reference **None**

TOS Register Effects **Valid after**

zl-user:dtp-call-indirect *Instruction*

dtp-call-indirect-prefetch

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".

Post Trap **None**

Memory Reference **Data-read**

TOS Register Effects **Valid after**

zl-user:dtp-call-generic *Instruction*

dtp-call-generic-prefetch

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".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

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".

zl-user:start-call *Instruction*

Format Operand from stack *Value(s) Returned* Not applicable

Argument(s) 1:
arg is any data type

Immediate Argument Type Signed

Description
Starts a function call of the function specified by *arg*.
See the section "Starting a Function Call".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:finish-call-n *I* *Instruction*

finish-call-n-apply

Format 10-bit immediate *Value(s) Returned* Not applicable

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

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".

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.

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user:finish-call-tos *I* *Instruction*

finish-call-tos-apply

Format 10-bit immediate *Value(s) Returned* Not applicable

Argument(s) 2:
I **dtp-fixnum**
arg **dtp-fixnum**

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 a number that is equal the number of arguments explicitly supplied with the call.

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".

finish-call-tos-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.

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user: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

Immediate Argument Type Unsigned

Description
Performs an argument match-up process that either traps, 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".

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".

Post Trap See the section "Trapping Out of Entry and Restarting".

Memory Reference See the section "Pull-apply-args".

TOS Register Effects Invalid after

zl-user:locate-locals *Instruction*

Format Operand from stack *Value(s) Returned* Not applicable

Argument(s) 0

Immediate Argument Type Not applicable

Description
Pushes (cr.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 cr.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.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

****
Notes: The **locate-locals** instruction can be flushed if necessary. -- BEE
****

zl-user:return-single *Instruction*

Format Operand from stack, immediate *Value(s) Returned* Not applicable

Argument(s) 1:
arg is **dtp-fixnum** 0, 1, or 2

Immediate Argument Type Unsigned

Description
Specifies the value to be returned on the top of stack according to the immediate operand: 0, the current top of stack; 1, t; 2, nil.
Removes the returning function's frames from the control, binding, and data stacks, and 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.
See the section "Function Returning".

Post Trap None

Memory Reference None

TOS Register Effects 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.

*

****

Notes: The actual values of the immediate operand to specify TOS, t, and nil have not been assigned yet. The values mentioned here are only placeholders.

DCP says that this instruction is flushable.

****

zl-user: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

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. See the section "Function Returning".

Post Trap None

Memory Reference None

TOS Register Effects Status afterwards is determined by value disposition and seen as status after **finish-call** in caller

zl-user:return-kludge *Instruction*

Format Operand from stack, immediate or sp-pop addressing modes only

Value(s) Returned Not applicable

Argument(s)
dtp-fixnum, non-negative

Immediate Argument Type Unsigned

Description
Returns the number of values specified by *arg* on top of the stack. Ignores the cleanup bits in the Control register. Used only for certain internal stack-manipulating subroutines.
See the section "Function Returning".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:take-values

Instruction

Format Operand from stack, immediate *Value(s) Returned arg*

Argument(s)
arg is dtp-fixnum

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

Binding Instructions

zl-user:bind-locative-to-value, zl-user:bind-locative, zl-user:unbind-n, zl-user:%restore-binding-stack

Instructions that perform binding operations check for stack overflow. Those that perform unbinding operations check for stack underflow.

zl-user:bind-locative-to-value *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 2:
arg1 dtp-locative
arg2 any Lisp data type

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*. 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. See the section "Binding Stack".

Post Trap None

Memory Reference Bind-read, followed by two raw-writes, followed by bind-write

TOS Register Effects Valid before, invalid after

zl-user:bind-locative *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 1:
arg dtp-locative

Immediate Argument Type Not applicable

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. See the section "Binding Stack".

Post Trap None

Memory Reference Bind-read, followed by two raw-writes

TOS Register Effects Invalid after

zl-user:unbind-n *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 0

Argument(s) 1:
I dtp-fixnum

Immediate Argument Type Unsigned

Description
Unbinds the top *I* 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. Copies bit 38 of each locative word on the binding stack into the Control register binding-cleanup bit as it pops the locative. See the section "Binding Stack".

Post Trap None

Memory Reference Two bind-reads, followed by bind-write

TOS Register Effects Unchanged

zl-user:%restore-binding-stack *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 1:
arg dtp-locative

Immediate Argument Type Not applicable

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. See the section "Binding Stack".

Post Trap None

Memory Reference Two bind-reads, followed by bind-write

TOS Register Effects **To be determined**

Catch Instructions

zl-user:catch-open, zl-user: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`	`xtra-arg`	`clnup-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`	`xtra-arg`	`clnup-catch`	`previous catch block`

The *catch-block-tag* word refers to an object that identifies the particular catch operation. 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.

zl-user:catch-open *N* *Instruction*

Format 10-bit immediate *Value(s) Returned* 0

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

Description
This instruction has two versions, catch and unwind-protect, which are specified by bit 0 of the immediate argument, *n*. Bit 0 is 0 for catch, 1 for unwind-protect. Bits 6 and 7 of *n* contain the value disposition. Bits 1-5 and 8-9 are not used. 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. 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 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 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.

See the section "Catch Blocks".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:catch-close *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0

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 catch-block-pointer and fetch three words: the 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 in bit 38 of the catch-block-binding-stack-pointer word. If 0, the instruction is done. Note that stack-pointer is not changed. If 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.

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".

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

NIL

Lexical Variable Accessors

zl-user:push-lexical-var-n, zl-user:pop-lexical-var-n, zl-user:movem-lexical-var-n

The three instructions described in this section allow the first eight lexical variables in a lexical environment to be accessed.

zl-user:push-lexical-var-n *Instruction*

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

Argument(s) 1:
arg **dtp-list** (must be a cdr-coded lexical environment, but not checked) or **dtp-locative**

Immediate Argument Type Unsigned

Description
Pushes on the stack the lexical variable of environment *arg* indexed by *n*. *n* is a number between 0 and 7 that is stored in the bottom three bits of the opcode.

Post Trap None

Memory Reference Data-read

TOS Register Effects Valid after

zl-user:pop-lexical-var-n *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 2:
arg1 any data type
arg2 **dtp-list** (must be a cdr-coded lexical environment, but not checked) or **dtp-locative**

Immediate Argument Type Unsigned

Description
Pops *arg1* off the stack and stores the result into the lexical variable of environment *arg2* indexed by *n*. *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.

Post Trap None

Memory Reference Data-write

TOS Register Effects Invalid after

zl-user:movem-lexical-var-n *Instruction*

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

Argument(s)
arg1 any data type
arg2 dtp-list (must be a cdr-coded lexical environment, but not checked) or dtp-locative

Immediate Argument Type Unsigned

Description
Stores *arg1*, without popping, into the lexical variable of environment *arg2* indexed by *n*. *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.

Post Trap None

Memory Reference Data-write

TOS Register Effects Valid after

Instance Variable Accessors

zl-user:push-instance-variable, zl-user:pop-instance-variable, zl-user:movem-instance-variable, zl-user:push-address-instance-variable, zl-user:push-instance-variable-ordered, zl-user:pop-instance-variable-ordered, zl-user:movem-instance-variable-ordered, zl-user:push-address-instance-variable-ordered, zl-user:%instance-ref, zl-user:%instance-set, zl-user:%instance-loc

Mapped Accesses to Self

The next four instructions are called within methods or generic function calls. They have parameters pertaining to the instance in question. 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 assuption 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).

zl-user: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.)

Immediate Argument Type Unsigned

Description
Pushes the instance variable indexed by *I* on the stack.
See the section "Mapped Accesses to Self".

Post Trap None

Memory Reference Data-read (to mapping table), header-read (to header of mapping table), data-read

TOS Register Effects Valid after

zl-user:pop-instance-variable *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 0

Argument(s)
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.)

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".
Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Post Trap None

Memory Reference Data-read (to mapping table), header-read (to header of mapping table), data-write

TOS Register Effects Invalid after

zl-user:movem-instance-variable *I* *Instruction*

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

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.)

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".
Note that only 38 bits are stored: the cdr-code bits of memory are unchanged.

Post Trap None

Memory Reference Data-read (to mapping table), header-read (to header of mapping table), data-write

TOS Register Effects Valid after

zl-user: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 or nil.)

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".

Post Trap None

Memory Reference Data-read (to mapping table), header-read (to header of mapping table)

TOS Register Effects Valid after

Unmapped Accesses to Self

The next four instructions are called within methods or generic function calls. They have parameters pertaining to the instance in question. Each of these instructions is an access to **self**, unmapped.

With the instance in FP|3, such an instruction uses the immediate argument *I* as the offset to an instance variable. These instructions do not check whether the offset is within bounds.

zl-user:push-instance-variable-ordered *I* *Instruction*

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

Argument(s)
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".

Post Trap None

Memory Reference Data-read

TOS Register Effects Valid after

zl-user: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**
(Note that the implicit argument self must be an instance data type.)

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".

Post Trap None

Memory Reference Data-write

TOS Register Effects Invalid after

zl-user:movem-instance-variable-ordered *I* *Instruction*

Format Operand from stack, immediate *Value(s) Returned* 0

Argument(s)
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 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".

Post Trap None

Memory Reference Data-write

TOS Register Effects Valid after

zl-user:push-address-instance-variable-ordered *I* *Instruction*

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

Argument(s)
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 address of the instance variable indexed by *I* on the stack. See the section "Unmapped Accesses to Self".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

****
Note: This is a prime candidate for deletion. -- BEE
****

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.

zl-user:%instance-ref *Instruction*

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

Argument(s) 2:
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 a 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".

Post Trap None

Memory Reference Header-read, data-read (to flavor descriptor), data-read (to instance-variable slot)

TOS Register Effects Valid before, valid after

zl-user:%instance-set *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 3:
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 a 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".

Post Trap None

Memory Reference Header-read, data-reads, data-write

TOS Register Effects Valid before, invalid after

zl-user:%instance-loc *Instruction*

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

Argument(s) 2:
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 a 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".

Post Trap **None**

Memory Reference **Header-read, data-reads**

TOS Register Effects **Valid before, valid after**

*
****

Notes:

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.
****

Subprimitive Instructions

zl-user:%ephemeralp, zl-user:%unsigned-lessp, %unsigned-lessp-no-pop, zl-user:%allocate-list-block, zl-user:%allocate-structure-block, zl-user:%pointer-plus, sys:%pointer-difference, zl-user:%pointer-increment, zl-user:%read-internal-register, zl-user:%write-internal-register, zl-user:%coprocessor-read, zl-user:%coprocessor-write, zl-user:%memory-read, zl-user:%memory-read-address, zl-user:%memory-write, zl-user:%tag, zl-user:%set-tag, sys:%store-conditional, sys:%p-store-contents, zl-user:%set-cdr-code-n, zl-user:%merge-cdr-no-pop, zl-user:%generic-dispatch, zl-user:%message-dispatch, zl-user:%locate-pht-entry, zl-user:%jump, zl-user:%check-preempt-request, zl-user:%halt

zl-user:%ephemeralp *Instruction*
No documentation available for ZL-USER:%EPHEMERALP as a Instruction.

zl-user:%unsigned-lessp *Instruction*
No documentation available for ZL-USER:%UNSIGNED-LESSP as a Instruction.

zl-user:%allocate-list-block *Instruction*

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

Argument(s) 2:
arg1 any type other than dtp-nil
arg2 dtp-fixnum

Immediate Argument Type Unsigned

Description
Using three internal registers, named *list-cache-area, list-cache-length,* and *list-cache-address,* this instruction:

1. Takes a pre-trap unless (eq *arg1* list-cache-area).

2. Computes list-cache-length minus *arg2*. Takes a pre-trap if the result is negative. Stores the result into list-cache-length unless a trap 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))))
```

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

****
Notes:

Refer to the file V:>MOON>IMACH>CONS.TEXT.
****

zl-user:%allocate-structure-block *Instruction*

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

Immediate Argument Type Unsigned

Argument(s) 2:
arg1 any type other than dtp-nil
arg2 dtp-fixnum

Description
Using three internal registers, named *structure-cache-area*, *structure-cache-length*, and *structure-cache-address*, this instruction:

1. Takes a pre-trap unless (eq *arg1* structure-cache-area).

2. Computes structure-cache-length minus *arg2*. Takes a pre-trap if the result is negative. Stores the result into structure-cache-length unless a trap 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).

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%pointer-plus *Instruction*

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

Argument(s) 2:
arg1 can be any data type, but dtp-locative is expected
arg2 any data type, but dtp-fixnum expected

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*.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

sys:%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

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%pointer-increment *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg any data type

Immediate Argument Type Not applicable

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

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user:%read-internal-register *I*

Instruction

Format 10-bit immediate

Value(s) Returned 1

Argument(s) 1:
I dtp-fixnum

Immediate Argument Type Unsigned

Description
Pushes the contents of the internal register specified by *arg* on top of the stack. See the section "Internal Registers".

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:%write-internal-register *I*

Instruction

Format 10-bit immediate

Value(s) Returned 0

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

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".

Post Trap None

Memory Reference None

TOS Register Effects Invalid after

zl-user:%coprocessor-read *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

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

Description
Reads the coprocessor register specified by the immediate field *I* and pushes the result on the stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:%coprocessor-write *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 0

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

Description
Writes *arg1* into the coprocessor register specified by the immediate field *I*.

Post Trap None

Memory Reference None

TOS Register Effects Invalid after

zl-user:%memory-read *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 2:
arg1 any Lisp data type
I **dtp-fixnum** (10-bit mask)

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 "Memory References".

Post Trap None

Memory Reference Controlled by the immediate field.

TOS Register Effects Valid after

****
Notes

DCP wants to know if this turns on cr.no-trap.
****

zl-user:%memory-read-address *I* *Instruction*

Format 10-bit immediate *Value(s) Returned* 1

Argument(s) 2:
arg1 any Lisp data type
I dtp-fixnum (10-bit 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.

Post Trap None

Memory Reference Controlled by the immediate field.

TOS Register Effects Valid after

zl-user:%tag *Instruction*

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

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

Immediate Argument Type Signed

Description
Returns the tag of *arg* as a fixnum.

Post Trap None

Memory Reference None

TOS Register Effects Valid after

zl-user:%set-tag *Instruction*

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

Argument(s) 2:
arg1 any data type
arg2 dtp-fixnum

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

****

There may be two versions of this instruction: one that turns on cr.trap-mode and one that doesn't.

BEE hopes that we don't get in trouble because this instruction sets the cdr code, but doesn't see how it could.

****

sys:%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

Description
If the contents 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. 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.

Post Trap None

Memory Reference Data-read, followed by data-write (using the possibly followed pointer)

TOS Register Effects Valid before, invalid after

sys:%p-store-contents *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 2:
arg1 address to store into
arg2 value to store (no type checking)

Immediate Argument Type Signed

Description
Stores *arg2* into memory location addressed by *arg1*, preserving the cdr code but not following invisible pointers.

Post Trap None

Memory Reference Raw-read followed by raw-write

TOS Register Effects Valid before, invalid after

zl-user:%memory-write *Instruction*

Format Operand-from-stack *Value(s) Returned* 0

Argument(s) 2:
arg1 address to store into
arg2 value to store (no type checking)

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-data-type instruction; in the I-Machine it is legal to have invisible pointers and special markers in the stack temporarily for this purpose.

Post Trap None

Memory Reference Raw-write

TOS Register Effects Valid after

zl-user:%set-cdr-code-n *Instruction*

Format Operand from stack, *Value(s) Returned* 0
address-operand mode (immediate and sp-pop addressing modes illegal)

Argument(s) 1:
arg any data type

Description
N, which is part of the opcode, is either 1 or 2. Sets the cdr code field of *arg* to *N*.

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user:%merge-cdr-no-pop *Instruction*

Format Operand from stack, *Value(s) Returned* 1
address-operand mode (immediate and sp-pop adddressing modes illegal)

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

Description
Sets the cdr-code field of *arg2* to the cdr-code field of *arg1*. *arg1* is not popped off the stack.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%generic-dispatch *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0

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.
See the section "Calling a Generic Function". 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|3 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. Traps if the method found is not **dtp-even-pc**, **dtp-odd-pc**, or **dtp-fixnum**.

Post Trap None

Memory Reference Several data-reads

TOS Register Effects Invalid after

zl-user:%message-dispatch *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0

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". 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. Traps if the method found is not dtp-even-pc, dtp-odd-pc, or dtp-fixnum.

Post Trap None

Memory Reference Several data-reads

TOS Register Effects Invalid after

zl-user:%locate-pht-entry *Instruction*

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

Argument(s) 1:
arg can be any data type, but a pointer type is expected

Immediate Argument Type Signed

Description
Returns a locative (in the physical portion of the virtual address space) to a PHT entry that either matches the argument address or is the first deleted or invalid entry encountered during the search if the argument address is not in the PHT. Any existing map cache entry for the page is invalidated as a side effect.

Post Trap None

Memory Reference Raw-read

TOS Register Effects Valid after

****
Notes DCP would like to see this return both words of the two-word entry.
****

zl-user:%jump *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 1:
arg dtp-even-pc or **dtp-odd-pc**

Immediate Argument Type Not applicable

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.

Post Trap None

Memory Reference None

TOS Register Effects Valid before, valid after

zl-user:%check-preempt-request *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s) 0

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".

Post Trap None

Memory Reference None

TOS Register Effects Unchanged

zl-user:%halt *Instruction*

Format Operand from stack *Value(s) Returned* 0

Argument(s)
None

Immediate Argument Type Not applicable

Description
Stops executing Lisp and transfers control to the supervisor.

Post Trap **None**

Memory Reference **None**

TOS Register Effects **Unchanged**

****
Notes: This needs to be worked out. DCP
****

*
****
Notes:

Deleted: follow-cell-forwarding (=> %memory-read-address), follow-structure-forwarding (=> %memory-read-address), location-boundp [=> (/= (%data-type (%memory-read data-read)) dtp-null)], %p-structure-offset (=> %memory-read-address followed by %pointer-plus), %p-contents-as-locative (=> %memory-read-address followed by %set-data-type), %p-contents-offset (=> (cdr (%p-structure-offset ...).
****