SUNSTONE ARCHITECTURE

11 / 03 / 87

SUNSTONE ARCHITECTURE

Symbolics Company Confidential

Draft formatted on 30 Oct 87 at 23:24

We appreciate any comments on the organization, technical completeness, and technical accuracy of this draft. (Comments about the product design should go to the appropriate mailing list instead.) Thanks.

Name:_____ , Date:_____

Printed in the United States of America.

SUNSTONE ARCHITECTURE
October 1987

Symbolics, Inc.
11 Cambridge Center
Cambridge, MA 02142

Text masters produced on Symbolics 3600™-family computers and printed on Symbolics LGP2 Laser Graphics Printers.

Printed in the United States of America.

Printing year and number: 88 87 9 8 7 6 5 4 3 2 1

Table of Contents

	Page
1. **INSTRUCTION SET ARCHITECTURE**	1
1.1 DATA TYPES	1
1.1.1 Data Types Whose Definition Is Different Than I-Machines.	1
1.1.2 Array Descriptors	1
1.1.3 Compiled Functions	1
1.2 VIRTUAL AND PHYSICAL ADDRESS SPACE	2
1.2.1 Memory Caches	2
1.3 REGISTERS	3
1.3.1 Window Registers	3
1.3.2 Internal Registers	4
1.4 INSTRUCTION FORMATS	15
1.4.1 Register To Register (RR) Format	15
1.4.2 Register Immediate Short (RIS) Format	17
1.4.3 Register Immediate Long (RIL) Format	17
1.4.4 Direct Branch Format	19
1.4.5 Instruction Sequencing	19
1.4.6 Instruction Field Descriptions	20
1.4.7 Data Type Checking	26
1.4.8 Data Type Setting	27
1.4.9 Memory Operations	27
1.5 INSTRUCTIONS	30
1.5.1 Arithmetic Operations	33
1.5.2 Logical Operations	39
1.5.3 Bit and Byte Operations	43
1.5.4 Call Operations	49
1.5.5 Return Operations	52
1.5.6 Move Operations	56
1.5.7 Direct Branch Operation	59
1.5.8 Conditional Operations	61
1.5.9 Type Operations	66
1.5.10 Load Operations	70
1.5.11 Store Operations	89
1.5.12 Coprocessor Operations	103
1.5.13 MC Register Operations	108
1.6 STACK GROUPS	112
1.6.1 Window Stack	112
1.6.2 Data Stack	113
1.6.3 Binding Stack	115
1.6.4 Stack Group Switching	116
1.7 FUNCTION CALLING	116
1.7.1 Calling	116
1.7.2 Entry	118
1.7.3 Generic Functions	118
1.7.4 Message Passing	119
1.7.5 Lexical Closures	119

1.7.6 Return 120
1.8 EXCEPTIONS 120
1.8.1 *Interrupts* 121
1.8.2 Traps 121
1.9 GARBAGE COLLECTION (GC) 134
1.10 ARRAY REFERENCES. 134
1.10.1 Array Hardware Support 135
1.10.2 Array Header Register 136
1.10.3 Array Length Register 137
1.10.4 Array Descriptors 137
1.10.5 Trap Conditions 138
1.11 STORE CONDITIONAL 139
1.12 INSTRUCTION RESTRICTIONS 140
1.12.1 Load Instruction Restrictions 140
1.12.2 Special Register Restrictions 140
1.12.3 MC Register Restrictions 141
1.12.4 Instruction Sequence Restriction Table 142

APPENDIX A. TABLE OF INSTRUCTION SIDE EFFECTS 143

Index 145

List of Figures

Figure 1. Window Registers Before Call 3
Figure 2. Window Registers After Call 4
Figure 3. RR Formats 16
Figure 4. RIS Formats 17
Figure 5. RIL Formats 18
Figure 6. Direct Branch Format 19
Figure 7. Arithmetic Operation Formats 33
Figure 8. Logical Operation Formats 39
Figure 9. Bit and Byte Operation Formats 43
Figure 10. Call Instruction Formats 49
Figure 11. Return Operation Formats 52
Figure 12. Move Operation Formats 56
Figure 13. Direct Branch Format 59
Figure 14. Conditional Operation Formats 62
Figure 15. Type Operation Formats 66
Figure 16. Load Operation Formats 71
Figure 17. Store Operation Formats 89
Figure 18. Coprocessor Operation Formats 103
Figure 19. MC Reg Operation Formats 109
Figure 20. Window Stack 113
Figure 21. Data Stack 114
Figure 22. Windows After a Trap 122
Figure 23. Trap Vector Format 127

List of Tables

Table 1. Special Registers 5
Table 2. Status Control Register Fields 7
Table 3. Global Registers 8
Table 4. Memory Control Registers 10
Table 5. Instruction Field Descriptions 20
Table 6. Opcodes 21
Table 7. Condition Field Definitions 24
Table 8. Memory Operations 28
Table 9. Instruction Classes 30
Table 10. Instructions 31
Table 11. Stack Group Registers 112
Table 12. Misc Opcode Specific Traps 124
Table 13. Trap Vector Addresses 128
Table 14. Exception Priorities 133
Table 15. Valid Array Types 136
Table 16. Instruction Sequence Restrictions 142
Table 17. Instructions/Conditions Side Effecting Special Registers 143

1. INSTRUCTION SET ARCHITECTURE

This document describes the software architecture of the sunstone processor. It assumes a knowledge of the I-machine architecture as described in the I-Machine Architecture Specification, Revision 2 document.

1.1 DATA TYPES

Sunstone uses essentially the same data types as the I-Machine. See "Chapter 1, Lisp-Machine Data Types" as presented in the Symbolics Document I-Machine Architecture Specification, Revision 2 for a description of the I-Machine's data types. There are some differences however, which are listed below.

All data types exist in the Sunstone machine, though their definitions may differ from the I-Machine data types.

1.1.1 Data Types Whose Definition Is Different Than I-Machines.

Data types with octal values 46 to 77 have different meaning in Sunstone than in the I-Machine.

dtp-even-pc octal 46, will be *dtp-pc* on Sunstone. *dtp-odd-pc* octal 47, will be a breakpoint trap on Sunstone *dtp-breakpoint*.

dtp-packed-instruction, *dtp-call-compiled-even*, *dtp-call-compiled-odd*, *dtp-call-indirect*, *dtp-call-generic*, *dtp-call-compiled-even-prefetch*, *dtp-call-compiled-odd-prefetch*, *dtp-call-indirect-prefetch*, *dtp-call-generic-prefetch*, and *dtp-packed-instruction* octal values 50 to 77 will all be *dtp-instruction*. For a more in-depth description of these see section 1.4 on page 15.

1.1.2 Array Descriptors

Sunstone has a different implementation of array registers than does the I-Machine. See section 1.10.4 on page 137 for a complete description of Sunstone's array support.

1.1.3 Compiled Functions

Compiled function structure is the same as I-Machine's with a 2 word prefix, body, and suffix. The instructions are completely different as described in subsequent sections. Sunstone has no half word instructions, only full word, and double word instructions. The data type of pc values is always *dtp-pc*. Where I-Machine uses the cdr-code bits as sequencing information, Sunstone uses the cdr-code bits for disabling interrupts and preempt. See section 1.4.6 on page 20.

1.2 VIRTUAL AND PHYSICAL ADDRESS SPACE

Sunstone provides support for a virtual memory system that is essentially identical to the I-Machine's, see "Chapter 2, Memory Layout and Addressing" as presented in the Symbolics Document I-Machine Architecture Specification, Revision 2. However Sunstone does not have microcode, thus the details of the implementation are somewhat different. The principal difference is that it is possible to get a Map-Cache miss on any virtual address reference, even if the addressed word resides in physical memory. However, it is always possible to access words that are addressed with VMA=PMA addresses. Therefore certain structures, like the PHT, the trap vectors, most trap routines and all the data that they access must be in VMA=PMA space.

The searching of the PHT and reloading of the Map-Cache, which is done in microcode in the I-Machine, is done in macrocode on Sunstone. The hash-box register in the memory control section makes this a quick operation.

1.2.1 Memory Caches

Sunstone's memory cache scheme has certain characteristics that must be understood in order to insure correct program behavior in all cases. There are two caches of interest: the small, on-chip, instruction cache and the large off-chip data/instruction cache.

Both of the caches are addressed by virtual, rather than physical, addresses. That means that care must be taken to not address the same physical location with more than one virtual address. If two different virtual addresses refer to the same physical address it will be possible that if the data at one of the virtual addresses is modified that change will not be visible to the other virtual addresses. This problem can occur if, for example, a word is addressed both by a normal mapped virtual address and also by a VMA=PMA address.

Since the instruction cache and the large cache are independent, updates to the large cache will not necessarily be reflected in the instruction cache. This will not ordinarily be a problem for normal compiled code since code modification is expected to be a very rare occurance. However, since certain "data" structures, most notably the function definition cell, are executed as instructions, care must be taken when those structures are modified. Whenever an instruction word is stored to a location that was previously used to store a different instruction word, it is possible that the old instruction word has been retained in the instruction cache. The instruction cache can be cleared by executing a word which will occupy the same cache line as the word stored. This can be done by maintaining a block of 64 return instructions and executing a call to the appropriate instruction:

```
store [r0]←r1            ;store an instruction word
and    r0 ←#77,r0        ;mask the low bits
add    r0 ←#returns,r0   ;index into the routine
call   pc ←r0
...
```

returns: <block of 64 return instructions>

1.3 REGISTERS

Sunstone has access to 32 general purpose window registers which implement an overlapping window scheme. Sunstone also has access to many other internal registers that are described below. The window scheme implemented resembles that of the RISC chips developed at Berkeley. None of the registers store the cdr-code bits.

1.3.1 Window Registers

Window Registers provide the means to pass arguments and return values between one function and another during a Call/Return sequence without having to write them out to slower memory. The window used by a function is referred to as the current window, and the window used for passing arguments and receiving values is called the build window. Sunstone permits access to 32 of the window registers at any given point in time, 16 in the current window and 16 in the build window. See figure 1 on page 3, and figure 2 on page 4.

The current window registers labeled R0 - R15 have register values 0 - 15 (0 - 17 octal); the build window registers labeled A0 - A15 have register values 16 - 31 (20 - 37 octal.)

Figure 1. Window Registers Before Call

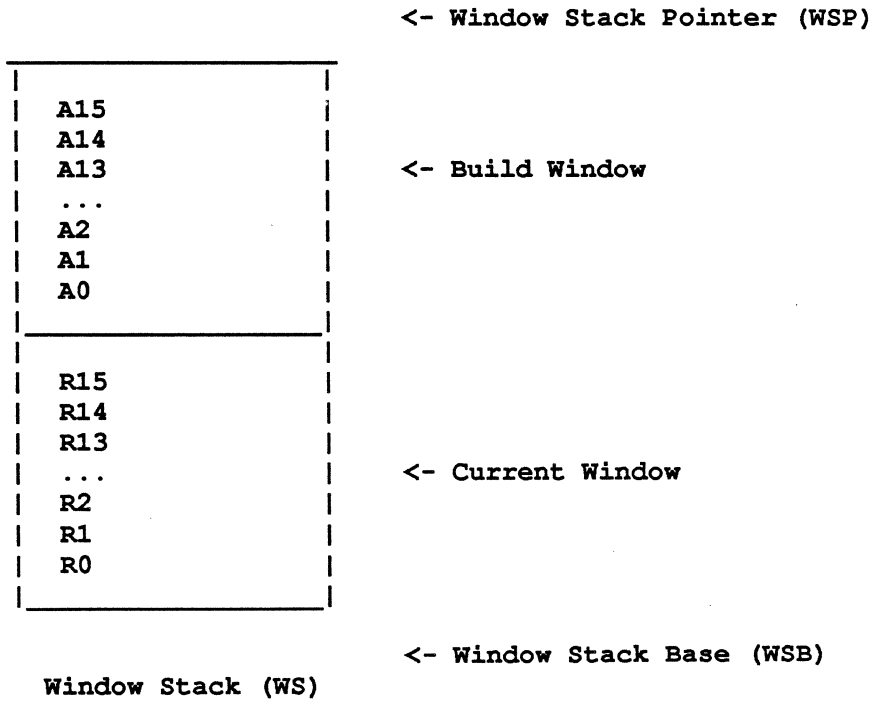
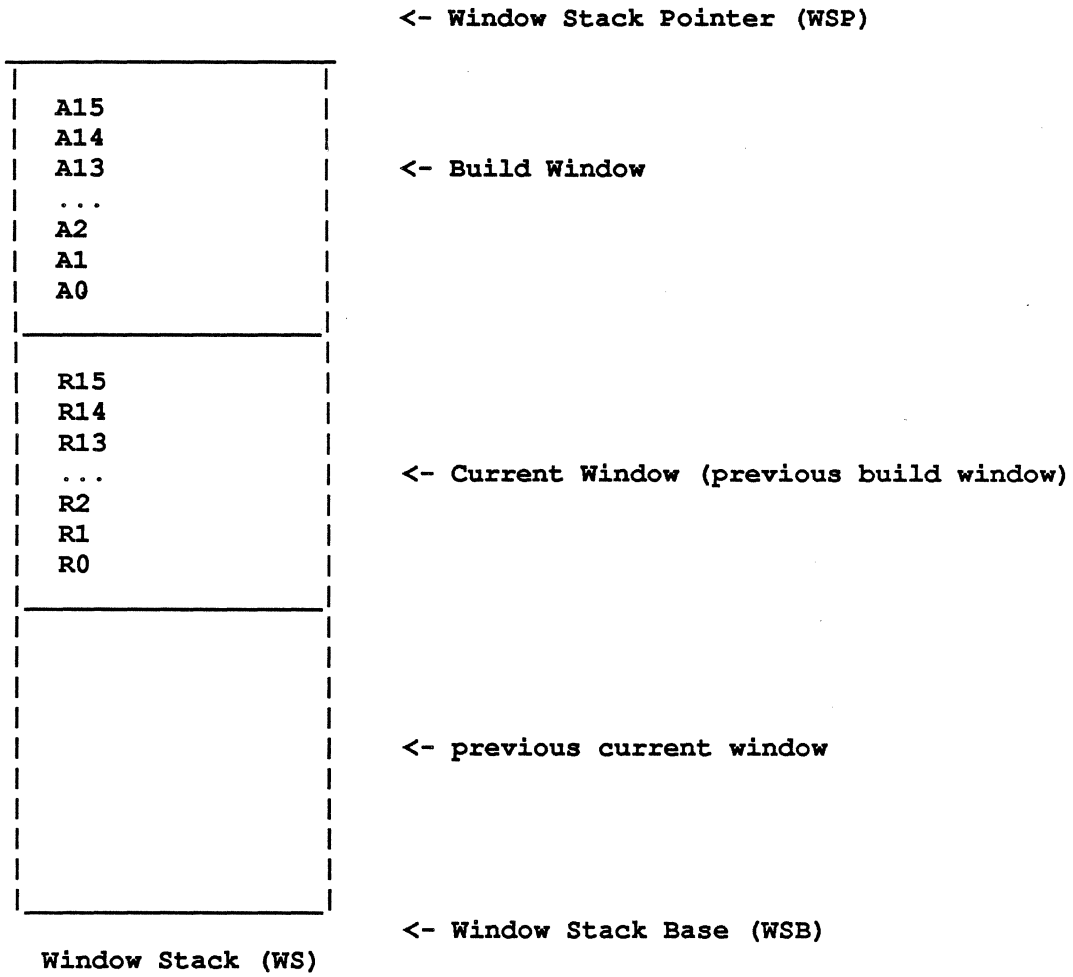


Figure 2. Window Registers After Call



1.3.2 Internal Registers

The internal registers are split up into three categories: Special, Global, and Memory Control (MC). There are 12 special registers which use 10 register addresses; they are hardwired and typically have special hardware attached directly. There are 22 global registers which are completely software definable. There are also provisions for many MC registers (up to 256); presently 40 of them have been assigned. The global and special registers are accessible in all instructions as are the window registers. The MC registers are loaded and stored via a *read-mc-reg* and *write-mc-reg* instruction.

1.3.2.1 *Special Registers*

There are 12 defined special registers which use 10 register addresses. All special registers can be used as a source or a destination in the instructions. See table 1 on page 5.

Table 1. Special Registers

Special Registers						
Name	Reg No	Abbr	Rd-Wr	Type	Size	Side-Effected by
Array Header	76	AHR	R-W	-	38	
Array Length	60	ALR	R-W	fixnum	32	load instruction
Byte Rotate	61	BRR	R-W	fixnum	10	*load-array*, *store-array*
Memory Address	64	MAR	R-W	-	38	*load-cdr*, *load-car-cdr*, *load-header*, *load-structure*
NIL	65	NIL	R	nil	32	
Number of Args	62	N-ARGS	R-W	fixnum	5	*call*, *jcall*, *return*, *return-subvert*
Program Counter	67	PC	W	pc	32	every instruction
Status Control	63	SCR	R-W	fixnum	32	many, load instruction
Status Control 24	66	SCR-24	W	fixnum	24	many
T	66	T	R	symbol	32	
Trap Result	77	TRR	R-W	-	38	
Zero	67	ZERO	R	fixnum	32	

1. *Array Header Register* - This is typically loaded by a *load-header* instruction when preparing to perform array references (see section 1.10.2 on page 136. The hardware uses the bits in this register to help perform the *load-array* and *store-array* instructions (see section 1.5.10 on page 78 for a description of the *load-array* instruction and section 1.5.11 on page 93 for a description of the *store-array* instructions). This 38-bit register can be both read and written.

2. *Array Length Register* - The hardware uses the value of this register to check for out of bounds array references, using the *load-array* and *store-array* instructions. This register is side-effected any time Array Header Register is used as a destination on a load instruction. For a more complete description of this see 1.10.4 on page 137. This 32-bit register can be both read and written. When read it will have a data type of *dtp-fixnum*.

3. *Byte Rotate Register* - This 10 bit register can be read and written. When read it has a data type of *dtp-fixnum* and the most significant 22 bits are 0. It is used by the *dpb* instruction in the Register to Register format, and it is used in the trap handlers for the *load-array* and

store-array instructions, when a packed array is being accessed. A *load-array* or *store-array* instruction will load the Byte Rotate Register with a byte specifier determined from the Array Header Register, which helps to support the trap handling of packed arrays. This side effect occurs even if the instruction traps. For more details on this see 1.10 on page 134.

4. *Memory Address Register* - A 38 bit register that can be both read and written. The memory address register is loaded by hardware with the address and type of calculated memory address in the *load-cdr*, *load-car-cdr*, *load-structure* and *load-header* instructions. This side effect occurs even if the instruction traps.

5. *NIL* - This is a 32-bit constant, and cannot be written, it can only be read, it is the Lisp symbol NIL with data type of *dtp-nil*.

6. *Number of Args* - This 5-bit register can be read and written. When read it has a data type of *dtp-fixnum* and bits 31:5 are 0. The call and return instructions can cause the N-Args Register to be loaded with bits in the instruction (see section 1.5.4 on page 49 for a description of the call instruction and section 1.5.5 on page 52 for a description of the return instruction).

7. *Program Counter* - A 32-bit write only register, this register contains the address of the current instruction. The execution of every instruction affects the contents of the PC. This shares its address with the Zero register,i.e., an attempt to write the Zero, will instead write the Program Counter.

8. *Status Control Register* - A 32-bit register that is read or written, when read it has a data type of *dtp-fixnum*. There are many things that side effect this register. See table 2 on page 7 for a description of the Status Control Register Fields.

9. *Status Control 24* - A separate write only address that allows an instruction to write only the most significant 24 bits of the Status Control Register leaving the cdr and type register portion of this register intact. This register shares its address with the T Register, i.e., an attempt to read this register yields the output of the T Register.

10. *Trap Result Register* - This 38-bit register is used by the hardware after a *return-subvert* instruction is executed. (See the *Return-Subvert* Instruction section 1.5.5 on page 54.)

11. *T* - This 32-bit constant is the lisp representation for T. It has a data type of *dtp-symbol*. As a constant it cannot be written, it can only be read. This shares its address with the status control 24 register,i.e., an attempt to write the T Register, will instead write SCR-24.

12. *Zero* - This 32-bit constant is 0. When read it has a data type of *dtp-fixnum*, it cannot be written. (This register shares its address with the Program Counter,i.e., an attempt to write this register, results in a write to the Program Counter.) Among other uses, the Zero Register will be used for unary minus.

Table 2. Status Control Register Fields

Status Control Register		
Field	Bits	Action Taken on Trap or Interrupt
Type Register	5:0	-
CDR Register	7:6	-
Subvert	8	clear
Trap On Call	9	clear
Trap On Return	10	clear
Trap On Instruction Completion	11	clear
Take Instruction Completion Trap	12	clear
Interrupt Level	15:13	set on interrupt only
Inhibit Preempts	16	copied from inhibit preempt condition
Inhibit Interrupts	17	copied from inhibit interrupt condition

The following describes the fields of the status control register.

1. *Type reg* - These 6 bits hold the data type of the word read by the most recent load instruction. Some store instructions use this register as a source for the data type bits to be written. All load instructions load these bits.

2. *Cdr reg* - These 2 bits hold the cdr-code bits of the word read by the most recent load instruction. Some of the store instructions use this register as a source for the cdr-code bits to write. All load instructions load these bits.

3. *Subvert Instruction* - Set automatically by the hardware when a *return-subvert* instruction occurs (typically in an "emulating" trap routine), this bit tells the next instruction -- the re-executing instruction -- to move the Trap Result Register contents into its destination. The re-executing instruction then clears this bit. If an interrupt occurs in-between the *return-subvert* and the re-executing instructions, the saved control status register will have this bit set. In that case, the interrupt routine must return with a *return-subvert* instruction rather than a *return* instruction. Setting this bit with other than the *return-subvert* instruction is undefined.

4. *Trap on Instruction Completion, Trap on Call, Trap on Return, Take Instruction Completion Trap* - Whenever an instruction completes, the processor copies the Trap on Instruction Completion Bit to the Take Instruction Completion Trap. At the same time, it clears the Trap on Instruction Completion Bit. The Take Instruction Completion Trap will trap an instruction. The handler for this trap decides whether or not to set the Trap On Instruction Completion Bit again when it returns.

   Trap On Call and Trap On Return are special versions of Trap On Instruction Completion. After code sets the Trap On Call bit, the processor will set the Take Instruction

Completion Trap only when a Call instruction completes. At this time it will also clear the Trap On Call. The Trap on Return bit performs the analogous function.

5. *Interrupt Level* - This field contains the currently executing interrupt level. The currently executing level only allows higher level interrupts. The real time user may use the highest levels to implement wired routines that may typically operate on tight time constraints.

6. *Inhibit Interrupt and Inhibit Preempt* - When a trap or an interrupt occurs, the processor sets these bits from the inhibit interrupt or inhibit preempt condition. A trapping or interrupting instruction will not reset the bits; only an instruction that writes or restores the Status/Control register can clear them. The hardware signals an inhibit interrupt or inhibit preempt condition from either these bits or the inhibit bits maintained from instruction to instruction. Each instruction writes inhibit bits from its cdr-code bits. So each instruction controls the inhibit condition of the following instruction. Inhibit interrupt inhibits all interrupt levels; inhibit preempt only inhibits Interrupt Level 1 interrupts. See also the description of Inhibit Interrupts and Inhibit Preempts in section 5 on page 20.

1.3.2.2 *Global Registers*

These 22 registers are completely software definable. The names listed in table 3 on page 8 are a suggested assignment of 17 of these registers. They are all 38-bit registers and fully readable and writable. These registers are selected as source or destination registers in any of the instructions.

Table 3. Global Registers

Global Registers					
Name	Reg No	Abbr	Name	Reg No	Abbr
Binding Stack Limit	40	BSL	List Block Length	50	LBL
Binding Stack Pointer	41	BSP	List Block Pointer	51	LBP
Catch Block Pointer	42	CBP	Structure Block Base	52	SBB
Data Stack Block	43	DSB	Structure Block Length	53	SBL
Data Stack Base	44	DSA	Structure Block Pointer	54	SBP
Data Stack Limit	45	DSL	Window Stack Base	55	WSB
Data Stack Pointer	46	DSP	Window Stack Limit	56	WSL
List Block Base	47	LBB	Window Stack Pointer	57	WSP
			Flag Register	70	FR

1.3.2.3 *Memory Control Registers*

These registers are only accessible via *read-mc-reg* and *write-mc-reg* instructions. (For more information on the *read-mc-reg* and *write-mc-reg* instructions, see section 1.5.12 pages 110 and 111 respectively.) The Memory Control Registers are located in a space of 256 registers. An MC-Register is chosen by the least-significant 8 bits of the computed virtual address of a *read-mc-reg* or *write-mc-reg*. The remaining 24 bits of the address are ignored except when reading/writing the Map Cache (see Map-Cache and Map-Cache-Validbit below). Registers that are smaller than 32 bits are read and written in the least significant bits, and the most significant bits are undefined. All MC registers are read with a type of *dtp-fixnum*. The MC-Register space is divided such that MC-Register codes 000-017 (octal) are internal to the processor and codes 020-377 are external to the processor. For a list of MC registers see table 4 on page 10.

Table 4. Memory Control Registers

Memory Control Registers				
Name	Abbr	Rd-Wr	MC-Reg Code in Octal	Size
Ephemeral Oldspace	EOR	R-W	000	32
Zone Oldspace	ZOR	R/W	001	32
PHT Hashbox	PHTH	R-W	002	24
FPU Configuration	FPC	R-W	003	3
Microsecond Clock	MSC	R-W	004	32
Timer	CLKA	R-W	005	32
Trap Base Register	TBR	R-W	006	13
Return Address	RAR	R-W	007	32
Window Buffer Control	WBC	R-W	010	21
Memory-Error-Status	MERR	R-W	011	7
spare			012 - 017	
Metering Counter A	MCA	R-W	020	32
Metering Counter B	MCB	R-W	021	32
Metering Mode A	MMA	R-W	022	4
Metering Mode B	MMB	R-W	023	4
Cache Control	CCR	R-W	024	16
reserved			025	
spare			026 - 027	
reserved			030	
Interrupt1	INT1	R-W	031	8
Interrupt2	INT2	R-W	032	8
Interrupt3	INT3	R-W	033	8
Interrupt4	INT4	R-W	034	8
Interrupt5	INT5	R-W	035	8
Interrupt6	INT6	R-W	036	8
Interrupt7	INT7	R-W	037	8
Map-Cache-MSH *	MCM	R-W	040	16
Map-Cache-LSH *	MCL	R-W	041	16
Map-Cache-Validbit *	MCV	R-W	042	1
reserved *			043	
Diagnostic-MSH *	DIAM	R-W	044	32
Diagnostic-LSH *	DIAL	R-W	045	32
IBUS-Error-Status-MSH *	IERRM	R-W	046	2
IBUS-Error-Status-LSH *	IERRL	R-W	047	2
IBUS-Error-Addr-MSH *	IEAM	R-W	050	16
IBUS-Error-Addr-LSH *	IEAL	R-W	051	16
ECC Log Counter-MSH *	ECCM	R-W	052	32
ECC Log Counter-LSH *	ECCL	R-W	053	32
Slot Number *	SLOT	R	054	6
reserved *			055	
IBUS Lock *	IBL	R-W	056	1
reserved *			057	
Spare			060 - 377	

* Items specific to the IBUS implementation of Sunstone.

Table 4. Memory Control Registers

Memory Control Registers				
Name	Abbr	Rd-Wr	MC-Reg Code in Octal	Size
Ephemeral Oldspace	EOR	R-W	000	32
Zone Oldspace	ZOR	R/W	001	32
FPU Configuration	FPC	R-W	002	3
Trap Base Register	TBR	R-W	003	13
Return Address	RAR	R-W	004	32
Window Buffer Control	WBC	R-W	005	21
Memory-Error-Status	MERR	R-W	006	7
spare			007 - 017	
Metering Counter A	MCA	R-W	020	32
Metering Counter B	MCB	R-W	021	32
Metering Mode A	MMA	R-W	022	4
Metering Mode B	MMB	R-W	023	4
Cache Control	CCR	R-W	024	16
reserved			025	
Microsecond Clock	MSC	R-W	026	32
Timer	CLKA	R-W	027	32
spare			026 - 027	
reserved			030	
Interrupt1	INT1	R-W	031	8
Interrupt2	INT2	R-W	032	8
Interrupt3	INT3	R-W	033	8
Interrupt4	INT4	R-W	034	8
Interrupt5	INT5	R-W	035	8
Interrupt6	INT6	R-W	036	8
Interrupt7	INT7	R-W	037	8
I-Map-Cache *	IMC	R-W	040-043	32
D-Map-Cache *	DMC	R-W	044-047	32
Diagnostic-MSH *	DIAM	R-W	050	32
Diagnostic-LSH *	DIAL	R-W	051	32
IBUS-Error-Status-MSH *	IERRM	R-W	052	2
IBUS-Error-Status-LSH *	IERRL	R-W	053	2
IBUS-Error-Addr-MSH *	IEAM	R-W	054	16
IBUS-Error-Addr-LSH *	IEAL	R-W	055	16
ECC Log Counter-MSH *	ECCM	R-W	056	32
ECC Log Counter-LSH *	ECCL	R-W	057	32
Slot Number *	SLOT	R	060	6
reserved *			061	
IBUS Lock *	IBL	R-W	062	1
reserved *			063	
Spare			064 - 377	

* Items specific to the IBUS implementation of Sunstone.

IMPORTANT

1. *Ephemeral Oldspace* - The ephemeral oldspace register contains a bit map that specifies, for each of the 32 ephemeral levels, which half of the level is newspace and which half is oldspace. A set bit indicates the upper half is oldspace, a reset bit indicates the lower half is oldspace. This register is identical to the I-Machine's Ephemeral Oldspace Register.

2. *Zone Oldspace Register* - This register is identical to the Zone Oldspace Register in the I-Machine. The Zone Oldspace Register contains a bit map that specifies whether each zone of dynamic space (there are 29 zones) is newspace or oldspace. Bits 31 and 0 of the Zone Oldspace Register are typically 0. Bit 31 represents physical memory zones, and bit 0 represents the ephemeral zone. Since new/old space is a characteristic of virtual memory, bit 31 is set to 0 (the physical memory space). Since bit 0 refers to ephemeral space it is never used.

3. *PHT Hashbox* - This register is used to perform a hashing function used in the PHT lookup algorithm. It implements the hashing function described in the I-Machine architecture document. When written, the PHT hashbox hashes the written data. The next read of the PHT hashbox will return the hashed version of the written data.

4. *FPU Configuration* - A 3-bit register that identifies the existence of a Floating-Point Unit and which floating-point/integer operations are implemented by it. The word is positive true logic and the bits are assigned as follows:

```
bit <2>   Fixed Point Multiply Hardware available

bit <1>   Single Floating Point Add, Subtract and Multiply
          Hardware available

bit <0>   Single Floating Point Compare Hardware available
```

5. *Microsecond Clock* - A 32-bit free-running clock which counts microseconds.

6. *Timer* - An independent 32-bit count down timer is used for event scheduling. The timer, when written with a 32-bit count, begins counting micro-seconds until the count is zero. At zero, the timer sets an interrupt and stops. The interrupt service routine must reload the count down timer to start it again. Writing a timer with 0 causes the longest interval; about 1hr. 15 min.

7. *Trap Base Register* - This 13-bit register is the base of a trap vector that is located in VMA=PMA space, so that the most significant 5 bits are 1, bits 26 - 14 are the trap-base register, and the least significant bits are based on the trap that occurs. See table 13 on page 128.

8. *Return Address Register* - A 32-bit register that is read or written, when read it has a data type of *dtp-pc*. This register is the top of the return address stack. This is side effected by calls, returns, traps, and possibly if the destination of an instruction is the window buffer control register.

9. *Window Buffer Control Register* - This 21-bit register encodes the Window Buffer Pointer, Window Buffer Overflow Limit, and Window Buffer Underflow Limit. Bits 20:16 are the Window Buffer Underflow Limit, bits 12:8 are the Window Buffer Overflow Limit, and bits 4:0 are the Window Buffer Pointer.

10. *Memory-Error-Status* - This register contains the error status of the memory system. The setting of any bit in this register is accompanied by a Hardware-Error Trap. The read/write formats are:

```
bits<31:7> is  unknown on reads, ignored on writes
```

```
bit<6>      is  S-Cache tags parity error

bits<5:4>   is  S-Cache data parity errors

bit<3>      is  Map-Cache tags parity error

bits<2:1>   is  Map-Cache data parity errors

bit<0>      is  External to processor error
```

When bit<0> is set, a memory system error outside of the Caches has occured. Software may read externally defined MC-Register(s) to determine error information. The IBUS implementation of Sunstone has the IBUS-Error-Status and IBUS-Error-Address registers defined for this purpose.

11. *Metering Counters and Metering Modes* - There are two independent counters for metering. Each metering counter is 32 bits wide and meters 8 different events. A *write-mc-reg* to the Metering Mode sets the mode according to the 4 LSBs of the written word. The four bits are interpreted as :

```
bits<2:0>                    Event Metered
     0       number of clocks

     1       number of clocks servicing S-Cache
             Instruction misses

     2       number of clocks servicing S-Cache
             Data  misses

     3       number of clocks servicing writes

     4       number of Instruction S-Cache misses

     5       number of Data S-Cache misses

     6       number of Map-Cache misses that cause traps

     7       number of writes

bit<3>
     1       start the counter

     0       stop the counter
```

A *read-mc-reg* of a Metering Counter reads the present value of the count and does not affect the counter nor event metered. A *read-mc-reg* of the Metering Mode returns a 4-bit code where bits<2:0> identify the event being metered; bit<3>, when set indicates the counter is running.

12. *Cache Control* - Cache Control Register is the mechanics by which the processor writes cache mode and reads cache status. Read only bits are unchanged by writes. The Cache Control Register is written/read with the following fields:

- Write Buffer Empty (bit 0, read-only): a set bit indicates that the write buffer is empty.
- S-Cache Inhibit (bit 1, read/write): a set bit forces all reads of the S-Cache to miss. Writes to the S-Cache, either by a store instruction or by cache refill, are not inhibited. This allows validation of the S-Cache on power-up.
- I-Cache Inhibit (bit 2, read/write): a set bit indicates that instructions are to be inhibited from the I-Cache. Instructions in pages marked with Cache-inhibit in the PHT are inhibited from the I-Cache regardless of the state of this bit.
- Other status and test points (bits 15:3, read-only): used to monitor any points of interest in the machine.

13. *Interrupt 1 thru 7* - These seven registers correspond to seven interrupt levels. Each contains eight bits that requesters on the IBUS may set. An interrupt bit is cleared when it is written with a 1.

14. *Map-Cache* - D-Map and I-Map-Cache - These are used for reading and writing Map-Cache entries. The Map Cache is read/written using the read-mc-reg and write-mc-reg instructions where the computed virtual address is defined to be:

```
virtual-address<31:8> Virtual Page Number to be read/written.

virtual-address<7:0>  D-Map-Cache code when reading/writing the D-Map Cache;
                      I-Map-Cache code when reading/writing the I-Map Cache.
```

The read/written map-cache data is identical for both the I-Map and D-Map Caches and has format below:

```
data<37:32> Data Type. The data type of a valid map-cache entry is
            dtp-fixnum (#o10). An invalid map-cache entry has data
            type dtp-single-float (#o12). All other data types may
            produce undefined results, potentially including hardware
            error (map cache parity) traps on subsequent memory
            references.

data<31:8>  Physical Page Number. The VPN to PPN mapping for this
            map-cache entry. It is ignored during unmapped references;
            Unmapped references do not have Map-Cache entries.

data<7>     Modified. If this bit is clear in the map-cache entry
            used by a store instruction, the instruction takes a
            page-modified trap.  Unmapped references (those in which
            address<31:27> = 37, or the address type is DTP-Physical)
            behave as if this bit were set.

data<6>     Write-Protect.  If this bit is set in the map-cache
            entry used by a store instruction, the instruction takes
            a write-protect trap.  Unmapped references behave as if
            this bit were clear.

data<5>     Cache-Inhibit.  If this bit is set in the Map-Cache
            entry used by a load or store instruction fetches the
            addressed data from main memory and leaves the contents
```

of the S-Cache unchanged. References with address type DTP-Physical behave as if this bit were set; otherwise, if address<31:27> = #o37, as if this bit were clear.

data<4> Transport-Trap. If this bit is clear in the map-cache entry used by a load instruction, the processor will not take a transport trap on the instruction, even when it encounters a pointer to an oldspace zone. Unmapped references behave as if this bit were clear.

data<3:0> Ephemeral Reference. When the data written by a store instruction is a pointer to ephemeral space, bits 25:24 of the pointer are used to select one of these four bits from the map-cache entry selected by the store address. If the selected bit is clear, the store instruction takes an ephemeral-reference-update trap. Stores to unmapped addresses behave as if the selected bit were set.

Map data bits 31:0 are assigned to match the PHT1 word of a PHT entry.

15. *Diagnostic* - Registers used to aid in machine checkout and debugging. Possible uses are: set/scan interface, monitoring test points, generating sync or trigger signals, etc. Since the IBUS implementation of memory control hardware is bit-sliced, there are two Diagnostic registers, each associated with one-half of the SBUS. The exact function of each is to be further defined by the hardware.

16. *IBUS-Error-Status* - These registers contain the error status of the IBUS. Since the implementation of the IBUS interface is bit-sliced, there are two registers, each responsible for one half of the IBUS. The register bits are assigned as:

MSH word: bits<31:2> is unknown on reads, ignored on writes

bit<1> is IBUS acknowledge error

bit<0> is IBUS uncorrectable ECC Error in MSH word of IBUS

LSH word: bits<31:2> is unknown on reads, ignored on writes

bit<1> is IBUS acknowledge error

bit<0> is IBUS uncorrectable ECC Error in LSH word of IBUS"

17. *IBUS-Error-Address* - These registers are generally set by hardware. Upon an uncorrectable ECC Error or an IBUS Acknowledge Error, the registers are loaded with the address of the erroneous word. Since the IBUS interface is bit-sliced, the 32-bit address is available as two 16-bit values. The format for reading/writing the IBUS-Error-Address is:

MSH word: bits<31:16> is unknown on reads, ignored on writes
bits<15:0> is error address<31:16>

LSH word: bits<31:16> is unknown on reads, ignored on writes
bits<15:0> is error address<15:0>"

- Write Buffer Empty (bit 0, read-only): a set bit indicates that the write buffer is empty.
- S-Cache Inhibit (bit 1, read/write): a set bit indicates that instructions and data are to be inhibited from the S-Cache. Instructions and data in pages marked with Cache-inhibit in the PHT are inhibited from the S-Cache regardless of the state of this bit.
- I-Cache Inhibit (bit 2, read/write): a set bit indicates that instructions are to be inhibited from the I-Cache. Instructions in pages marked with Cache-inhibit in the PHT are inhibited from the I-Cache regardless of the state of this bit.
- Other status and test points (bits 15:3, read-only): used to monitor any points of interest in the machine.

13. *Interrupt 1 thru 7* - These seven registers correspond to seven interrupt levels. Each contains eight bits that requesters on the IBUS may set. An interrupt bit is cleared when it is written with a 1.

14. *Map-Cache* - This is used for reading and writing the Map-Cache entries. The Map Cache is read/written using the *read-mc-reg* and *write-mc-reg* instructions where computed virtual address is defined to be:

```
virtual-address<31:8> is the Virtual Page Number
                      to be read/written

virtual-address<7:0>  is the Map-Cache MC-Register
                      code (MSH or LSH)
```

Since the IBUS implementation of the Map-Cache control logic is bit-sliced, reading and writing the 32 bits of the Map Cache requires two read/writes, each 16 bits wide. The read/write formats for the two words are:

```
MSH word:       data<31:16> is ignored on writes, unknown
                            on reads

                data<15:4>  is PPN<23:12>

                data<3>     is Modified

                data<2>     is Write-Protect

                data<1>     is Cache Inhibit

                data<0>     is Transport-Trap


LSH word:       data<31:16> is ignored on writes, unknown
                            on reads

                data<15:4>  is PPN<11:0>

                data<3:0>   is Ephemeral Reference bits<3:0>
```

Writing to either the MSH or LSH of the Map Cache automatically validates the selected entry. When read, the valid bit for the selected entry is unchanged. The state of the valid bit can be read using the Map-Cache-Validbit MC-Register below.

15. *Map-Cache-Validbit* - This is used to access the valid bit of Map Cache entries. Addressing of the Map Cache valid bit is the same as addressing the Map Cache itself except that the virtual-address<7:0> is Map-Cache-Validbit code. When the Map-Cache-Validbit MC-Register is written, the selected Map Cache entry is invalidated. When the Map-Cache-Validbit is read, the LSB of the returned data is the state of the valid bit of the selected Map Cache entry. All other returned bits are unknown.

16. *Diagnostic* - Registers used to aid in machine checkout and debugging. Possible uses are: set/scan interface, monitoring test points, generating sync or trigger signals, etc. Since the IBUS implementation of memory control hardware is bit-sliced, there are two Diagnostic registers, each associated with one-half of the SBUS. The exact function of each is to be further defined by the hardware.

17. *IBUS-Error-Status* - These registers contain the error status of the IBUS. Since the implementation of the IBUS interface is bit-sliced, there are two registers, each responsible for one half of the IBUS. The register bits are assigned as:

```
MSH word:  bits<31:2> is unknown on reads, ignored on writes

           bit<1>     is IBUS acknowledge error

           bit<0>     is IBUS uncorrectable ECC Error in MSH
                      word of IBUS

LSH word:  bits<31:2> is unknown on reads, ignored on writes

           bit<1>     is IBUS acknowledge error

           bit<0>     is IBUS uncorrectable ECC Error in LSH
                      word of IBUS"
```

18. *IBUS-Error-Address* - These registers are generally set by hardware. Upon an uncorrectable ECC Error or an IBUS Acknowledge Error, the registers are loaded with the address of the erroneous word. Since the IBUS interface is bit-sliced, the 32-bit address is available as two 16-bit values. The format for reading/writing the IBUS-Error-Address is:

```
MSH word:  bits<31:16> is unknown on reads, ignored on writes
           bits<15:0>  is error address<31:16>

LSH word:  bits<31:16> is unknown on reads, ignored on writes
           bits<15:0>  is error address<15:0>"
```

19. *ECC Log Counter* - The ECC Log Register counts the number of single bit errors occurring on the IBUS. When the ECC Log Counter overflows, it sets an interrupt bit in its associated interrupt register. Since the IBUS interface is bit-sliced, single bit errors are counted seperately for each half of the IBUS. Reading the ECC-Log-Counter-MSH returns the number of single bit errors for the MSH of the IBUS. Reading the ECC-Log-Counter-LSH returns the number of single bit errors for the LSH of the IBUS.

20. *Slot Number* - A read-only register that returns the slot number that the Sunstone processor is plugged into.

21. *IBUS Lock* - This single bit indicates when a Locked IBUS transaction is to be done. When set, the next load or store instruction will become a locked IBUS read or write cycle. The IBUS Lock remains set until the IBUS-Lock MC-Register is reset.

18. *ECC Log Counter* - The ECC Log Register counts the number of single bit errors occurring on the IBUS. When the ECC Log Counter overflows, it sets an interrupt bit in its associated interrupt register. Since the IBUS interface is bit-sliced, single bit errors are counted seperately for each half of the IBUS. Reading the ECC-Log-Counter-MSH returns the number of single bit errors for the MSH of the IBUS. Reading the ECC-Log-Counter-LSH returns the number of single bit errors for the LSH of the IBUS.
19. *Slot Number* - A read-only register that returns the slot number that the Sunstone processor is plugged into.
20. *IBUS Lock* - This single bit indicates when a Locked IBUS transaction is to be done. When set, the next load or store instruction will become a locked IBUS read or write cycle. The IBUS Lock remains set until the IBUS-Lock MC-Register is reset.

1.4 INSTRUCTION FORMATS

Four different formats of Sunstone instructions specify the operation, type, and operands to use. The Register-to-Register (RR), Register Immediate Short (RIS), and Direct Branch formats all use a single word, 40 bit instruction. Register Immediate Long (RIL) instruction is a double word, 80 bit instruction. Each format, except the direct branch, uses a data type of *dtp-instruction*, these are data type codes 50 to 77. Sunstone interprets a data type of *dtp-compiled-function* as a direct branch instruction; all other data types trap if encountered as instructions.

Each instruction format breaks down into fields that further define the general operation of the instruction. The source1, source2, and destination fields refer to one of 64 registers. 32 of the registers they refer to are special purpose registers such as Number-of-Arguments, Program-Counter, etc. The other 32 registers refer to the Register Window sets, 16 registers for the Current Window and 16 registers for the Build Window.

1.4.1 Register To Register (RR) Format

All opcodes except *read-coproc* are available in the RR format. All RR instructions are one word in length, or 40-bits. The decoding varies slightly among opcodes (see figure 3 on page 16.) For a complete description of each field see section 1.4.6 on page

1.4 INSTRUCTION FORMATS

Four different formats of Sunstone instructions specify the operation, type, and operands to use. The Register-to-Register (RR), Register Immediate Short (RIS), and Direct Branch formats all use a single word, 40 bit instruction. Register Immediate Long (RIL) instruction is a double word, 80 bit instruction. Each format, except the direct branch, uses a data type of *dtp-instruction*, these are data type codes 50 to 77. Sunstone interprets a data type of *dtp-compiled-function* as a direct branch instruction; all other data types trap if encountered as instructions.

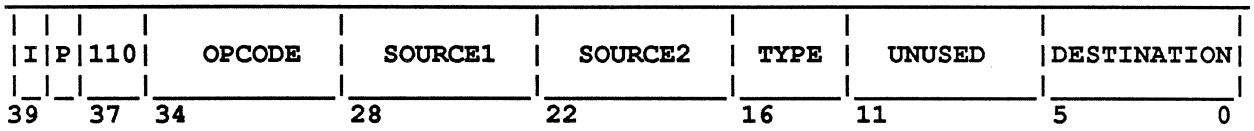
Each instruction format breaks down into fields that further define the general operation of the instruction. The source1, source2, and destination fields refer to one of 64 registers. 32 of the registers they refer to are special purpose registers such as Number-of-Arguments, Program-Counter, etc. The other 32 registers refer to the Register Window sets, 16 registers for the Current Window and 16 registers for the Build Window.

1.4.1 Register To Register (RR) Format

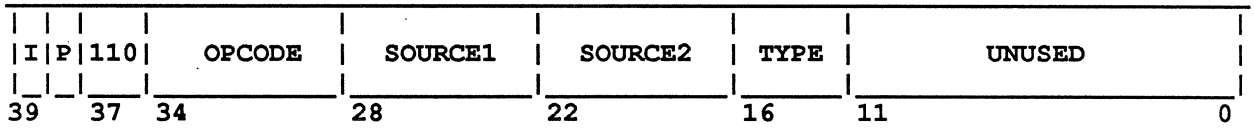
All opcodes are available in the RR format. All RR instructions are one word in length, or 40-bits. The decoding varies slightly among opcodes (see figure 3 on page 16.) For a complete description of each field see section 1.4.6 on page 20

Figure 3. RR Formats

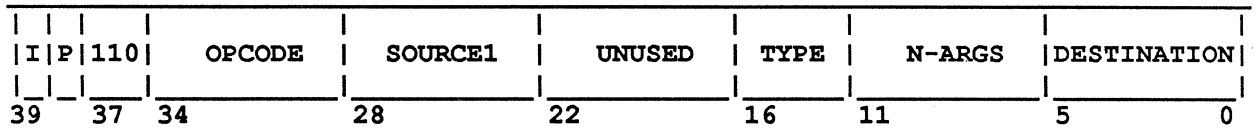
add, add-no-trap, sub, sub-no-trap, multiply, and, or, xor, rot, ash, lsh, ldb, dpb, move, move-type, load, read-mc-reg:



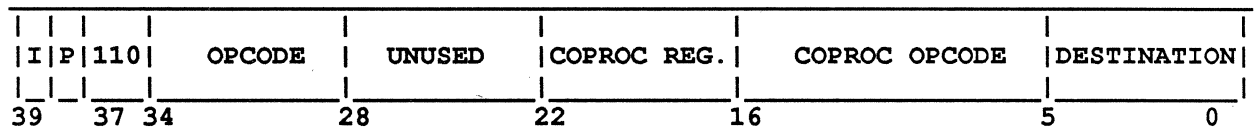
store, write-mc-reg:



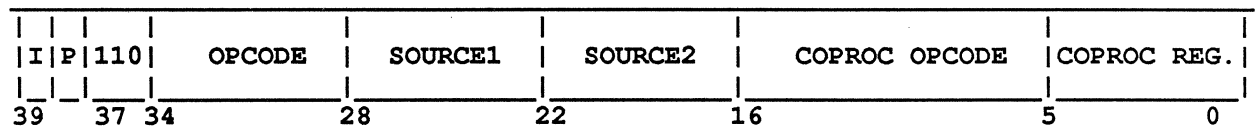
call,jcall, return, return-subvert:



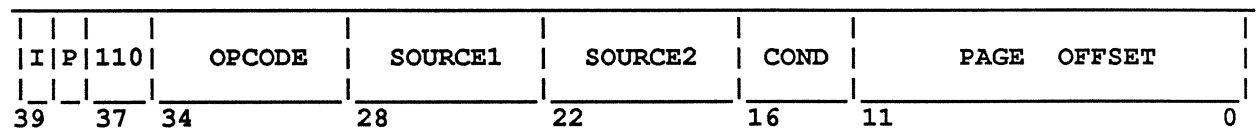
read-coproc:



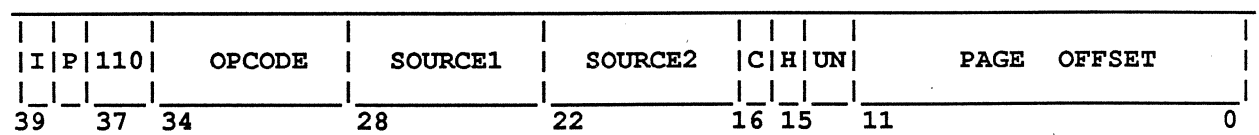
write-coprcc, load-coproc, store-coproc:



branch-next, branch-take, trap:



branch-next-type, branch-take-type, trap-type:

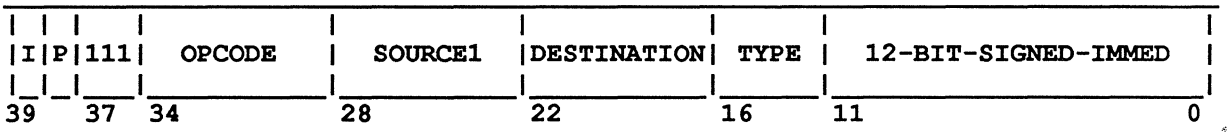


1.4.2 Register Immediate Short (RIS) Format

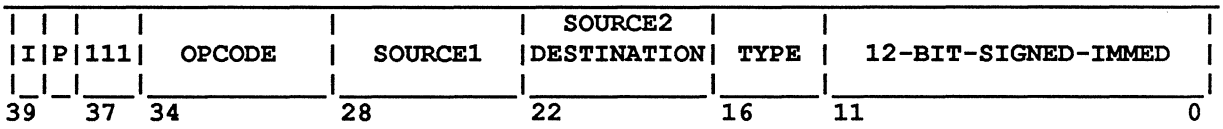
RIS format instructions are one word long, or 40-bits. There are eleven opcodes that are not available in the RIS format: *write-coproc, load-coproc, store-coproc, branch-next-type, branch-take-type, trap-type, call, jcall, return, and return-subvert.* The hardware does not check for these opcodes, and their operation is undefined. The decoding varies slightly among opcodes (see figure 4 on page 17). For a complete description of each field see section 1.4.6 on page 21.

Figure 4. RIS Formats

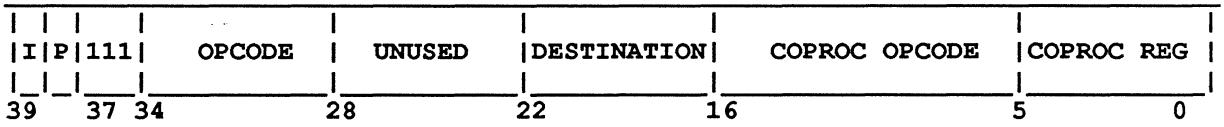
add, add-no-trap, sub, sub-no-trap, multiply, and, or, xor, rot, ash, lsh, ldb, move, move-type, load, read-mc-reg:



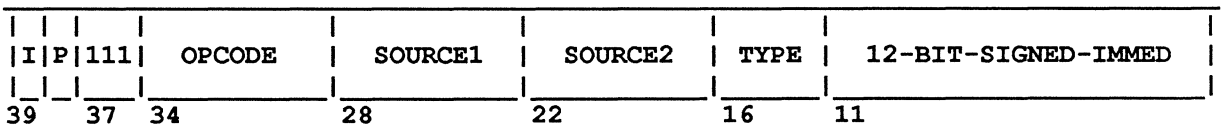
dpb:



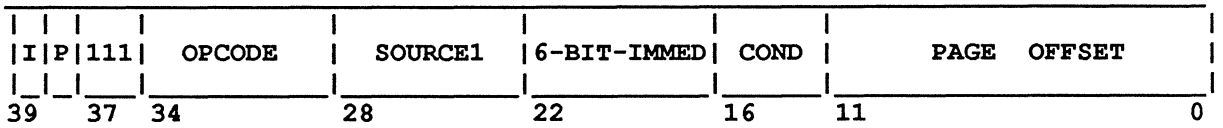
read-coproc:



store, write-mc-reg:



branch-next, branch-take, trap:

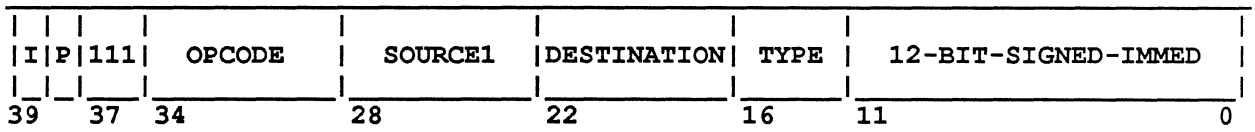


1.4.2 Register Immediate Short (RIS) Format

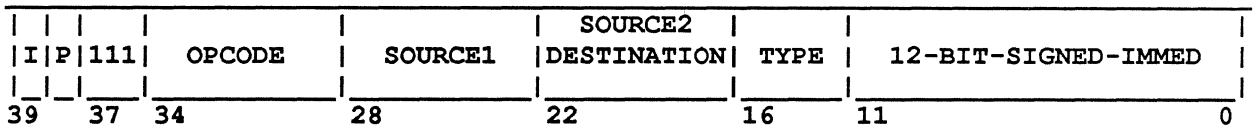
RIS format instructions are one word long, or 40-bits. There are eleven opcodes that are not available in the RIS format: *read-coproc, write-coproc, load-coproc, store-coproc, branch-next-type, branch-take-type, trap-type, call, jcall, return, and return-subvert.* The hardware does not check for these opcodes, and their operation is undefined. The decoding varies slightly among opcodes (see figure 4 on page 17). For a complete description of each field see section 1.4.6 on page 20.

Figure 4. RIS Formats

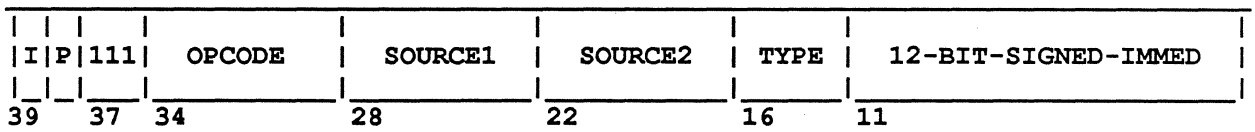
add, add-no-trap, sub, sub-no-trap, multiply, and, or, xor, rot, ash, lsh, ldb, move, move-type, load, read-mc-reg:



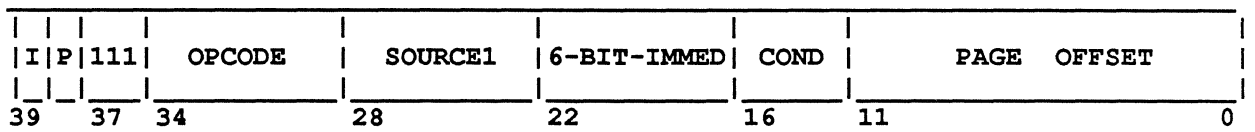
dpb:



store, write-mc-reg:



branch-next, branch-take, trap:

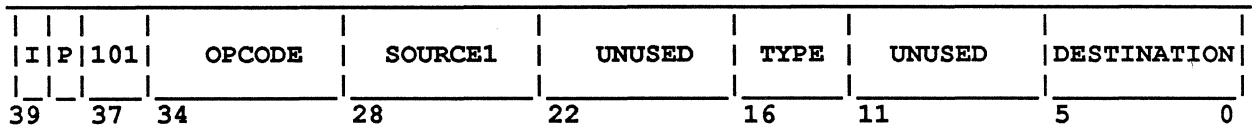


1.4.3 Register Immediate Long (RIL) Format

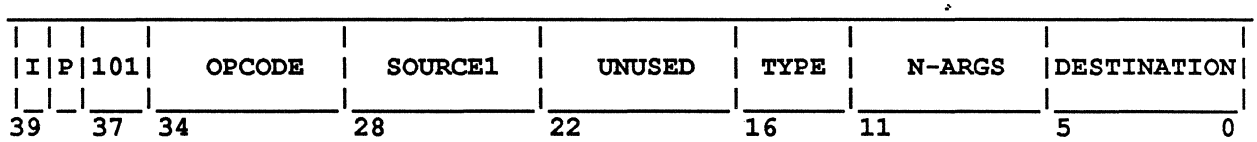
RIL format instructions are two words long, or 80-bits. There are seven opcodes that are undefined in the RIL format: *rot, ash, lsh, ldb, dpb, read-coproc* and *move-type*. The hardware does not check for these opcodes, and their operation is undefined. The second word of each RIL instruction is a 40-bit immediate value. The decoding of the first word of the instruction varies slightly among opcodes (see figure 5 on page 18). For a complete description of each field see section 1.4.6 on page 20.

Figure 5. RIL Formats

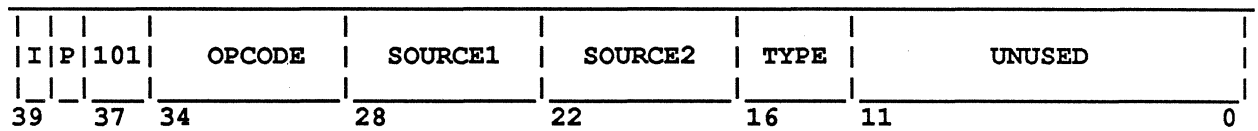
add, add-no-trap, sub, sub-no-trap, multiply, and, or, xor, move, load, read-mc-reg:



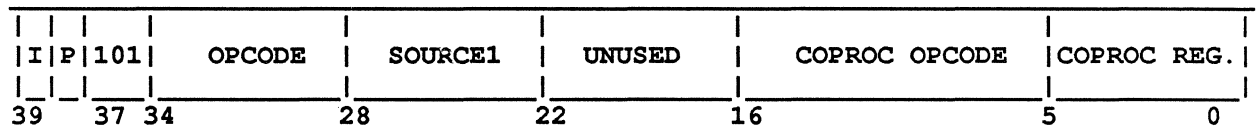
call, jcall, return, return-subvert:



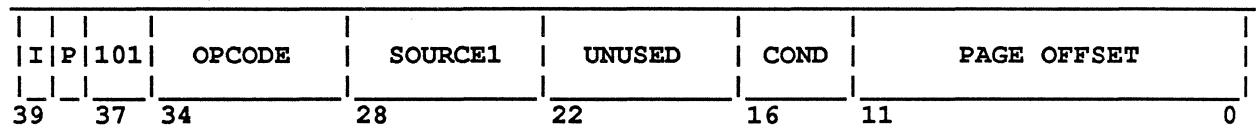
store, write-mc-reg:



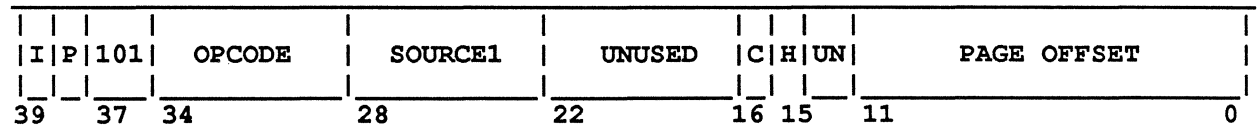
write-coproc, load-coproc, store-coproc:



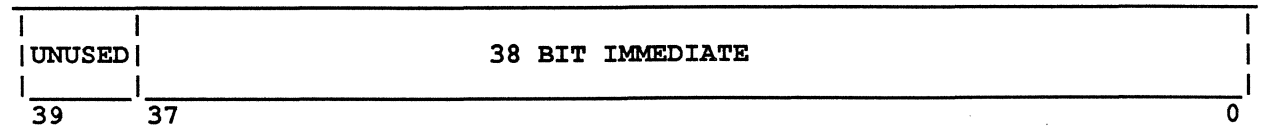
branch-next, branch-take, trap:



branch-next-type, branch-take-type, trap-type



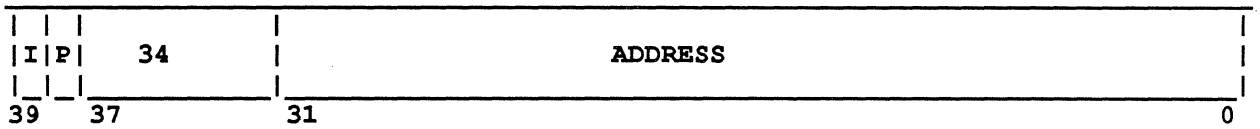
The second word for all RIL instructions:



1.4.4 Direct Branch Format

The direct branch format is a single word, 40-bit instruction with a 6-bit data type of *dtp-compiled-function*, and the 32 least significant bits are interpreted as a 32-bit virtual address to use as the new PC. See figure 6 on page 19.

Figure 6. Direct Branch Format



1.4.5 Instruction Sequencing

Instructions that do not explicitly affect the PC behave as follows:

- RR and RIS instructions increment the PC by 1
- RIL instructions on even boundries increment the PC by 2
- RIL instructions on odd boundries increment the PC by 1

Note that normal RIL instructions will be aligned on an even word boundary. RIL instructions on an odd boundary is a special case that is used to align normal RIL instructions with an instruction that can be executed in zero cycles. Since these odd word RIL instructions might be executed in some cases of interrupts or traps, the instruction must not have any harmful side-effects.

Note also that only the low order 12 bits of the PC are incremented, i.e. there is no carry from bit 11 to bit 12 of the PC. Consequently, the PC wraps around 4K boundaries.

1.4.6 Instruction Field Descriptions

Table 5 on page 20 lists the various instruction fields and describes how they are used.

Table 5. Instruction Field Descriptions

Instruction Field Descriptions		
Bits	Length	Use
39	1	Inhibit interrupts
38	1	Inhibit preemption
37:35	3	Format
37:32	6	Data type
34:29	6	Opcode
31:0	32	PC address
28:23	6	Source1
22:17	6	Source2, destination, 6-bit-signed-immediate, coproc-reg
16:12	5	Type Check, Cond
16:6	11	coproc-opcode
16	1	High/Low
15	1	condition
11:6	6	N-Args
11:0	12	Page Offset, 12-bit-signed-immediate
5:0	6	Destination, coproc-reg

1. *Inhibit Interrupts* - Bit 39, when set causes an Inhibit Interrupt condition on instruction completion. This inhibits interrupts for the **next** instruction. When clear, it allows interrupts on instruction completion, unless the Status Control Inhibit Interrupt is set. If a trap occurs, the inhibit interrupts condition will be saved by the hardware into the Status Control Inhibit Interrupts Bit (see table 2 on page 7).

2. *Inhibit Preemption* - Bit 38, when set causes an Inhibit Preempts condition on instruction completion. This means that preempts cannot occur between the instruction with the preempts inhibited bit set and the completion of the following instruction. When clear, it allows preempts on instruction completion, unless the Status Control Inhibit Preempt is set. If a trap occurs, the inhibit preempts condition will be saved by the hardware into the Status Control Inhibit Preempts bit (see table 2 on page 7).

3. *Format* - The three bit field 37:35 specifies what the instruction format is: 6 - RR, 7 - RIS, and 5 - RIL. For the direct-branch format this field will be 3 and bits 33:32 will be a 4.

1.4.6 Instruction Field Descriptions

Table 5 on page 21 lists the various instruction fields and describes how they are used.

Table 5. Instruction Field Descriptions

Instruction Field Descriptions		
Bits	Length	Use
39	1	Inhibit interrupts
38	1	Inhibit preemption
37:35	3	Format
37:32	6	Data type
34:29	6	Opcode
31:0	32	PC address
28:23	6	Source1
22:17	6	Source2, destination, 6-bit-signed-immediate,
16:12	5	Type Check, Cond
16:6	11	coproc-opcode
16	1	High/Low
15	1	condition
11:6	6	N-Args
11:0	12	Page Offset, 12-bit-signed-immediate
5:0	6	Destination, coproc-reg

1. *Inhibit Interrupts* - Bit 39, when set causes an Inhibit Interrupt condition on instruction completion. This inhibits interrupts for the **next** instruction. When clear, it allows interrupts on instruction completion, unless the Status Control Inhibit Interrupt is set. If a trap occurs, the inhibit interrupts condition will be saved by the hardware into the Status Control Inhibit Interrupts Bit (see table 2 on page 7).

2. *Inhibit Preemption* - Bit 38, when set causes an Inhibit Preempts condition on instruction completion. This means that preempts cannot occur between the instruction with the preempts inhibited bit set and the completion of the following instruction. When clear, it allows preempts on instruction completion, unless the Status Control Inhibit Preempt is set. If a trap occurs, the inhibit preempts condition will be saved by the hardware into the Status Control Inhibit Preempts bit (see table 2 on page 7).

3. *Format* - The three bit field 37:35 specifies what the instruction format is: 6 - RR, 7 - RIS, and 5 - RIL. For the direct-branch format this field will be 3 and bits 33:32 will be a 4.

4. *Data Types* - All instructions have a data type of *dtp-instruction*, or *dtp-compiled-function*. There are 24 different values for *dtp-instruction*, octal values 50 - 77, and *dtp-compiled-function* has a value of 34 octal. All other data type values cause a trap to occur, though it may not be an error. It is the software's responsibility when setting the PC to test for an illegal data type.

5. *Opcode* - This six bit field bits 34 - 29 specifies which of 58 instructions to execute. See table 6 on page 21.

Table 6. Opcodes

Table of Opcodes			
Instruction	Opcode	Instruction	Opcode
load-ephemeralp	00	or	40
load-oldspacep	01	xor	41
load-raw	02	and	42
undefined	03	move	43
load-cdr	04	ash	44
load-structure	05	lsh	45
load-gc-copy	06	rot	46
load-scavenge	07	write-mc-reg	47
load-bind	10	ldb	50
load-header	11	dpb	51
load-data	12	add-no-trap	52
load-data-iv	13	sub-no-trap	53
load-car-cdr	14	add	54
load-cdr-finish	15	sub	55
load-array	16	mult	56
load-coproc	17	write-coproc	57
store-cdr-next	20	jcall	60
store-cdr-nil	21	call	61
store-cdr-normal	22	return	62
store-cdr-3	23	return-subvert	63
store-cdr-reg	24	branch-take-type	64
store-type-reg	25	branch-take	65
store-38-bits	26	move-type	66
undefined	27	read-mc-reg	67
store-bind	30	trap-type	70
undefined	31	trap	71
store-data	32	undefined	72
store-data-iv	33	undefined	73
undefined	34	branch-next-type	74
store-rplacd	35	branch-next	75
store-array	36	undefined	76
store-coproc	37	read-coproc	77

6. *PC Address* - This field is used in the direct branch format. It is loaded into the PC as the address of the next instruction to execute.

7. *Source1* - A six bit field,it is comprised of bits 28 - 23 of the instruction. The bits are interpreted the same as the the source2 and destination fields. This field specifies one of 64 registers to be read: 16 in the current window, 16 in the build window, 10 special registers, or 22 global registers (see section 1.3.2 on page 4).

8. *Source2* - A six bit field, it is comprised of bits 22 - 17 of the instruction. This field is not present for every instruction in every format. When it is present the bits are interpreted the same as the source1 and destination fields. The six bit field specifies one of 64 registers to be read: 16 in the current window, 16 in the build window, 10 special registers, and 22 global registers (see section 1.3.2 on page 4).

9. *Destination* - This 6 bit value specifies one of 64 registers to store the results of the operation in. The registers are defined as in the source1 and source2 fields. The field specifies one of 64 registers to store the result of the operation in: 16 in the current window, 16 in the build window, 10 special registers, and 22 global registers. The destination is present in all instructions but: *store, branch-next, branch-take, trap, branch-next-type, branch-take-type, trap-typeload-coproc, write-mc-reg, write-coproc.* The data type stored in the destination is that of source1 except for the RIS and RIL formats for instructions: *move, return,* and *return-subvert* also RR and RIS formats for *move-type.* In the case of *move, return,* and *return-subvert* the data type of the result will be that of the immediate, which in the case of the RIS format will always be *dtp-fixnum. Move-type* is a special case, see section 1.4.8 on page 27. For *read-mc-reg, read-coproc,* and load instructions, the data type loaded into the destination register is the data type of the word read.

10. *6-bit-signed-immediate* - Used only in the RIS format by the branch instructions: *branch-next, branch-take, trap.* This 6 bit field is bits 22:17 and is a sign extended value with a data type of *dtp-fixnum.*

11. *Type Check* - This 5 bit field uses bits 16 to 12 to represent what data types to trap on. This field is not present in the *read-coproc, write-coproc, load-coproc, store-coproc, branch-next, branch-next-type, branch-take, branch-take-type, trap,* and *trap-type* instructions. Some of the instructions that have no type field still have data type checking. The branch-next, branch-take, and trap instructions (all which use the COND field instead of the TYPE field) will test the type according to the condition. See the COND field description below. Some of these traps will always be error traps, and some will be errors only for certain data types. See table 7 on page 24. The names for type checks in this field typically are the names of the data types that are legal. There is one notable exception, ≠hardware-arith. The ≠hardware-arith type check will always check both sources, even in the immediate formats. It checks that the data types are the same or it traps. It also traps if the data types are not *dtp-fixnum* if there is no floating point coprocessor present, or it traps if the data types are not both *dtp-fixnum* or both *dtp-flonum* if there is a floating point coprocessor present. The data type traps are listed according to the value of the type field, are as follows:

    - *None* - Type Check Field = 0 ; No data types will trap.
    - ≠*fixnum* - Type Check Field = 1; Trap if data type is not *dtp-fixnum.* This is always an error trap.
    - ≠*flonum* - Type Check Field = 2; Trap if data type is not *dtp-single-float.* This is an error trap if the data type is not one of the numeric data types 10 - 17: *dtp-fixnum, dtp-small-ratio, dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex,* or *dtp-spare-number.*
    - ≠*instance* - Type Check Field = 3; Trap if the data type is not one of: *dtp-instance, dtp-list-instance dtp-array-instance,* or *dtp-string-instance.* This is always an error trap.
    - ≠*array* - Type Check Field = 4; Trap if the data type is not *dtp-array.* This is always an error trap.

- ≠*locative* - Type Check Field = 5; Trap if the data type is not *dtp-locative*. This is always an error trap.
- ≠*compiled-function* - Type Check Field = 6; Trap if the data type is not *dtp-compiled-function*. This is an error trap if the data type is not one of: *dtp-generic-function, dtp-instance, dtp-symbol, dtp-lexical-closure*, or *dtp-dynamic-closure*.
- ≠*character* - Type Check Field = 7; Trap if the data type is not *dtp-character*. This is always an error trap.
- ≠*list* - Type Check Field = #o10; Trap if the data type is not *dtp-list*. This is always an error trap.
- ≠*list-locative* - Type Check Field = #o11; Trap if the data type is not *dtp-list* or *dtp-locative*. This is always an error trap.
- ≠*list-nil* - Type Check Field = #o12; Trap if the data type is not *dtp-list* or *dtp-nil*. This is an error trap if the data type is anything but *dtp-list-instance*.
- ≠*list-loc-nil* - Type Check Field = #o13; Trap if the data type is not *dtp-list, dtp-locative*, or *dtp-nil*. This is an error trap for all data types but *dtp-list-instance*.
- ≠*array-string* - Type Check Field = #o14; Trap if the data type is not *dtp-array* or *dtp-string*. This is an error trap if the data type is not *dtp-array-instance* or *dtp-string-instance*.
- ≠*hardware-arith* - Type Check Field = #o15; This trap depends on the floating point and fixnum multiply support offered by a hardware coprocessor, as indicated by the FPU Configuration Register (see page 11). Without hardware support, trap if both sources are not *dtp-fixnum* for *add, sub, branch-take, branch-next*, and *trap* instructions, and always for *mult*instruction. However, when the FPU Configuration register indicates support for fixnum multiply, then a *mult* instruction will not trap if both sources are *dtp-fixnum*. Additionally, when it indicates support for floating point add, subtract and multiply, the *add, sub*, and *mult* instructions will not trap if both sources are *dtp-single-float*. The same is true for *branch-take, branch-next*, and *trap*, when it indicates support for floating point compares. Regardless of the FPU Configuration, this is an error trap if both sources' data types are not one of the numeric data types #o10-#o17: *dtp-small-ratio, dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex*, or *dtp-spare-number*.
- ≠*pointer* - Type Check Field = #o16; Trap if the data type is not a legal pointer. The legal pointer types are: *dtp-monitor-forward, dtp-header-p dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward, dtp-element-forward, dtp-double-float, dtp-bignum, dtp-big-ratio, dtp-complex, dtp-spare-number, dtp-instance, dtp-list-instance, dtp-array-instance, dtp-string-instance, dtp-nil, dtp-list, dtp-array, dtp-string, dtp-symbol, dtp-locative, dtp-lexical-closure, dtp-dynamic-closure, dtp-compiled-function, dtp-generic-function, dtp-spare-pointer-1, dtp-spare-pointer-2, dtp-bound-location, dtp-logic-variable, dtp-gc-forward, dtp-pc*, and *dpt-null*.
- ≠*list-locative* - Type Check Field = #o31; Trap if the data type is not *dtp-list* or *dtp-locative*. This is an error trap for all data types except *dtp-list-instance*.

12. *Cond* - This 5 bit field in bits 16:12 for instructions: *branch-next,branch-take*, and *trap*. See table 7 on page 24.

Table 7. Condition Field Definitions

Table of Cond Field Definition		
Value	Definition	Data Type Test
0	<	≠hardware-arith
1	>	≠hardware-arith
2	≥	≠hardware-arith
3	≤	≠hardware-arith
4	=	≠hardware-arith
5	≠	≠hardware-arith
6	32 bit =	none
7	eq	none
10	32 bit <	none
11	32 bit >	none
12	unsigned <	≠fixnum
13	unsigned >	≠fixnum
14	logtest	≠fixnum
15	endp	≠list-nil-list-instance
16	char =	≠char
17	eql	non-immediate-number
20	not <	≠hardware-arith
21	not >	≠hardware-arith
22	not ≥	≠hardware-arith
23	not ≤	≠hardware-arith
24	not =	≠hardware-arith
25	not ≠	≠hardware-arith
26	not 32 bit =	none
27	not eq	none
30	32 bit ≤	none
31	32 bit ≥	none
32	unsigned ≥	≠fixnum
33	unsigned ≤	≠fixnum
34	not logtest	≠fixnum
35	not endp	≠list-nil-list-instance
36	not char =	≠char
37	not eql	non-immediate-number

The four least significant bits of the cond field define the test condition and type check. The msb, if set, indicates the negation of the test condition. All of the test conditions, except endp, test as source1<cond>source2. Endp tests source1.

Of the 16 test conditions, six of them, namely <, >, ≥, ≤,=,≠ are signed numeric comparisons, using the ≠hardware-arith type check. Two unsigned numeric comparisons, unsigned-< and unsigned->, use the ≠fixnum type check. Four of the test conditions do not have type checks; they are: eq, 32-bit-=, 32-bit-<, and 32-bit->. Note that the eq is a 38 bit equality check. The eql condition is the same as eq except for a type check that traps for non immediate numbers of identical data types. The char= condition is a 32-bit-= with a ≠char type check. The logtest condition indicates a true condition if a bitwise logical and (ie, logand) of the two sources leaves any bit set, and type checks for ≠fixnum. The endp condtion checks to see if the data type of source 1 is *dtp-nil*. Endp has a type check that traps if the source1 data type is not one of *dtp-nil*, *dtp-list*, or *dtp-list-instance*.

13. *Coproc-Opcode* - This 11 bit field, bits 16:6, is available for support of coprocessor hardware. The definition of this field is dependent upon the coprocessor.

14. *High/Low* - This bit 15 is used only in the type instructions: *branch-next-type*, *branch-take-type*, and *trap-type*. This bit determines if the 32-bit mask is applied to the upper half or the lower half of the 64-bit value (see section 1.5.9 on page 66). When this bit is a 1 the mask is applied to the upper half, when this bit is 0 the mask is applied to the lower half.

15. *Condition* - This bit 16 is used only in the type instructions: *branch-next-type*, *branch-take-type*, and *trap-type*. This bit indicates branching on a true or false result of the type test being done. If the bit is 1, the branch is taken if the result of the type test is non-zero, if the bit is 0, the branch is taken if the result of the type test is zero (see section 1.5.8 on page 61).

16. *N-Args* - This 6 bit field, bits 11:6, is used by the call and return instructions. If the most significant bit is a 1, the N-Args register is loaded with the value in the least significant 5 bits of this field, if the most significant bit is a 0, the N-Args register is unchanged.

17. *Page Offset* - This 12 bit field, when present, is in bits 11 to 0. This field is used by the branch and trap instructions. For the branch instructions, this field represents the least significant 12 bits of the address to branch to if the condition being tested is true. The most significant 20 bits are the most significant 20 bits of the address of the instruction being executed. It can be thought of as an offset within the current page. For the trap instructions, this field is the least significant 12 bits of the trap address. The most significant 5 bits of the trap address are 1s. Bits 13 and 12 of the trap address are 0s. Bits 26 to 14 come from the trap base register. See section 1.8.2.8 on page 126).

18. *12-Bit Signed-Immediate* - Bits 11 to 0 in the RIS format for non branch and trap instructions. This field is sign extended and has a data type of *dtp-fixnum*.

1.4.7 Data Type Checking

The data types of either, neither or both source registers of an instruction are checked. The instruction format and the opcode determine what data gets type checked and the Type Check Field in the instruction determines what data type check to perform. In the case of a load or store instruction, the data at the referenced address is also type checked as specified by the opcode. A trap occurs if the result of the data type check is not what was specified by the instruction.

Not all traps on data types are errors, so the trap handler sometimes has to test the data types of the sources and determine what the proper method of handling the trap is. Therefore the trap vector addresses for data type traps include information on the data type being checked, the opcode, and possibly the data type of the sources. See section 1.8 on page 120 for a more detailed description of trapping.

Sunstone provides two methods of data type checking. The first method uses a combination of the opcode, instruction format and the type check field to determine what condition to trap on. The second method utilizes a special instruction, *trap-type*, to specify what conditions to trap on.

In the first method, the Type Check Field, a 5 bit field in the instruction, specifies what data types to trap on. The instruction format determines which Source registers are tested. If the instruction format is RIL or RIS, the data type checking mechanism checks only the Source1 Register. If the instruction format is RR, the data type checking mechanism checks to see if both the Source1 and the Source2 Registers are the same type, and that they are the type specified by the Type Field. An exception to the above occurs when the opcode indicates either a load or store instruction. Load and store instructions use the type field only to check the Source1 register. In the RR format,the load instruction traps if the Source2 register is not a fixnum. The opcode for the load and store instructions specifies what data types of referenced data to trap on and how to handle the trap, see section 1.4.9 on page 27 on memory operations. Also, if the 5 bit type check field in the instruction specifies ≠hardware-arith and the format is RIS or RIL, both sources are checked.

The second method, using the *trap-type* instruction, allows you to explicitly specify what to trap on. The *trap-type* instruction, applies a mask to a 64 bit decoded representation of the data type of Source1. One bit in the instruction selects which half of the 64 bit word the mask is applied to, and another bit selects whether to trap on true or false results. In the RR instruction format, the Source2 register contains the mask data. In the RIL instruction format, the immediate data word contains the 32 bit mask.

Of the two methods, the first method is considerably more efficient. Although the *trap-type* instruction allows more flexibility, it is at the cost of one or two cycles. This is because the second method uses a separate instruction just to test the data types, whereas the first method tests the data types as part of normal instruction execution.

The first method also provides a means to efficiently handle the case where instructions are identical, but the data type trap handler needs to do something different according to the application. An example

of this situation is a *store-data* instruction with the data type check specified as ≠list-locative. This instruction is used when emulating the I-Machine instructions rplaca and pop-lexical-var-n. When emulating pop-lexical-var-n, the trap handler is always an error trap routine. When emulating rplaca, if the data type of the source is *dtp-list-instance* it is not an error. Although the instruction traps in both cases on a source with data type *dtp-list-instance*, the trap handler needs two separate entry points to handle both cases. The most significant bit of the Type Check Field solves this problem. Four bits of the type field specify what data types to trap on, and one bit of the five bit field is used as part of the trap vector, providing 2 unique trap vector addresses for each of the data type traps.

There are a few instructions that have no data type checking on themselves, they are: *read-coproc, write-coproc,load-coproc,store-coproc, branch-next-type, branch-take-type* and *trap-type*. The instructions: *branch-next, branch-take,* and *trap,* all test the source according to the type check defined by the Cond field. See section 1.4 on page 15 for further explanation.

1.4.8 Data Type Setting

The data type is set when it is moved to its destination, either the destination register or a memory location. Typically the data type is set to that of Source1. There are, however, a few exceptions. The *read-mc-reg, read-coproc,* and load instructions, move the full 38 bits read into the destination register. Also, with the exception of the *store-type* instruction, store instructions store into memory all 38 bits of Source2. The *return* and *return-subvert* instructions when in the RR format, get the data type stored in the destination register from Source1. When in the RIL format they get the data type stored in the destination register from the immediate field. When in the RR format, the MOVE instruction sets the data type of the destination register from the Source1 register. When in the RIS or RIL formats, the *move* instruction sets the data type of the result stored in the destination register from the immediate value. For the *move-type* instruction, the data type of the result stored in the destination register comes from the least significant six bits of the Source2 register (Source2<5:0>). When the *move-type* instruction is in the RIS format, the data type of the result stored in the destination register comes from the least significant six bits of the immediate field (Immediate<5:0>).

1.4.9 Memory Operations

Load and Store are the only instructions which access memory. The table 1.4.9 on page 29 lists the many variations of both instructions. Referencing the table reveals the difference between the load instructions is mostly a matter of what data types are trapped on, and what the trap handler does for each of these cases. Note that many of the memory operations are functionally identical to the I-Machine versions, and the hardware traps to allow software to handle all of cases shown. For example the hardware traps to allow the software to load or store indirectly through forwarding pointers. For a more complete description of each instruction, see section 1.5.10 on page 70, and section 1.5.11 on page 89.

Table 8. Memory Operations

Memory Operations							
Operation	I-Data	P-Data	I-Head	P-Head	Null	Bound	Logic
load-data	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-data-iv	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-cdr	-	-	ERR	ERR	-	-	-
load-car-cdr	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-cdr-finish	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-structure	-	-	-	-	-	-	-
load-header	ERR	ERR	-	TRNSPT	ERR	ERR	ERR
load-array	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-coproc	-	TRNSPT	ERR	ERR	ERR	BTRP	LOGIC
load-bind	-	TRNSPT	ERR	ERR	TRNSPT	TRNSPT	TRNSPT
load-scavenge	-	TRNSPT	-	TRNSPT	TRNSPT	TRNSPT	TRNSPT
load-gc-copy	-	-	-	-	-	-	-
load-raw	-	-	-	-	-	-	-
load-ephemeralp	-	-	-	-	-	-	-
load-oldspacep	-	-	-	-	-	-	-
store-data	-	-	ERR	ERR	-	BTRP	LOGIC
store-data-iv	-	-	ERR	ERR	-	BTRP	LOGIC
store-rplacd	-	-	ERR	ERR	-	BTRP	LOGIC
store-array	-	-	ERR	ERR	-	BTRP	LOGIC
store-coproc	-	-	ERR	ERR	-	BTRP	LOGIC
store-bind	-	-	ERR	ERR	-	-	-
store-cdr-nil	-	-	-	-	-	-	-
store-cdr-next	-	-	-	-	-	-	-
store-cdr-normal	-	-	-	-	-	-	-
store-cdr-3	-	-	-	-	-	-	-
store-cdr-reg	-	-	-	-	-	-	-
store-type-reg	-	-	-	-	-	-	-
store-38-bits	-	-	-	-	-	-	-

Legend:

-	Normal action
ERR	This is an error trap.
TRNSPT	If the data type of the word read contains an address in oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.
MTRP	Take a monitor forward trap. However, if the word meets the transport trap condition as described above, take a transport trap instead.
IND	Take an indirect trap to follow the forwarding pointer chain. However, if the word meets the transport trap condition as described above, take a transport trap instead.
BTRP	Take a bound location trap to a routine to search the deep binding cache. However, if the word meets the transport trap condition as described above, take a transport trap instead.
LOGIC	Take a logic variable trap to a routine that replaces the data type of the value read with DTP-EVCP. However, if the word meets the transport trap condition as described above, take a transport trap instead.

Memory Operations Cont.						
Operation	Hfwd	Efwd	1fwd	Evcp	Monitor	GC
load-data	IND	IND	IND	IND	MTRP	ERR
load-data-iv	IND	IND	IND	IND	MTRP	ERR
load-cdr	IND	IND	-	-	-	ERR
load-car-cdr	IND	IND	IND	IND	MTRP	ERR
load-cdr-finish	IND	IND	IND	IND	MTRP	ERR
load-structure	IND	-	-	-	-	ERR
load-header	IND	ERR	ERR	ERR	ERR	ERR
load-array	IND	IND	IND	IND	MTRP	ERR
load-coproc	IND	IND	IND	IND	MTRP	ERR
load-bind	IND	IND	IND	TRNSPT	MTRP	ERR
load-scavenge	TRNSPT	TRNSPT	TRNSPT	TRNSPT	TRNSPT	ERR
load-gc-copy	-	-	-	-	-	ERR
load-raw	-	-	-	-	-	
load-ephemeralp	-	-	-	-	-	
load-oldspacep	-	-	-	-	-	
store-data	IND	IND	IND	IND	MTRP	ERR
store-data-iv	IND	IND	IND	IND	MTRP	ERR
store-rplacd	IND	IND	IND	IND	MTRP	ERR
store-array	IND	IND	IND	IND	MTRP	ERR
store-coproc	IND	IND	IND	IND	MTRP	ERR
store-bind	IND	IND	IND	-	MTRP	ERR
store-cdr-nil	-	-	-	-	-	
store-cdr-next	-	-	-	-	-	
store-cdr-normal	-	-	-	-	-	
store-cdr-3	-	-	-	-	-	
store-cdr-reg	-	-	-	-	-	
store-type-reg	-	-	-	-	-	
store-38-bits	-	-	-	-	-	

Data type classifications:

I-Data *dtp-fixnum, dtp-small-ratio, dtp-single-float, dtp-character, dtp-physical-address, dtp-spare-immediate-1, dtp-instruction*

P-Data *dtp-double-float, dtp-bignum , dtp-big-ratio, dtp-complex, dtp-spare-number, dtp-instance, dtp-list-instance, dtp-array-instance, dtp-string-instance, dtp-nil, dtp-list, dtp-array, dtp-string, dtp-symbol, dtp-locative, dtp-lexical-closure, dtp-dynamic-closure, dtp-compiled-function, dtp-generic-function, dtp-spare-pointer-1, dtp-spare-pointer-2, dtp-bound-location, dtp-logic-variable, dtp-pc, dtp-breakpoint*

I-Head *dpt-header-i*

P-Head *dtp-header-p*

Null *dtp-null*

Bound *dtp-bound-location*

Logic *dtp-logic-variable*

Hfwd *dtp-header-forward*

Efwd *dtp-element-forward*

1fwd *dtp-one-q-forward*

Evcp *dtp-external-value-cell-pointer*

Monitor *dtp-monitor-forward*

GC *dpt-gc-forward*

In calculating the memory address, Source1 and source2 are added together in the RR format for loads and source1 and the MAR are added together for stores. Source1 and the 12-bit-signed-immediate field are added for the RIS format, and source1 and the 38-bit-immediate are added for the RIL format. The data type of the calculated memory address is the data type of source1 for the RR and RIS format instructions, and is the data type of the 38-bit immediate for the RIL format instruction.

1.5 INSTRUCTIONS

There are 58 defined instructions for Sunstone, see table 10 on page 31. The instructions are listed in this document by class, see table 9 on page 30 for a listing of classes.

Table 9. Instruction Classes

Table of Instruction Classes	
Class	Instructions
arithmetic	*add, add-no-overflow, sub, sub-no-overflow, mult*
logical	*and, or, xor*
bit & byte	*ash, lsh, rot, ldb, dpb*
call	*call*
return	*return, return-subvert*
move	*move, move-type*
direct branch	*branch*
conditional	*branch-next, branch-take, trap*
type	*branch-next-type, branch-take-type, trap-type*
load	*load-array, load-bind, load-car-cdr, load-cdr-finish, load-cdr, load-data, load-data-iv, load-ephemeralp, load-gc-copy, load-header, load-oldspacep, load-raw, load-scavenge, load-structure*
store	*store-38-bits, store-array, store-bind, store-cdr-3, store-cdr-next, store-cdr-nil, store-cdr-normal, store-cdr-reg, store-data, store-data-iv, store-rplacd, store-type-reg*
coprocessor	*read-coproc, write-coproc, load-coproc, store-coproc*
mc reg	*read-mc-reg, write-mc-reg*

In calculating the memory address, Source1 and Source2 are added together in the RR format for loads; source1 and the MAR are added together for stores. Source1 and the 12-bit-signed-immediate field are added for the RIS format, and source1 and the 38-bit-immediate are added for the RIL format. The data type of the calculated memory address is the data type of source1 for the RR and RIS format load instructions and for RIS format store instructions; it is the data type of the 38-bit immediate for RIL format instructions; and it is the data type of the MAR for RR format store instructions.

Format	Load-Address	Type	Store-Address	Type
RR	Src1+Src2	Src1	Src1+MAR	MAR
RIS	Src1+Imm-12	Src1	Src1+Imm-12	Src1
RIL	Src1+Imm-38	Imm-38	Src1+Imm-38	Imm-38

1.5 INSTRUCTIONS

There are 58 defined instructions for Sunstone, see table 10 on page 33. The instructions are listed in this document by class, see table 9 on page 32 for a listing of classes.

Table 10. Instructions

Table of Instructions			
Instructions	Class	Opcode	Formats
add	arithmetic	54	RR, RIS, RIL
add-no-overflow	arithmetic	52	RR, RIS, RIL
and	logical	42	RR, RIS, RIL
ash	bit and byte	44	RR, RIS
branch	direct branch	***	DIRECT BRANCH
branch-next	conditional	75	RR, RIS, RIL
branch-next-type	type	74	RR, RIL
branch-take	conditional	65	RR, RIS, RIL
branch-take-type	type	64	RR, RIL
call	call	61	RR, RIL
jcall	call	60	RR, RIL
dpb	bit and byte	51	RR, RIS
ldb	bit and byte	50	RR, RIS
load-array	load	16	RR, RIS, RIL
load-bind	load	10	RR, RIS, RIL
load-car-cdr	load	14	RR, RIS, RIL
load-cdr-finish	load	15	RR, RIS, RIL
load-cdr	load	04	RR, RIS, RIL
load-coproc	coprocessor	17	RR, RIL
load-data	load	12	RR, RIS, RIL
load-data-iv	load	13	RR, RIS, RIL
load-ephemeralp	load	00	RR, RIS, RIL
load-gc-copy	load	06	RR, RIS, RIL
load-header	load	11	RR, RIS, RIL
load-oldspacep	load	01	RR, RIS, RIL
load-raw	load	02	RR, RIS, RIL
load-scavenge	load	07	RR, RIS, RIL
load-structure	load	05	RR, RIS, RIL
lsh	bit and byte	45	RR, RIS
move	move	43	RR, RIS, RIL
move-type	move	66	RR, RIS
mult	arithmetic	56	RR, RIS, RIL
or	logical	40	RR, RIS, RIL
read-coproc	coprocessor	77	RR, RIL
read-mc-reg	mc reg	67	RR, RIS, RIL
return	return	62	RR, RIL
return-subvert	return	63	RR, RIL
rot	bit and byte	46	RR, RIS
store-38-bits	store	26	RR, RIS, RIL
store-array	store	36	RR, RIS, RIL
store-bind	store	30	RR, RIS, RIL
store-cdr-3	store	23	RR, RIS, RIL
store-cdr-next	store	20	RR, RIS, RIL
store-cdr-nil	store	21	RR, RIS, RIL
store-cdr-normal	store	22	RR, RIS, RIL
store-cdr-reg	store	24	RR, RIS, RIL
store-coproc	coprocessor	37	RR, RIL
store-data	store	32	RR, RIS, RIL
store-data-iv	store	33	RR, RIS, RIL
store-rplacd	store	35	RR, RIS, RIL
store-type-reg	store	25	RR, RIS, RIL

***direct-branch has a data type of *dtp-compiled-function* and has no opcode.

Table of Instructions (cont.)			
Instructions	Class	Opcode	Formats
sub	arithmetic	55	RR, RIS, RIL
sub-no-overflow	arithmetic	53	RR, RIS, RIL
trap	conditional	71	RR, RIS, RIL
trap-type	type	70	RR, RIL
write-coproc	coprocessor	57	RR, RIL
write-mc-reg	mc reg	47	RR, RIS, RIL
xor	logical	41	RR, RIS, RIL

Table 10. Instructions

Table of Instructions			
Instructions	Class	Opcode	Formats
add	arithmetic	54	RR, RIS, RIL
add-no-overflow	arithmetic	52	RR, RIS, RIL
and	logical	42	RR, RIS, RIL
ash	bit and byte	44	RR, RIS
branch	direct branch	***	DIRECT BRANCH
branch-next	conditional	75	RR, RIS, RIL
branch-next-type	type	74	RR, RIL
branch-take	conditional	65	RR, RIS, RIL
branch-take-type	type	64	RR, RIL
call	call	61	RR, RIL
jcall	call	60	RR, RIL
dpb	bit and byte	51	RR, RIS
ldb	bit and byte	50	RR, RIS
load-array	load	16	RR, RIS, RIL
load-bind	load	07	RR, RIS, RIL
load-car-cdr	load	14	RR, RIS, RIL
load-cdr-finish	load	15	RR, RIS, RIL
load-cdr	load	10	RR, RIS, RIL
load-coproc	coprocessor	17	RR, RIL
load-data	load	12	RR, RIS, RIL
load-data-iv	load	13	RR, RIS, RIL
load-ephemeralp	load	00	RR, RIS, RIL
load-gc-copy	load	04	RR, RIS, RIL
load-header	load	06	RR, RIS, RIL
load-oldspacep	load	01	RR, RIS, RIL
load-raw	load	02	RR, RIS, RIL
load-scavenge	load	05	RR, RIS, RIL
load-structure	load	11	RR, RIS, RIL
lsh	bit and byte	45	RR, RIS
move	move	43	RR, RIS, RIL
move-type	move	66	RR, RIS
mult	arithmetic	56	RR, RIS, RIL
or	logical	40	RR, RIS, RIL
read-coproc	coprocessor	77	RR, RIL
read-mc-reg	mc reg	67	RR, RIS, RIL
return	return	62	RR, RIL
return-subvert	return	63	RR, RIL
rot	bit and byte	46	RR, RIS
store-38-bits	store	26	RR, RIS, RIL
store-array	store	36	RR, RIS, RIL
store-bind	store	27	RR, RIS, RIL
store-cdr-3	store	23	RR, RIS, RIL
store-cdr-next	store	20	RR, RIS, RIL
store-cdr-nil	store	21	RR, RIS, RIL
store-cdr-normal	store	22	RR, RIS, RIL
store-cdr-reg	store	24	RR, RIS, RIL
store-coproc	coprocessor	37	RR, RIL
store-data	store	32	RR, RIS, RIL
store-data-iv	store	33	RR, RIS, RIL
store-rplacd	store	35	RR, RIS, RIL
store-type-reg	store	25	RR, RIS, RIL

*** direct-branch has a data type of *dtp-compiled-function* and has no opcode.

Table of Instructions (cont.)			
Instructions	Class	Opcode	Formats
sub	arithmetic	54	RR, RIS, RIL
sub-no-overflow	arithmetic	55	RR, RIS, RIL
trap	conditional	66	RR, RIS, RIL
trap-type	type	65	RR, RIL
write-coproc	coprocessor	57	RR, RIL
write-mc-reg	mc reg	67	RR, RIS, RIL
xor	logical	56	RR, RIS, RIL

1.5.1 Arithmetic Operations

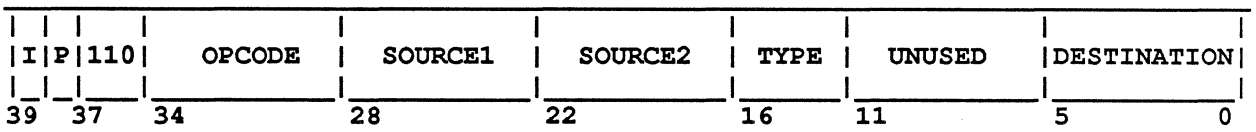
The arithmetic operations are as follows: *add, add-no-overflow, sub, sub-no-overflow, mult.* All of these operations are available in all three formats: RR, RIS, and RIL. The general description of these operations is:

Destination ← value op Source1

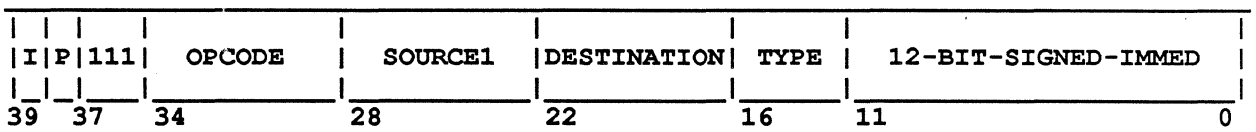
where OP is one of the arithmetic operations, and value is either Source2 in the RR format or the immediate field in the RIS and RIL formats. The data type, of both Source1 (and source2 in the RR format) is checked according the the value of the instructions type field. The type field of the instruction for *add, sub and mult* will usually be ≠hardware-arith. The type field of the instruction for *add-no-overflow* and *sub-no-overflow* will typically be ≠fixnum. If ≠hardware-arith is specified as the type check, both sources will be checked regardless of the instruction format. For all other type checking only source1 is type checked, except in the RR format when both source1 and source2 are type checked. See the description of the type field on page 22. The data type of the result stored in the register specified by the destination field in the instruction word is the data type of source1. See figure 7 on page 33.

Figure 7. Arithmetic Operation Formats

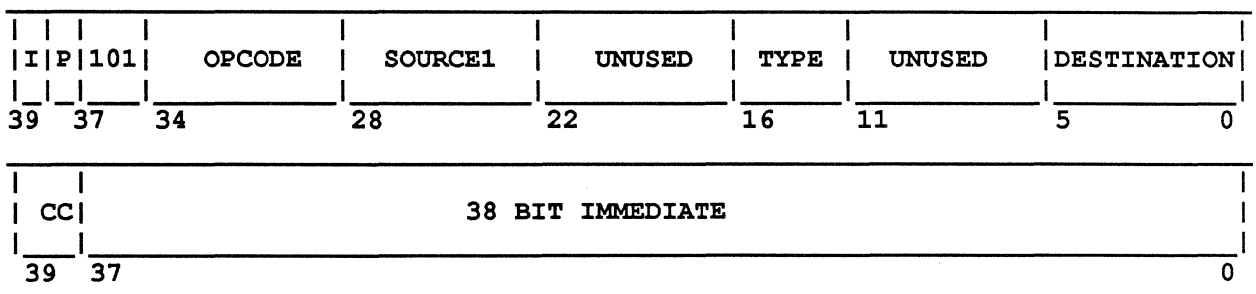
RR:



RIS:



RIL:



ADD

Opcode: 54

Formats:

```
RR   destination ← source1 + source2
RIS  destination ← source1 + 12-bit-signed-immediate
RIL  destination ← source1 + 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠hardware-arith which will test the data type of both sources even in the RIS and RIL formats. See the examples below and section 1.8.2 on page 121.

Traps: Overflow trap if the addition results in an overflow condition. If a floating point coprocessor is present then coprocessor traps may occur.

Description: Implements the Lisp + operation, and adds the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked, except when the type check is ≠ hardware-arith With type check ≠hardware-arith, fixnum operations are performed by the normal data path. Floating point operations will trap unless a coprocessor is present to handle that operation.

Examples:

```
(setq baz (+ foo bar))  add   baz ← foo,bar      ≠hardware-arith   ;RR
(setq foo (1+ foo))     add   foo ← foo,#1       ≠hardware-arith   ;RIS
(setq foo (1- foo))     add   foo ← foo,#-1      ≠hardware-arith   ;RIS
(setq baz (+ foo 2047)) add   baz ← foo,#2047    ≠hardware-arith   ;RIS
(setq foo (- foo 1234)) add   foo ← foo,#-1234   ≠hardware-arith   ;RIS
(setq baz (+ foo 4096)) add   baz ← foo,#4096    ≠hardware-arith   ;RIL
```

ADD-NO-OVERFLOW

Opcode: 52

Formats:

```
RR   destination ← source1 + source2
RIS  destination ← source1 + 12-bit-signed-immediate
RIL  destination ← source1 + 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Performs twos complement addition on the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked.

Examples:

```
(setq baz (sys:%32-bit-plus foo bar))
          add-no-overflow   baz ← foo,bar    ≠fixnum   ;RR

(setq baz (sys:%32-bit-plus foo 2047))
          add-no-overflow   baz ← foo,#2047  ≠fixnum   ;RIS

(setq baz (sys:%32-bit-plus foo 4096))
          add-no-overflow   baz ← foo,#4096  ≠fixnum   ;RIL
```

SUB

Opcode: 55

Formats:

```
RR   destination ← source2 - source1
RIS  destination ← 12-bit-signed-immediate - source1
RIL  destination ← 32-bit-signed-immediate - source1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠hardware-arith which will test the data type of both sources even in the RIS and RIL formats. See the examples below and section 1.8.2 on page 121.

Traps: Overflow trap if the subtraction results in an overflow condition. If a floating point coprocessor is present then coprocessor traps may occur.

Description: Implements the Lisp - operation and subtracts the contents of the register specified by source1 from the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked, except when the type check is ≠ hardware-arith. NOTE: None of the above formats support subtracting an immediate value from the contents of a register. To do so you must use the add instruction. Adding a negative immediate value is equivalent to subtracting a like value with a positive sign. See example below. With type check ≠hardware-arith, fixnum operations are performed by the normal data path. Floating point operations will trap unless a coprocessor is present to handle that operation.

Examples:

```
(setq baz (- bar foo))  sub  baz ← foo,bar ≠hardware-arith  ;RR
(setq baz (- 2047 foo)) sub  baz ← foo,#2047    ≠hardware-arith  ;RIS
(setq baz (- 4096 foo)) sub  baz ← foo,#4096    ≠hardware-arith  ;RIL

Subtracting an immediate value from the contents of a register:

(setq foo (- foo 1234)) add   foo ← foo,#-1234   ≠hardware-arith   ;RIS
```

SUB-NO-OVERFLOW

Opcode: 53

Formats:

```
RR   destination ← source2 - source1
RIS  destination ← 12-bit-signed-immediate - source1
RIL  destination ← 32-bit-signed-immediate - source1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Performs twos complement subtraction on the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked. NOTE: None of the above formats support subtracting an immediate value from the contents of a register. To do so you must use the add-no-overflow instruction. Adding a negative immediate value is equivalent to subtracting a like value with a positive sign. See example below.

Examples:

```
(setq baz (sys:%32-bit-difference bar foo))
                sub-no-overflow   baz ← foo,bar ≠fixnum   ;RR

(setq baz (sys:%32-bit-difference 2047 foo))
                sub-no-overflow   baz ← foo,#2047 ≠fixnum ;RIS

(setq baz (sys:%32-bit-difference 4096 foo))
                sub-no-overflow   baz ← foo,#4096 ≠fixnum ;RIL

Subtracting an immediate value from the contents of a register:

(setq baz (sys:%32-bit-plus foo 2047))
                add-no-overflow   baz ← foo,#2047 ≠fixnum ;RIS
```

MULT

Opcode: 56

Formats:

```
RR   destination ← source1 * source2
RIS  destination ← source1 * 12-bit-signed-immediate
RIL  destination ← source1 * 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠hardware-arith which will test the data type of both sources even in the RIS and RIL formats. See the examples below and section 1.8.2 on page 121.

Traps: Overflow trap if the multiplication results in an overflow condition. If a floating point coprocessor is present then coprocessor traps may occur.

Description: Implements the Lisp * function on the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked, except when the type check is ≠ hardware-arith. No multiply operations are performed by the normal data path. All *mult* instructions will trap unless a coprocessor is present that can handle the operation. The coprocessor might handle fixnum and/or single-float multiplications.

Examples:

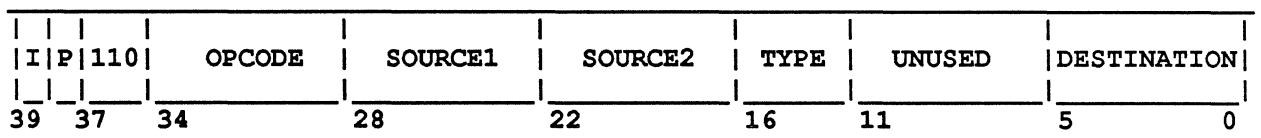
```
(setq baz (* foo bar))  mult   baz ← foo,bar     ≠hardware-arith   ;RR
(setq baz (* foo 2047)) mult   baz ← foo,#2047   ≠hardware-arith   ;RIS
(setq baz (* foo 4096)) mult   baz ← foo,#4096   ≠hardware-arith   ;RIL
```

1.5.2 Logical Operations

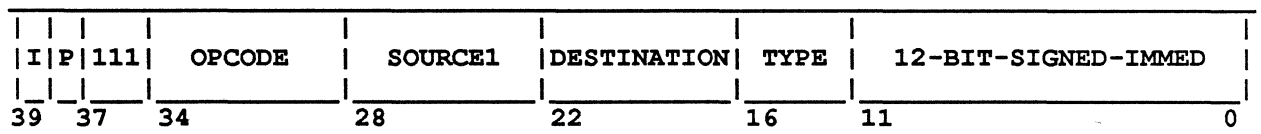
The logical operations are: *and, or, xor*. All of these operations are available in all three formats: RR, RIS, and RIL. The general description of these operations is: Destination ← Source1 op value where OP is one of the Logical Operations, and value is either Source2 in the RR format or the immediate field in the RIS and RIL formats. The data type of Source1 (and source2 in the RR format) is checked according the the value of the instructions type field. The type field of the instruction will usually be ≠fixnum, which will cause a trap if the data type of source1 (and source2 in the RR format) are not the same, or if the data type is not *dtp-fixnum*. The data type of the result stored in the destination is the same as that of source1. See figure 8 on page 39.

Figure 8. Logical Operation Formats

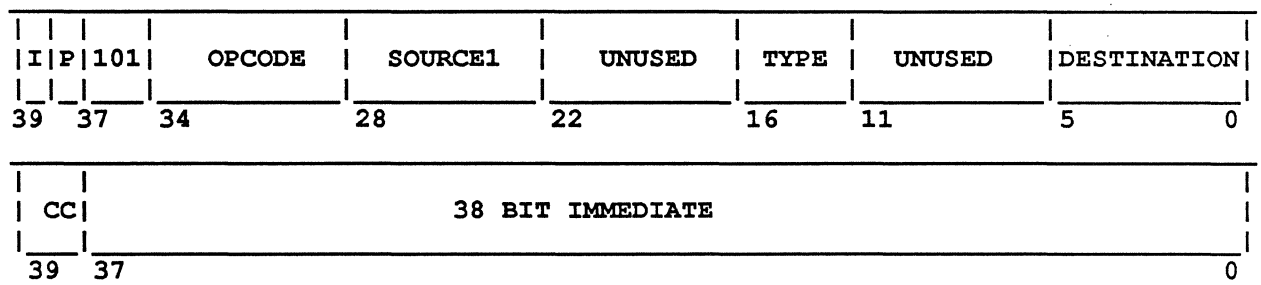
RR:



RIS:



RIL:



AND

Opcode: 42

Formats:

```
RR   destination ← source1 and source2
RIS  destination ← source1 and 12-bit-signed-immediate
RIL  destination ← source1 and 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None

Description: Performs 32-bit bitwise logical AND between the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked.

Examples:

```
(setq baz (logand foo bar))   and      baz ← foo,bar   ≠fixnum   ;RR
(setq baz (logand foo 2047))  and      baz ← foo,#2047 ≠fixnum   ;RIS
(setq baz (logand foo 4096))  and      baz ← foo,#4096 ≠fixnum   ;RIL
```

OR

Opcode: 40

Formats:

```
RR   destination ← source1 or source2
RIS  destination ← source1 or 12-bit-signed-immediate
RIL  destination ← source1 or 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None

Description: Performs 32-bit bitwise logical OR between the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked.

Examples:

```
(setq baz (logior foo bar))    or  baz←  foo,bar     ≠fixnum ;RR
(setq baz (logior foo 2047))   or  baz ← foo,#2047   ≠fixnum ;RIS
(setq baz (logior foo 4096))   or  baz ← foo,#4096   ≠fixnum ;RIL
```

XOR

Opcode: 41

Formats:

```
RR   destination ← source1 xor source2
RIS  destination ← source1 xor 12-bit-signed-immediate
RIL  destination ← source1 xor 32-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None

Description: Performs 32-bit bitwise exclusive-or between the contents of the register specified by source1 and the immediate field (for RIS and RIL formats) or the contents of the register specified by source2 (for RR format) and stores the result in the register specified by destination. The immediate fields are not type checked.

Examples:

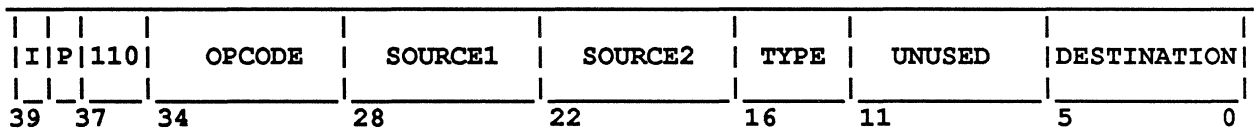
```
(setq baz (logxor foo bar))   xor      baz ← foo,bar   ≠fixnum   ;RR
(setq baz (logxor foo 2047))  xor      baz ← foo,#2047 ≠fixnum   ;RIS
(setq baz (logxor foo 4096))  xor      baz ← foo,#4096 ≠fixnum   ;RIL
```

1.5.3 Bit and Byte Operations

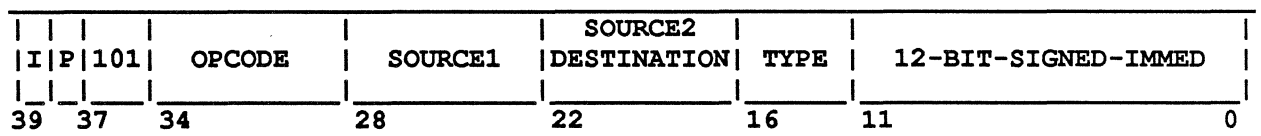
The bit and byte operations are: *ash, lsh, rot, ldb, dpb.* All of these operations are available in only RR and RIS formats, and are not available in the RIL format. The data type of Source1 (and source2 for RR format) is checked according the the value of the instructions type field. The type field of the instruction will usually be ≠fixnum, which will cause a trap if the data type of source1 (or Source2 in the RR format) are not both fixnum. The data type of the result is the data type of source1. See figure 9 on page 43.

Figure 9. Bit and Byte Operation Formats

RR:



RIS:



ASH

Opcode: 44

Formats:

```
RR   destination ← Shift source1 by source2
RIS  destination ← Shift source1 by 12-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: Traps on overflow. An overflow trap will occur if the shift amount is greater than 31, or if any significant bits are lost in the course of a left shift.

Description: Shifts the contents of the source1 register arithmetically according to the contents of the source2 register (or the immediate field if RIS format) and places the result in the destination register. Shift right if the sign of the source2 register (or immediate field) is negative, shift left if the sign is positive. A right shift will shift in the sign bit; a left shift will shift in zeros. The immediate fields are not type checked.

Examples:

```
(setq baz (ash foo bar))    ash  baz ← foo,bar   ≠fixnum   ;RR
(setq baz (ash foo 10))     ash  baz ← foo,#10   ≠fixnum   ;RIS
```

LSH

Opcode: 45

Formats:

```
RR   destination ← Shift source1 by source2
RIS  destination ← Shift source1 by 12-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Shifts the contents of the source1 register logically according to the contents of the source2 register (or the immediate field if RIS format) and places the result in the destination register. Shift right if the sign of the source2 register (or immediate field) is negative, shift left if the sign is positive. Zeros are shifted in. A |shift amount| > 31 results in 0. The immediate fields are not type checked.

Examples:

```
(setq baz (lsh foo bar))    lsh  baz ← foo,bar   ≠fixnum   ;RR
(setq baz (lsh foo 10))     lsh  baz ← foo,#10   ≠fixnum   ;RIS
```

ROT

Opcode: 46

Formats:

```
RR   destination ← Rotate source1 by source2
RIS  destination ← Rotate source1 by 12-bit-signed-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Rotates the contents of the source1 register left according to the contents of the source2 register (or the immediate field if RIS format) and places the result in the destination register. Only the least significant 5 bits are considered for the rotate amount, in part this means that the sign bit is ignored. The immediate fields are not type checked.

Examples:

```
(setq baz (rot foo bar))   rot   baz ← foo,bar   ≠fixnum   ;RR
(setq baz (rot foo 10))    rot   baz ← foo,#10   ≠fixnum   ;RIS
```

LDB

Opcode: 50

Formats:

```
RR   destination ← (ldb source2 source1)
RIS  destination ← (ldb 12-bit-signed-immediate source1)
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Helps to implement the lisp LDB function. Extracts the field specified by the contents of the source2 register (or the immediate field for RIS format) from the contents of the register specified by source1 and places it in the register specified by the destination field. Only the least significant 10 bits of the contents specified by the source2 field (or the immediate field) are used to specify the byte to be extracted.Bits 0 through 4 of the byte specifier indicate the location of the bottom bit of the field, and bits 5 through 9 specify the (field-size - 1). The byte specifier is not type checked.

Examples:

```
(setq baz (ldb (byte 10 0) foo)) ldb  baz ← foo,#440 ≠fixnum;RIS
```

DPB

Opcode: 51

Formats:

```
RR   destination ← (dpb source1 byte-rotate-register source2)
RIS  source2     ← (dpb source1 12-bit-signed-immediate source2)
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for ≠fixnum. See the examples below and section 1.8.2 on page 121.

Traps: None.

Description: Helps to implement the lisp DPB function. Deposits the contents of the source1 register into a field of the contents of the register specified by the source2 field, and places the result in the register specified by the destination field. The field where the value is being deposited is specified by the byte-rotate-register for the RR format, or the immediate field for the RIS format. In either case only the least significant 10 bits are considered. Bits 0 through 4 of the byte specifier indicate the location of the bottom bit of the field, and bits 5 through 9 specify the (field-size - 1). The byte specifier is not type checked.

Examples:

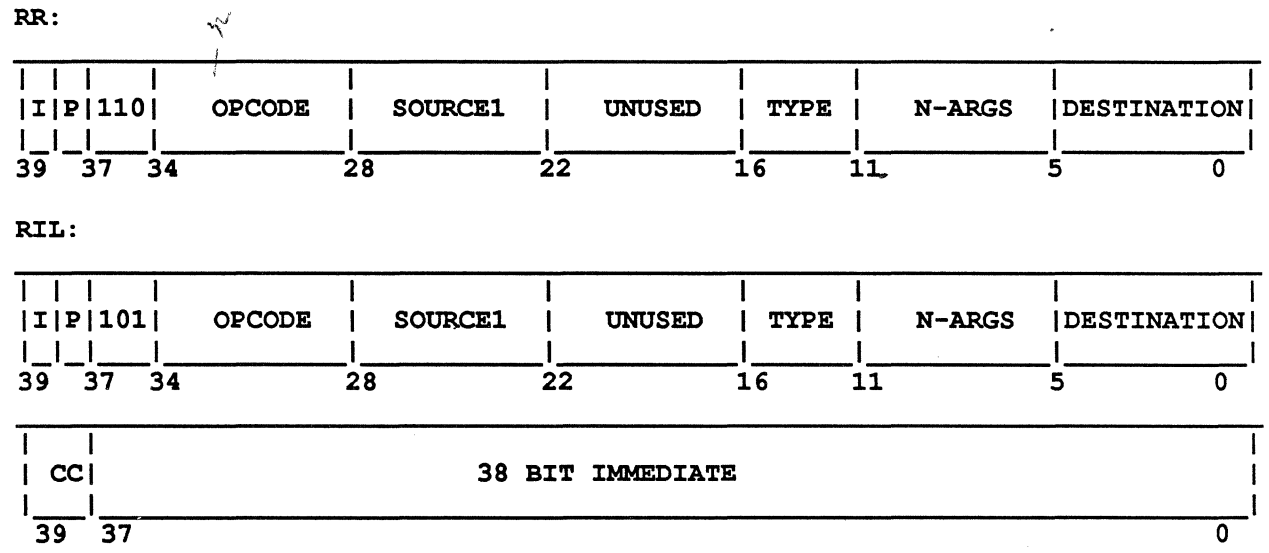
```
(setq foo (dpb bar (byte 10 0) foo))
        dpb     foo ← bar,#440,foo  ≠fixnum   ;RIS

(setq baz (dpb bar (byte 10 0) foo))
        move    baz ← foo
        dpb     baz ← bar,#440,baz  ≠fixnum   ;RIS
```

1.5.4 Call Operations

The *call* and *jcall* instruction are available in formats: RR and RIL. Data Type check of source1 according to the type field, this will typically test for none in the RIL format and for a legal PC type in the RR format. See figure 10 on page 49.

Figure 10. Call Instruction Formats



Call

Opcode: 61

Formats:

```
RR   Destination ← Source1,
     N-Args      ← n-args<4:0> if n-args<5> = 1

RIL  PC          ← 38-bit-immediate,
     Destination ← Source1
     N-args      ← n-args<4:0> if n-args<5> = 1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none. See the examples below and in section 1.8.2 on page 121.

Traps: If the window buffer overflows. Window Buffer Overflow occurs if the Window Buffer Pointer = Window Buffer Overflow Limit, before the instruction increments the Window Buffer Pointer.

Description: The PC is set according to the format (see above). The return address is pushed onto the return stack, the return address is PC+1 for the RR format, and is PC+2 for the RIL format. The window pointer is incremented by 1. The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. The contents of the register addressed by source1 is moved into the register specified by the destination field in the instruction. In the case of the RR instruction the destination is typically the PC. Note that indirect calls are implemented by having the call point to the word containing the address of the thing to be called, like the function cell of a symbol, since this cell will have a *dtp-compiled-function*, it will branch to the desired destination (see section 1.5.7 on page 59 for a description of direct branches). The immediate fields are not type checked. In the RR format the destination is typically PC.

Examples:

```
(assoc item a-list)
        move    a0 ← item
        call    @assoc,n-args ← #1,a1  ← a-list  ; RIL

(funcall function item a-list)
        move    a0 ← item
        move    a1 ← a-list
        call    function,n-args ← #2    ≠Compiled-function  ; RR
```

Jcall

Opcode: 60

Formats:

```
RR   Destination ← Source1,
     N-Args      ← n-args<4:0> if n-args<5> = 1

RIL  Destination ← Source1,
     N-args      ← n-args<4:0> if n-args<5> = 1

     PC          ← 38-bit-immediate,
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none.

Traps: None.

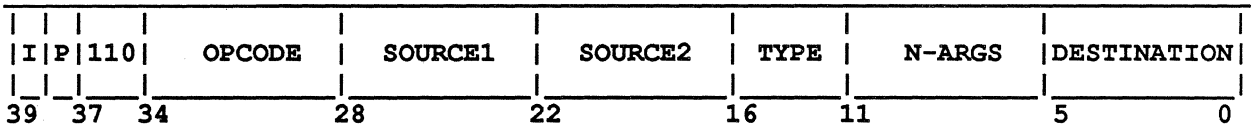
Description: The PC is set according to the format (see above). The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. And in the RIL format, the contents of the register specified by source1 are moved into the register specified by the destination field in the instruction. The immediate fields are not type checked. In the RR format the destination is typically PC. This instruction is intended to support Tail Recursion Elimination (TRE). The *Jcall* instruction differs from the call instruction in that it does not increment the Window Buffer Pointer.

1.5.5 Return Operations

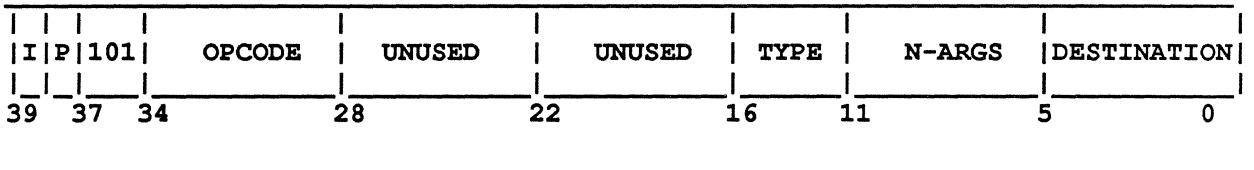
The return operations are *return* and *return-subvert* instructions. The return instructions are available in two formats: RR and RIL. Data Type check of source1 according to the type field, this will typically test for none. The data type of the result depends on the format, in the RR format it will be the data type of source1, and in the RIL format it will be the data type of the 38-bit immediate. See figure 11 on page 52.

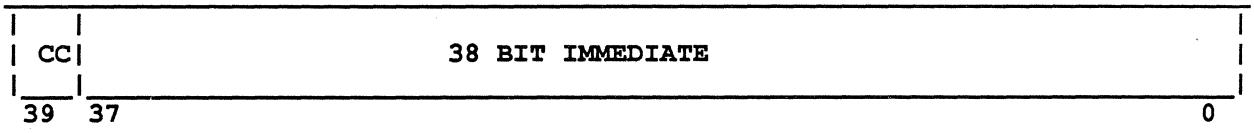
Figure 11. Return Operation Formats

RR:



RIL:





Return

Opcode: 62

Formats:

```
RR      PC          ← Top of Return Stack,
        destination ← source1<37:32>|source2<31:0>
        N-args      ← n-args<4:0> if n-args<5> = 1

RIL     PC          ← Top of Return Stack,
        destination ← 38-bit-immediate
        N-args      ← n-args<4:0> if n-args<5> = 1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none. See the examples below and the section section 1.7.2 on page 118.

Traps: Take a Window Buffer Underflow Trap if the Window Buffer underflows. See section 1.7.2.7, General Traps, on page 122 for a description of the trap.

Description: The PC is set to the value on the top of the return stack. The Window Buffer Pointer (which is also the return stack pointer) is decremented by 1. The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. And the register specified by the destination field of the instruction is loaded with, depending on format, merging of source1 data type and source2 data (RR format), or the 38-bit-immediate (RIL format). The immediate field is not type checked.

Examples:

```
(defun foo (a) a)   return   r15 ← r0 n-args ← 1   ;a
```

Return

Opcode: 62

Formats:

```
RR      PC           ← Top of Return Stack,
        destination  ← source1<37:32>|source2<31:0>
        N-args       ← n-args<4:0> if n-args<5> = 1

RIL     PC           ← Top of Return Stack,
        destination  ← 38-bit-immediate
        N-args       ← n-args<4:0> if n-args<5> = 1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none. See the examples below and the section section 1.8.2 on page 121.

Traps: If the window buffer underflows. Window Buffer Underflow occurs if the Window Buffer Pointer = Window Buffer Underflow Limit, before the instruction decrements the Window Buffer Pointer.

Description: The PC is set to the value on the top of the return stack. The Window Buffer Pointer (which is also the return stack pointer) is decremented by 1. The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. And the register specified by the destination field of the instruction is loaded with, depending on format, merging of source1 data type and source2 data (RR format), or the 38-bit-immediate (RIL format). The immediate field is not type checked.

Examples:

```
(defun foo (a) a)    return    r15 ← r0 n-args ← 1    ;a
```

Return-subvert

Opcode: 63

Formats:

```
RR      PC          ← Top of Return Stack,
        destination ← source1<37:32>|source2<31:0>
        N-args      ← n-args<4:0> if n-args<5> = 1

RIL     PC          ← Top of Return Stack,
        destination ← 38-bit-immediate
        N-args      ← n-args<4:0> if n-args<5> = 1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none. See the examples below and section 1.8.2 on page 121.

Traps: If the window buffer underflows. Window Buffer Underflow occurs if the Window Buffer Pointer = Window Buffer Underflow Limit, before the instruction decrements the Window Buffer Pointer.

Description: The PC is set to the value on the top of the return stack. The Window Buffer Pointer (which is also the return stack pointer) is decremented by 1. The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. And the register specified by the destination field of the instruction is loaded with, depending on format, merging of source1 data type and source2 data (RR format), or the 38-bit-immediate (RIL format). The immediate field is not type checked.

The subvert bit is set in the status control register. In essence, this causes the next instruction to operate like a move instruction, that is, it subverts (or perverts) the execution of the instruction being returned to. An example of where this might be used is upon returning from a data type trap of an add operation where the operands were both bignums. The trap routine performs the add of the bignums in software and does a *return-subvert*. The instruction returned to is the trapped instruction, the add, which has already been performed, and whose operation needs to be subverted. The immediate fields are not type checked. The usual use of this instruction is in the RR format with the destination being the Status Control Register, and the source being the saved version of the Status Control Register.

All arithmetic, logical, bit & byte, read-coproc, and load instructions that are subverted will move the contents of the trap-result register into the destination register specified by the subverted instruction. All side-effects of the MAR, byte rotate register, and status control register are inhibited. Data type checking of the operands is inhibited.

Subverted call and return instructions operate normally except that the contents of the Trap Result register is moved into the specified destination register. For the calls, the data type checking of the Trap Result Register is done as specified by the type field of the instruction being subverted. For the returns, data type checking of the operands is inhibited.

Return-subvert

Opcode: 63

Formats:

```
RR      PC          ← Top of Return Stack,
        destination ← source1<37:32>|source2<31:0>
        N-args      ← n-args<4:0> if n-args<5> = 1

RIL     PC          ← Top of Return Stack,
        destination ← 38-bit-immediate
        N-args      ← n-args<4:0> if n-args<5> = 1
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none. See the examples below and section 1.7.2 on page 118.

Traps: Take a Window Buffer Underflow Trap if the Window Buffer underflows. See section 1.7.2.7, General Traps, on page 122 for a description of the trap.

Description: The PC is set to the value on the top of the return stack. The Window Buffer Pointer (which is also the return stack pointer) is decremented by 1. The N-args register is loaded with the least significant five bits of the n-args field if the most significant bit of the field is a 1. And the register specified by the destination field of the instruction is loaded with, depending on format, merging of source1 data type and source2 data (RR format), or the 38-bit-immediate (RIL format). The immediate field is not type checked.

The subvert bit is set in the status control register. In essence, this causes the next instruction to operate like a move instruction, that is, it subverts (or perverts) the execution of the instruction being returned to. An example of where this might be used is upon returning from a data type trap of an add operation where the operands were both bignums. The trap routine performs the add of the bignums in software and does a *return-subvert*. The instruction returned to is the trapped instruction, the add, which has already been performed, and whose operation needs to be subverted. The immediate fields are not type checked. The usual use of this instruction is in the RR format with the destination being the Status Control Register, and the source being the saved version of the Status Control Register.

All arithmetic, logical, bit & byte, read-coproc, and load instructions that are subverted will move the contents of the trap-result register into the destination register specified by the subverted instruction. All side-effects of the MAR, byte rotate register, and status control register are inhibited. Data type checking of the operands is inhibited.

Subverted call and return instructions operate normally except that the contents of the Trap Result register is moved into the specified destination register. For the calls, the data type checking of the Trap Result Register is done as specified by the type field of the instruction being subverted. For the returns, data type checking of the operands is inhibited.

Conditional instructions that are subverted use the least significant bit of the trap-result register and the most significant bit of the cond field in the instruction, bit 16, to determine if the branch is to be taken. If the two bits are different the branch is taken, if the bits are the same then the branch is not taken. Data type checking of the operands is inhibited.

Store instructions that are subverted are just treated as nops. Data type checking of the operands is inhibited

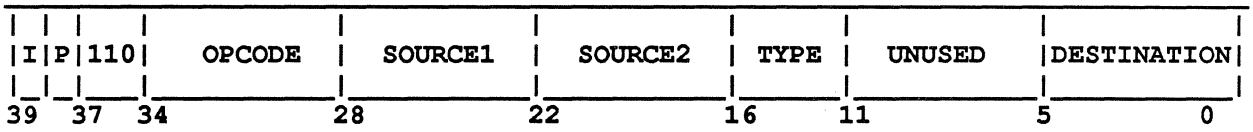
The effect of subverting write-coproc, load-coproc, store-coproc, direct branches and type instructions is undefined.

1.5.6 Move Operations

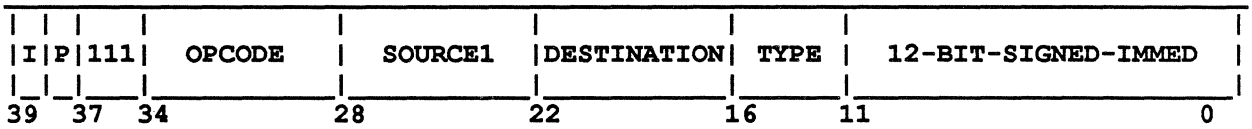
The move operations are: *move*, and *move-type*. The *move* instruction is available in all three formats, and the *move-type* instruction is only available in the RR and RIS formats. Data Type check of Source1 (and Source2 in the RR format) according to the type field, this will typically test for none. The data type of the result depends on the format and the instruction. For the *move* instruction, in the RR format it will be the data type of source1, in the RIS format it will be fixnum, and in the RIL format it will be the data type of the 38-bit immediate. For the *move-type* instruction,the data type of the result will be the 6 bits of the immediate field (RIS format) or the least significant 6 bits of the source2 register.See figure 12 on page 56.

Figure 12. Move Operation Formats

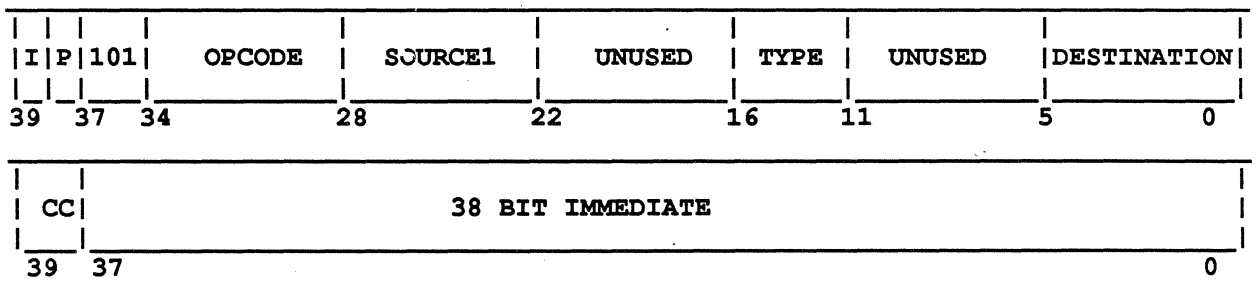
RR: *move, move-type*



RIS: *move, move-type*



RIL: *move*



Move

Opcode: 43

Formats:

```
RR   destination ← source1<37:32>|source2<31:0>
RIS  destination ← 12-bit-immediate
RIL  destination ← 38-bit-immediate
```

Source Data Type Traps: The type field of the instruction specifies what data types of Source1 (and Source2 in the RR Format) to trap on. This field will typically test for none.

Traps: None

Description: If the Format is immediate, moves the immediate value into the destination register. If the Format is RR, merges the type of source1 with the data of source2 and stores the result in the register specified by the destination field of the instruction.

Move-type

Opcode: 66

Formats:

```
RR      destination ← source2<5:0>  |source1<31:0>
RIS     destination ← immediate<5:0>|source1<31:0>
```

Source Data Type Traps: The type field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on. This field will typically test for none.

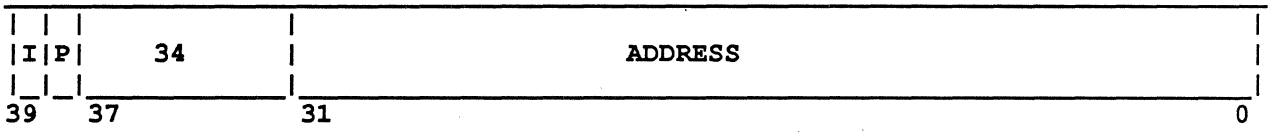
Traps: None

Description: Source1 data is moved into the register specified by the destination field of the instruction with the data type of the result being the least significant 6 bits of the immediate field (RIS format) or the source2 register.

1.5.7 Direct Branch Operation

Instructions with data type of *dtp-compiled-function* are interpreted by the hardware to be direct branch operations. See figure 13 on page 59.

Figure 13. Direct Branch Format



Branch

Opcode: None, it has a data type of *dtp-compiled-function*

Formats: `Direct branch` `PC ← address`

Traps: None

Description: The address field is used as the address of the next instruction.

1.5.8 Conditional Operations

The conditional operations are: *branch-next*, *branch-take*, and *trap*. Available in all three Formats. Data Type check of Source1 (and Source2 in the RR Format) according to the Cond Field. The Cond Field specifies the conditions and type trap enables listed.

If the cond of the operands is not true, the branch is not taken. If the branch is to be taken, the 12 bit page offset is inserted into the low 12 bits of the current PC as the branch address. The -take is an indication that the branch is expected to be taken, that is, the result will be true. The -next is an indication that the branch is expected not to be taken, that is, the cond result will not be true. See figure 14 on page 62.

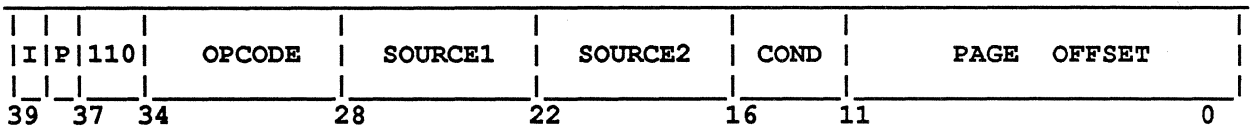
In the RIS format, the 6 bit field is a sign extended field.

In the case of the trap instruction, a trap occurs if the condition is true. If the trap occurs, the page offset field is used as the low 12 bits of the trap address; the high bits are taken from the trap base register, and bits 13:12 of the address are zeros.

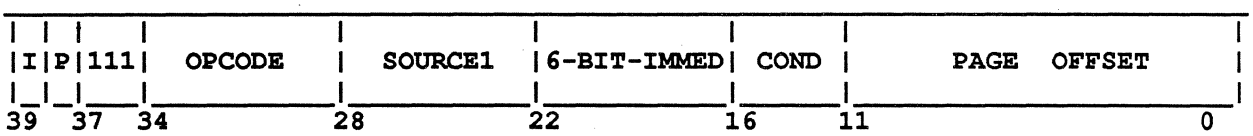
All fixnum compares are handled by the normal data path. The compares that specify ≠hardware-arith type checks will trap for floating point operations unless a coprocessor is present that can handle the operation.

Figure 14. Conditional Operation Formats

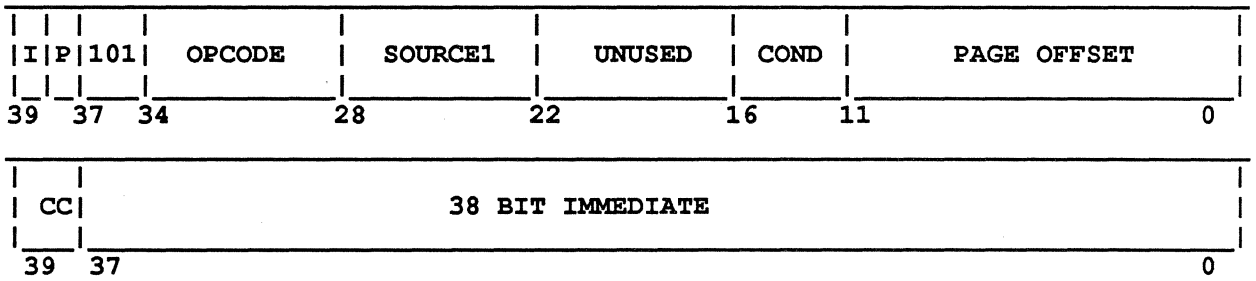
RR:



RIS:



RIL:



Branch-next

Opcode: 75

Formats:

```
RR    if source1 cond source2
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIS   if source1 cond 6-bit-immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIL   if source1 cond 38 bit immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+2
```

Source Data Type Traps: The cond field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on.

Traps: None.

Description: The two operands are compared as specified by the cond field in the instruction. If the result of the cond is T, the branch is taken, the page offset is inserted into the low 12 bits of the current PC. If the result of the COND is not T, the next instruction is executed. This instruction is used when the result of the cond is anticipated to be False.

Branch-next

Opcode: 71

Formats:

```
RR    if source1 cond source2
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIS   if source1 cond 6-bit-immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIL   if source1 cond 38 bit immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+2
```

Source Data Type Traps: The cond field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on.

Traps: None.

Description: The two operands are compared as specified by the cond field in the instruction. If the result of the cond is T, the branch is taken, the page offset is inserted into the low 12 bits of the current PC. If the result of the COND is not T, the next instruction is executed. This instruction is used when the result of the cond is anticipated to be False.

Branch-take

Opcode: 65

Formats:

```
RR    if source1 cond source2
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIS   if source1 cond 6-bit-immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIL   if source1 cond 38 bit immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+2
```

Source Data Type Traps: The cond field of the instruction specifies what data types of source1 (and source2 in the RR format) to trap on.

Traps: None.

Description: The two operands are compared as specified by the cond field in the instruction. If the result of the cond is T, the branch is taken, the page offset is inserted into the low 12 bits of the current PC. If the result of the COND is not T, the next instruction is executed. This instruction is used when the result of the cond is anticipated to be True.

Trap

Opcode: 71

Formats:

```
RR    if source1 cond source2
         PC ← <11111>|trap-base-register|<00>|page-offset, TRAP
      else
         PC ← PC+1

RIS   if source1 cond 6-bit-immediate
         PC ← <11111>|trap-base-register|<00>|page-offset, TRAP
      else
         PC ← PC+1

RIL   if source1 cond 38 bit immediate
         PC ← <11111>|trap-base-register|<00>|page-offset, TRAP
      else
         PC ← PC+2
```

Source Data Type Traps: Specified by the COND field,these traps check both operands to be of the same data type and traps if they are not. Checks both operands to be of the type specified by the cond field, and traps if they are not.

Traps: If the condition results in true, trap.

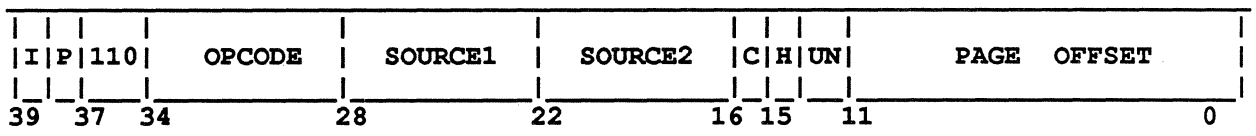
Description: The two operands are compared as specified by the cond field in the instruction. If the result of the cond is T, the incremented PC is pushed onto the return stack, the page offset is inserted into the low 12 bits of the new PC, the most significant five bits of the new pc are ones, bits 26:14 of the new PC is the trap base register, and bits 13 and 12 are zero and the WBP is incremented. If the result of the COND is not T, the next instruction is executed. It is anticipated that the trap will not occur.

1.5.9 Type Operations

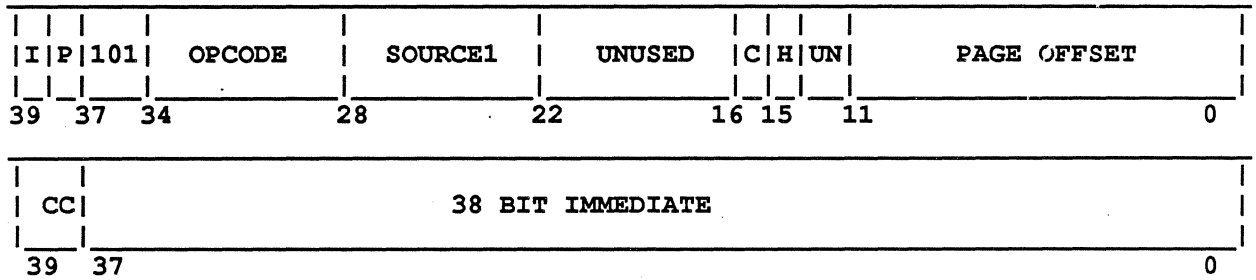
The type operations are: *branch-next-type*, *branch-take-type*,and *type-trap*. The type operations are available in the RR and RIL formats. These operations test the data type of Source1according to a mask obtained from the second operand, which is the Source2 register contents in RR format or the low 32 bits of the immediate in RIL format. The data type of the first operand is decoded to set a single bit in a 64 bit field, e.g. if the data type of source1 is DTP-CHARACTER (#o43), then bit #o43 will be set in the 64 bit field. This 64 bit field is then LOGANDed with the 32 bit mask; depending on the H bit in the instruction, the mask is either LOGANDed with the high half (representing data types #o40 to #o77), if H is one, or the low half (representing data types #o0 to #o37) if H is zero. If the result of the LOGAND is non-zero, then the instruction's result is true. The C bit in the instruction is used, as in the cond field of other branch and trap instructions, to determine if the instruction is testing for true or false. If the C bit is set, the branch or trap will occur only if the instruction's result is true; if the C bit is clear the branch or trap will occur only if the result is false.

Figure 15. Type Operation Formats

RR:



RIL:



Branch-next-type

Opcode: 74

Formats:

```
RR    if source1 type source2
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIL   if source1 type 38 bit immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+2
```

Source Data Type Traps: None.

Traps: None.

Description: Branches to the specified location if the type condition (as described on page 66) indicates that the branch should be taken. This instruction is used if it is predicted that the branch will not be taken.

Example: Branch to true-address if the data type of the word in r6 is *dtp-header-p or dtp-one-q-forward*:

```
branch-next-type @true-address,r6,#o44,H=0,C=1
```

Branch-take-type

Opcode: 64

Formats:

```
RR    if source1 type source2
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+1

RIL   if source1 type 38 bit immediate
         PC ← PC<31:12>|page-offset<11:0>
      else
         PC ← PC+2
```

Source Data Type Traps: None.

Traps: None.

Description: Description: Branches to the specified location if the type condition (as described on page 66) indicates that the branch should be taken. This instruction is used if it is predicted that the branch will be taken.

Example: Branch to true-address if the data type of the word in r6 is not one of the instruction types.

```
branch-take-type @true-address,r6,#o37777777400,H=1,C=0
```

Trap-type

Opcode: 70

Formats:

```
RR    if source1 type source2
         PC ← <11111>|trap-base-register|<00>|page-offset, TRAP
      else
         PC ← PC+1

RIL   if source1 type 38 bit immediate
         PC ← <11111>|trap-base-register|<00>|page-offset, TRAP
      else
         PC ← PC+2
```

Source Data Type Traps: None.

Traps: Traps if the type results in True.

Description: Traps if the type condition (as described on page 66) indicates that the trap should be taken. If the trap is taken, the incremented PC is pushed onto the return stack and the WBP is incrmented. The new PC value has the most significant five bits equal to ones (VMA=PMA space), bits 26:14 are taken from the trap base register, bits 13:12 are zeros, and the low twelve bits are taken from the instruction page offset field. If the trap is not taken, execute the next instruction normally.

Example: Trap to not-number-trap if the data type of the word in r6 is not one of the number types. Note that not-number-trap must be in VMA=PMA space and have bits 26:14 equal to the value in the trap base register and bits 13:12 equal to zero.

```
trap-type @not-number-trap,r6,#o177400,H=0,C=0
```

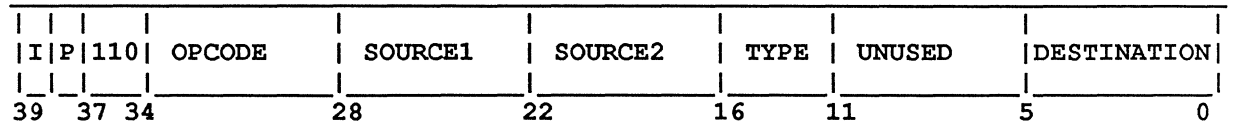
1.5.10 Load Operations

The load operations are as follows: *load-data, load-cdr, load-car-cdr, load-cdr-finish, load-array, load-structure, load-bind, load-header, load-scavenge, load-gc-copy, load-raw, load-data-iv, load-ephemeralp, load-oldspacep.* Available in all three formats, RR, RIS and RIL, load operations load data from memory into the register specified by the destination field in the instruction. The address of memory is calculated by adding the contents of the source1 register with the second operand, either the contents of source2 (RR format) or the immediate field (RIS and RIL formats). See figure 16 on page 71. The type check field specifies what data types to trap on for source1. In RR format only, source2 is always checked for *dtp-fixnum*. Typically in the RIL format the type field will select ≠fixnum or none, and in the RR and RIS format it will vary widely with each load instruction. All load operations load the cdr-reg and the type-reg. Typically the cdr-reg is loaded with bits 39:38 and the type-reg is loaded with bits 37:32, of the data being loaded. The exception to this is for instructions *load-cdr, load-car-cdr, load-header,* and *load-structure.* These four instructions will load the type-reg and cdr-reg as above except when the source1 data type is *dtp-nil.* In this case the cdr-reg is loaded with cdr-nil (a value of 1), and the type-reg is loaded with *dtp-nil* (a value of #o24). As a side effect of these four instructions, the MAR is loaded with the calculated data type and address, except when the data type of source1 is *dtp-nil* in which case MAR is loaded with NIL. If the destination of any load instruction is the Array Header Register, the Array Length Register is loaded as a side effect. The Array Length register is loaded with the second word read if it is a long prefix array, otherwise the array length register is loaded with the length field of the array header that was read. (See Table 8 on page 28.)

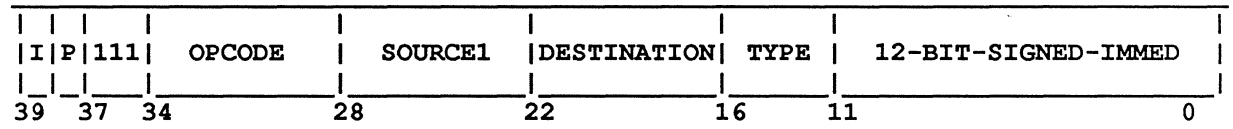
See coprocessor operations for the *load-coproc* instruction which has a special format.

Figure 16. Load Operation Formats

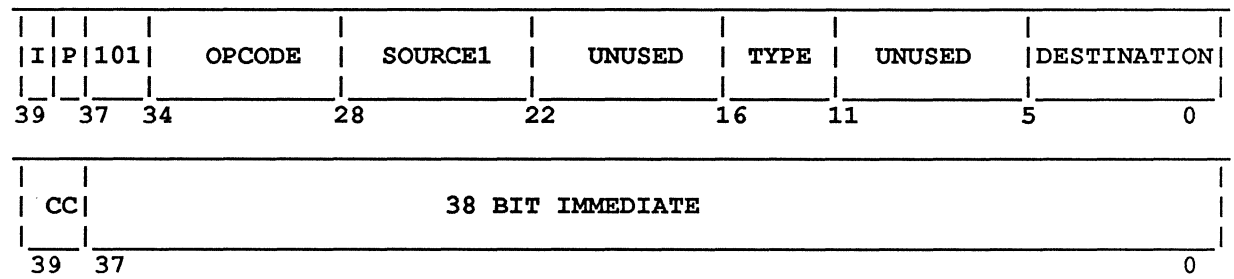
RR:



RIS:



RIL:



load-data

Opcode: 12

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This is the typical load instruction. This reads a word from memory located at the specified address and stores the data read in the register specified by the destination field. If the data type of the word read is one of: *null, header-p, header-i, gc-forward monitor-forward, external-value-cell-pointer, one-q-forward, header-forward, element-forward, external-value-cell-pointer, one-q-forward, header-forward, element-forward,* or a pointer to oldspace, a trap occurs as described above. This is similar to an I-Machine read with the data-read cycle type.

load-data-iv

Opcode: 13

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This reads a word from memory located at the specified address and stores the data read in the register specified by the destination field. This is identical to *load-data*, but is used to present a separate trap entry address (particularly for arg2 not being a fixnum, a speedup hack for currently unimplemented feature dealing with common lisp object oriented standard with-slots construct). If the data type of the word read is one of: *null, header-p, header-i, gc-forward, monitor-forward, external-value-cell-pointer, one-q-forward, header-forward, element-forward, external-value-cell-pointer, one-q-forward, header-forward, element-forward,* or pointer to oldspace, a trap occurs as described above.

load-car-cdr

Opcode: 14

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This is used for implementing the lisp car function. This is also used when a lisp cdr function follows a lisp car function (see the example below). The cdr-reg gets loaded as if it were a *load-cdr* operation,and the destination register gets loaded as if it were a *load-data* operation. With the exception that, if the data type of the source1 is *dtp-nil*, then nil is loaded in the MAR and the destination register. See the *load-cdr-finish* on page 76. As a side effect of this instruction executing, the MAR gets loaded with the calculated address.

Example:

```
(dolist (foo bar) ... )
        move             temp ← bar
   Loop:
        load-car-cdr     foo  ← [temp]    ≠list-loc-nil
        load-cdr-finish  temp ← [mar,#1]  ≠list-loc-nil
        ...
        branch-take-eq   not-endp,temp,nil,@loop
```

load-cdr-finish

Opcode: 15

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: Trap if cdr-reg is illegal value.

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This is used as the second part of implementing a Lisp cdr operation. This is similar to the *load-data* operation except the value in the cdr-reg controls what the result of the load operation is. If the cdr-reg value is cdr-nil, nil is loaded into the destination, if the cdr-reg value is cdr-next, the calculated memory address that is referencing memory is loaded into the destination register, if the cdr-reg value is cdr-normal, the value read from memory is loaded into the destination register. If the cdr-reg value is illegal, then a trap occurs. If the cdr-reg has any value other than cdr-normal, none of the memory-traps (error, monitor, indirect, transport) will actually be accepted. If the data-type of source1 in RR and RIS, or the immediate in RIL is *dtp-locative*, it is used as the address of the memory operation and the cdr-reg is ignored. If the data type of the source is *dtp-nil*, the destination is set to NIL and no memory reference is performed.

Example:

```
(setq bar (cdr foo))
  load-cdr          bar ← [foo]     ≠list-loc-nil ;RIS with
                                                  ;immed of 0

  load-cdr-finish   bar ← [mar,#1]  ≠list-loc-nil ;RIS with
                                                  ;immed of 1
```

load-array

Opcode: 16

Formats:

```
RR      destination ← [source1 + source2 + 1]
RIS     destination ← [source1 + 12-bit-signed-immediate + 1]
RIL     destination ← [38-bit-immediate + source1 + 1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: This instruction may take three array traps. In order of priority they are: a long prefix trap, an out of range trap, or a packed array trap. These take precedence over the Memory Error, Monitor, Indirect and Transport traps, as they effect the address calculated for the memory reference. However, the source data type traps takes precedence over these array traps.

Long Prefix -- Trap if the array-header register has bit 23 set and bit 12 clear. Bit 23 indicates that the array has a long prefix. Bit 12, the long prefix trap inhibit, tells the hardware to ignore the long prefix trap. This bit indicates that the array-header came from an array descriptor. The software that sets up array descriptors will set this bit to inhibit the long prefix trap.

Out of Range -- Trap if the offset(source2 in RR format, 12-bit-immediate in RIS format and 38-bit-immediate in the RIL format) is out of range, by comparing with unsigned≥ to the array length register. This is an error trap.

Packed -- Trap if the array is a byte packed array (i.e., bits 29:27 ≠0).

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

load-array

Opcode: 16

Formats:

```
RR      destination ← [source1 + source2 + 1]
RIS     destination ← [source1 + 12-bit-signed-immediate + 1]
RIL     destination ← [38-bit-immediate + source1 + 1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: This instruction may take three array traps. In order of priority they are: a long prefix trap, an out of range trap, or a packed array trap. These take precedence over the Memory Error, Monitor, Indirect and Transport traps, as they effect the address calculated for the memory reference. However, the source data type traps takes precedence over these array traps.

Long Prefix -- Trap if the array-header register has bit 23 set and bit 12 clear. Bit 23 indicates that the array has a long prefix. Bit 12, the long prefix trap inhibit, tells the hardware to ignore the long prefix trap. This bit indicates that the array-header came from an array descriptor. The software that sets up array descriptors will set this bit to inhibit the long prefix trap.

Out of Range -- Trap if the offset(source2 in RR format, 12-bit-immediate in RIS format and 38-bit-immediate in the RIL format) is out of range, by comparing with unsigned≥ to the array length register. This is an error trap.

Packed -- Trap if the array is a byte-packed array (AHR 29:27 ≠ 0) or the array type is character (AHR 31:30 = 1) or boolean (AHR 31:30 = 2).

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This is similar to the load-data instruction with the following exceptions. The instruction enables the three traps mentioned above. It adds one to the address calculated by adding source1 and source2. If the array is element type character, the resulting data type is forced to dtp-character. If the array is element type boolean, the resulting data is forced to either T or NIL based on the bit of the trap result register selected by the byte-rotate register. (Note that since all boolean arrays are byte packed as 32 bits to a word it must first take a packed array trap.) As a side effect, this instruction loads the byte-rotate register with a value based on the byte packing encoding in the array header register. See section 1.9 on 131 for more details on array references.

Example:

```
(setq result (aref foo bar))
        load-header     array-header ← [foo]     ≠array-string
        load-array      result ← [mar,bar]       ≠array-string
```

Description: This is similar to the load-data instruction with the following exceptions. The instruction enables the three traps mentioned above. It adds one to the address calculated by adding source1 and source2. If the array is element type character, the resulting data type is forced to dtp-character. If the array is element type boolean, the resulting data is forced to either T or NIL based on the bottom bit of the trap result register. (Note that since all boolean arrays are byte packed as 32 bits to a word it must first take a packed array trap.) As a side effect, this instruction loads the byte-rotate register with a value based on the byte packing encoding in the array header register. See section 1.10 on 134 for more details on array references.

Example:

```
(setq result (aref foo bar))
        load-header     array-header ← [foo]     ≠array-string
        load-array      result ← [mar,bar]       ≠array-string
```

load-cdr

Opcode: 10

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Indirect trap: If the data type of the word read is one of: *dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This is used as the first part of implementing a lisp cdr operation. Reads a word from memory located at the specified address and stores the data read in the register specified by the destination field, unless the data type of source1 is *dtp-nil*; in this case the destination register is loaded with NIL. If the data type of the word read is one of: header-p, header-i, gc-forward, header-forward, or element-forward, a trap occurs as described above. This typically is used to just get the cdr-code bits, which are loaded into the cdr-reg. This will typically be followed by *load-cdr-finish*. This is similar to the I-Machine's cdr-read cycle type. see also the *load-cdr-finish* operation on page 76. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR gets loaded with NIL.

Example:

```
(setq bar (cdr foo))
  load-cdr          bar ← [foo]      ≠list-loc-nil ;RIS with
                                                   ;immed of 0

  load-cdr-finish   bar ← [mar,#1]   ≠list-loc-nil ;RIS with
                                                   ;immed of 1
```

load-cdr

Opcode: 04

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Indirect trap: If the data type of the word read is one of: *dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This is used as the first part of implementing a lisp cdr operation. Reads a word from memory located at the specified address and stores the data read in the register specified by the destination field, unless the data type of source1 is *dtp-nil*; in this case the destination register is loaded with NIL. If the data type of the word read is one of: header-p, header-i, gc-forward, header-forward, or element-forward, a trap occurs as described above. This typically is used to just get the cdr-code bits, which are loaded into the cdr-reg. This will typically be followed by *load-cdr-finish.* This is similar to the I-Machine's cdr-read cycle type. see also the *load-cdr-finish* operation on page 78. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR gets loaded with NIL.

Example:

```
(setq bar (cdr foo))
  load-cdr          bar ← [foo]      ≠list-loc-nil ;RIS with
                                                   ;immed of 0

  load-cdr-finish   bar ← [mar,#1]   ≠list-loc-nil ;RIS with
                                                   ;immed of 1
```

load-structure

Opcode: 05

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of *dtp-gc-forward* - trap to the error trap handler.

Indirect trap: If the data type of the word read is one of *dtp-header-forward*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: Similar to the I-Machine's struc-offset macro instruction. Loads a structure header. This type of reference follows header-forwarding pointers, by taking indirect traps, as necessary and traps out if a *dtp-gc-forward* is encountered. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR and the destination register get loaded with NIL.

load-structure

Opcode: 11

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of *dtp-gc-forward* - trap to the error trap handler.

Indirect trap: If the data type of the word read is one of *dtp-header-forward*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: Similar to the I-Machine's struc-offset macro instruction. Loads a structure header. This type of reference follows header-forwarding pointers, by taking indirect traps, as necessary and traps out if a *dtp-gc-forward* is encountered. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR and the destination register get loaded with NIL.

load-header

Opcode: 06

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: All data types except *dtp-header-p, dtp-header-forward, and dtp-header-i*, trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Indirect trap: If the data type of the word read is one of: *dtp-header-forward*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This reads the word at the requested address. If the word read is of *dtp-header-p* or *dtp-header-i* it is stored in the destination register. If the word read is of data type *dtp-header-forward*, the invisible pointer is followed in the trap routine. If any other data type is encountered an error trap occurs. If the data type that is finally read (without a trap) contains an address in oldspace, a transport trap will occur. This is similar to the I-Machine's header-read cycle type. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR and the destination register get loaded with NIL.

load-header

Opcode: 11

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: All data types except *dtp-header-p, dtp-header-forward, and dtp-header-i*, trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Indirect trap: If the data type of the word read is one of: *dtp-header-forward*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This reads the word at the requested address. If the word read is of *dtp-header-p* or *dtp-header-i* it is stored in the destination register. If the word read is of data type *dtp-header-forward*, the invisible pointer is followed in the trap routine. If any other data type is encountered an error trap occurs. If the data type that is finally read (without a trap) contains an address in oldspace, a transport trap will occur. This is similar to the I-Machine's header-read cycle type. As a side effect of this instruction executing, the MAR gets loaded with the calculated address. If the data type of the source is *dtp-nil*, then the MAR and the destination register get loaded with NIL.

load-bind

Opcode: 10

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This reads the word at the requested address. This is similar to the I-Machine's bind-read cycle type. This is also similar to the *load-data* instruction except in the handling of *dtp-null* and *dtp-external-value-cell-pointer*.

load-bind

Opcode: 07

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: This reads the word at the requested address. This is similar to the I-Machine's bind-read cycle type. This is also similar to the *load-data* instruction except in the handling of *dtp-null* and *dtp-external-value-cell-pointer*.

load-scavenge

Opcode: 05

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Description: Used by the garbage collector. If the data read is of data type *dtp-gc-forward*, an error trap occurs. If the data read is a pointer and points to oldspace, the transport trap occurs. This is similar to the I-Machine's scavenge-read cycle type.

load-scavenge

Opcode: 07

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-gc-forward* - trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Description: Used by the garbage collector. If the data read is of data type *dtp-gc-forward*, an error trap occurs. If the data read is a pointer and points to oldspace, the transport trap occurs. This is similar to the I-Machine's scavenge-read cycle type.

load-gc-copy

Opcode: 06

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-gc-forward* - trap to the error trap handler.

Description: Used by the garbage collector. If the data read is of data type *dtp-gc-forward*, an error trap occurs. This is similar to *load-scavenge*. This is similar to the I-Machine's gc-copy-read cycle type.

load-gc-copy

Opcode: 04

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Memory Error Trap: If the data type of the word read is one of: *dtp-gc-forward* - trap to the error trap handler.

Description: Used by the garbage collector. If the data read is of data type *dtp-gc-forward*, an error trap occurs. This is similar to *load-scavenge*. This is similar to the I-Machine's gc-copy-read cycle type.

load-raw

Opcode: 02

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: none

Description: Similar to the I-Machine's raw-read cycle type. Does no indirection, transport traps, or error traps.

load-oldspacep

Opcode: 01

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: none

Description: Similar to a *load-raw*. No indirection or transport or error traps. If the data referenced is a pointer and points to oldspace load into the destination register the data read, otherwise load NIL.

load-ephemeralp

Opcode: 00

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: none

Description: Similar to a *load-raw*. No indirection or transport or error traps. If the data referenced is a pointer and points to ephemeral space, load into the destination register the data read, otherwise load NIL.

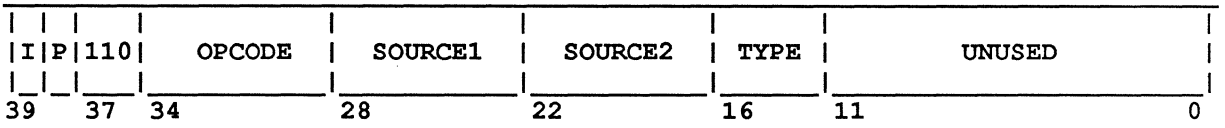
1.5.11 Store Operations

The Store Operations are: *store-data, store-data-iv, store-array, store-bind, store-cdr-nil, store-cdr-next, store-cdr-normal, store-cdr-reg, store-type-reg, store-rplacd,* or *store-38-bits*. Stores data in memory at the address calculated by adding source1 with the MAR for RR format or the immediate field for RIS and RIL formats. See figure 17 on page 89. The RR format will typically only be used by the *store-array* instruction. Source2 is the data to be stored at the calculated address. The data type of source1 is tested according to the type field. When using the RR format to store, the contents of source1 are always added to the MAR to be used as the address of memory in which to store the contents of Source2. *Store-data, store-data-iv, store-array, store-rplacd, store-bind,* and *store-38-bits* all preserve the cdr-code of the location into which they write.

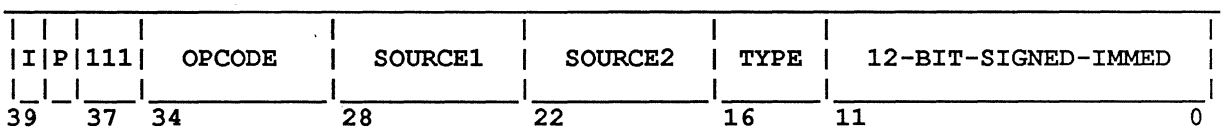
See coprocessor operations for the *store-coproc* instruction which has a special format.

Figure 17. Store Operation Formats

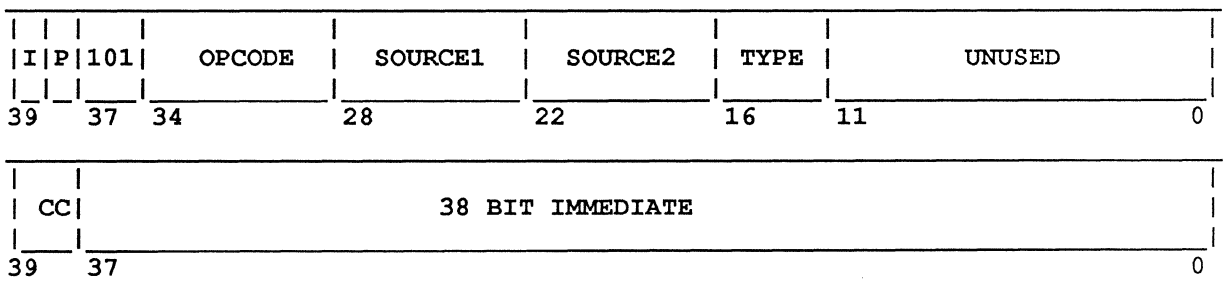
RR:



RIS:



RIL:



store-data

Opcode: 32

Formats:

```
RR      [source1 + mar] ← source2
RIS     [source1 + 12-bit-signed-immediate] ← source2
RIL     [38-bit-immediate + source1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the MAR in RR format,or the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Similar to the I-Machine's data-write.

store-data-iv

Opcode: 33

Formats:

```
RR      [Source1 + mar] ← source2
RIS     [Source1 + 12-bit-signed-immediate] ← source2
RIL     [38-bit-immediate + Source1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the MAR in RR format,or the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Similar to store-data and is used in flavor accessing for a unique trap address should the type of Source1 not be fixnum.

store-rplacd

Opcode: 35

Formats:

```
RR      [source1 + mar] ← source2
RIS     [source1 + 12-bit-signed-immediate] ← source2
RIL     [38-bit-immediate + source1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: Traps if the cdr-reg is not cdr-normal.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the MAR in RR format,or the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Used to support efficient implementation of rplacd. If the data-type of source1 in RR and RIS, or the immediate in RIL is *dtp-locative*, it is used as the address of the memory operation.

Example:

```
(rplacd foo value)
        load-cdr        bar ← [foo]             ≠list-loc
        store-rplacd    [mar,#1] ← value        ≠list-loc
```

store-rplacd

Opcode: 35

Formats:

```
RIS     [source1 + 12-bit-signed-immediate] ← source2
RIL     [38-bit-immediate + source1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: Traps if the cdr-reg is not cdr-normal.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Used to support efficient implementation of rplacd. If the data-type of source1 in RIS is *dtp-locative*, it is used as the address of the memory operation.

Example:

```
(rplacd foo value)
        load-cdr        bar ← [foo]             ≠list-loc
        store-rplacd    [mar,#1] ← value        ≠list-loc
```

store-array

Opcode: 36

Formats:

```
RR      [source1 + mar + 1] ← source2
RIS     [source1 + 12-bit-signed-immediate + 1] ← source2
RIL     [38-bit-immediate + source1 + 1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: This instruction may take four array traps. In order of priority they are: array element trap, a long prefix trap, an out of range trap, or a packed array trap. These take precedence over the Memory Error, Monitor, Indirect and Transport traps, as they effect the address calculated for the memory reference. However, the source data type traps takes precedence over these array traps.

Array-Element -- Traps in three array element type cases:

1. when the array element type is boolean,
2. when the array element type is character and source2 isn't character or has bits set above the array element size (i.e., bit 20 is set for a 16-bit-string array), and
3. when the array element type is fixnum and source2 isn't fixnum.

Long Prefix -- Trap if the array-header register has bit 23 set and bit 12 clear. Bit 23 indicates that the array has a long prefix. Bit 12, the long prefix trap inhibit, tells the hardware to ignore the long prefix trap. This bit indicates that the array-header came from an array descriptor. The software that sets up array descriptors will set this bit to inhibit the long prefix trap.

Out of Range -- Trap if the offset (source2 in RR format, 12-bit-immediate in RIS format and 38-bit-immediate in the RIL format) is out of range, by comparing with unsigned ≥ to the array length register. This is an error trap.

Packed -- Trap if the array is a byte-packed array (AHR 29:27 ≠ 0) or the array type is character (AHR 31:30 = 1).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

store-array

Opcode: 36

Formats:

```
RR      [source1 + mar + 1] ← source2
RIS     [source1 + 12-bit-signed-immediate + 1] ← source2
RIL     [38-bit-immediate + source1 + 1] ← source2
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: This instruction may take four array traps. In order of priority they are: array element trap, a long prefix trap, an out of range trap, or a packed array trap. These take precedence over the Memory Error, Monitor, Indirect and Transport traps, as they effect the address calculated for the memory reference. However, the source data type traps takes precedence over these array traps.

Array-Element -- Traps in three array element type cases:

1. when the array element type is boolean,
2. when the array element type is character and source2 isn't character or has bits set above the array element size (i.e., bit 20 is set for a 16-bit-string array), and
3. when the array element type is fixnum and source2 isn't fixnum.

Long Prefix -- Trap if the array-header register has bit 23 set and bit 12 clear. Bit 23 indicates that the array has a long prefix. Bit 12, the long prefix trap inhibit, tells the hardware to ignore the long prefix trap. This bit indicates that the array-header came from an array descriptor. The software that sets up array descriptors will set this bit to inhibit the long prefix trap.

Out of Range -- Trap if the offset (source2 in RR format, 12-bit-immediate in RIS format and 38-bit-immediate in the RIL format) is out of range, by comparing with unsigned ≥ to the array length register. This is an error trap.

Packed -- Trap if the array is a byte packed array (i.e., bits 29:27 ≠0).

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: Reads the contents of the memory location specified by adding one to the sum of source1 and either the MAR in RR format, the 12-bit-immediate in RIS format or the 38-bit-immediate in RIL format. The instruction enables the four array traps mentioned above, in addition to the source1 data type trap which has priority over the array traps and the data type traps based on the word read from the specified location before the store which have lowest priority. If no error trap, stores into the memory at the specified address. As a side effect, this instruction loads the byte-rotate register with a value based on the byte packing encoding in the array header register. See section 1.10 on page 134 for more details on array references.

Example:

```
(aset element array subscript)
        load-header     array-header ← [array]          ≠array-string
        store-array     [mar,subscript] ← element       ≠array-string
```

store-bind

Opcode: 30

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the MAR in RR format,or the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Similar to the I-Machine's bind-write cycle type.

store-bind

Opcode: 27

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward* - trap to the error trap handler.

Monitor trap: If the data type of the word read is *dtp-monitor-forward*, take a monitor trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Indirect trap: If the data type of the word read is one of: *dtp-one-q-forward, dtp-header-forward or dtp-element-forwarding*, take an indirect trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority. The indirect trap routine will follow the forwarding pointer chain.

Description: Reads the contents of the memory location specified by the sum of the contents of source1 and the MAR in RR format,or the 12-bit-signed-immediate in the RIS format, or the 32-bit-immediate in the RIL format. Traps on the data type of the word read as specified above. If no error trap, stores into memory at the specified address. Similar to the I-Machine's bind-write cycle type.

store-cdr-nil

Opcode: 21

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top two bits of the 40-bit word (the cdr-code) set to cdr-nil.

store-cdr-next

Opcode: 20

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top two bits of the 40-bit word (the cdr-code) set to cdr-next.

store-cdr-normal

Opcode: 22

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top two bits of the 40-bit word (the cdr-code) set to cdr-normal.

store-cdr-3

Opcode: 23

Formats:

RR	[source1 + mar] ← source2
RIS	[source1 + 12-bit-signed-immediate] ← source2
RIL	[38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top two bits of the 40-bit word (the cdr-code) set to 3.

store-cdr-reg

Opcode: 24

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top two bits of the 40-bit word (the cdr-code) set to the value stored in the cdr-reg.

store-type-reg

Opcode: 25

Formats:

RR	[source1 + mar] ← source2
RIS	[source1 + 12-bit-signed-immediate] ← source2
RIL	[38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's raw-write cycle type. Stores source2 into memory at the calculated address with the top eight bits, the cdr-code and the type field, coming from the contents of the Cdr reg and the Type reg respectively.

store-38-bits

Opcode: 26

Formats:

RR [source1 + mar] ← source2
RIS [source1 + 12-bit-signed-immediate] ← source2
RIL [38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: This is similar to the I-Machine's %p-store-contents macro instruction. Stores source2 into memory at the address calculated. This performs no type checking on the memory data at the location being written, and does not change the cdr-code.

1.5.12 Coprocessor Operations

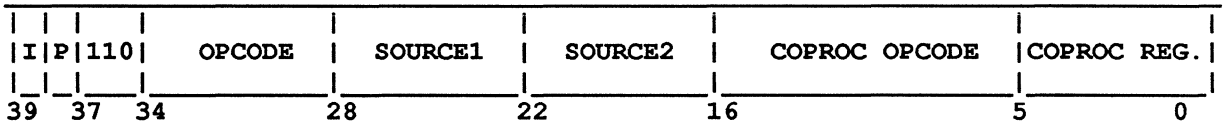
The coprocessor operations are: *read-coproc*, *write-coproc*, *load-coproc*, and *store-coproc* Three of the coprocessor instructions are available in all RR and RIL formats; they are: *write-coproc*, *load-coproc*, and *store-coproc*. No Data type check is performed. The *load-coproc* and *store-coproc* instructions use Source1 and Source2 or Immediate to calculate memory address like other memory instructions. They also follow the same trap rules as *load-data* and *store-data* instructions. The 11-bit Coproc Opcode Field specifies what operation the coprocessor (eg, floating point) hardware is to perform. The 6-bit Coproc Reg Field specifies a source or destination register in the coprocessor.

The *read-coproc* instruction is only available in the RIS format. Source1 and Source2 are not used. No data type check is performed. See figure 18 on page 105.

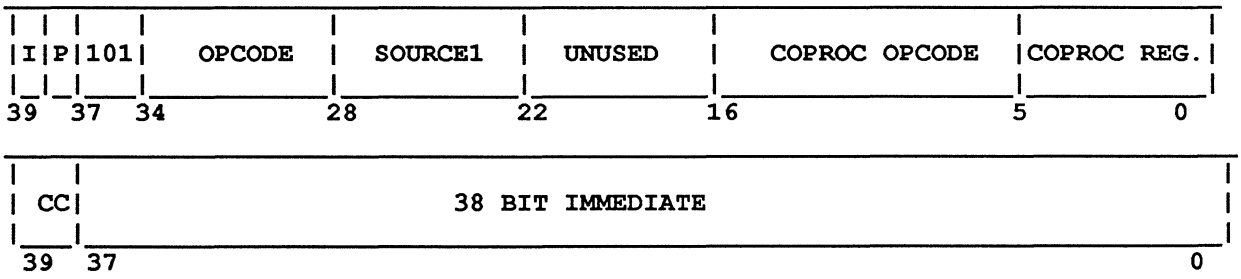
Figure 18. Coprocessor Operation Formats

write-coproc, load-coproc, store-coproc:

RR:

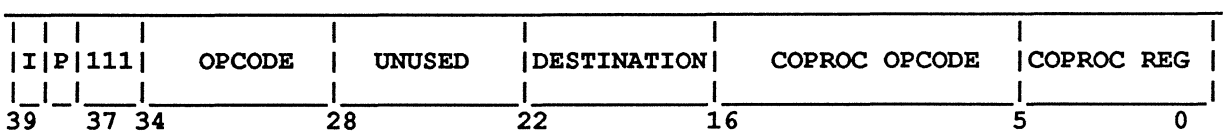


RIL:



read-coproc:

RIS:



1.5.12 Coprocessor Operations

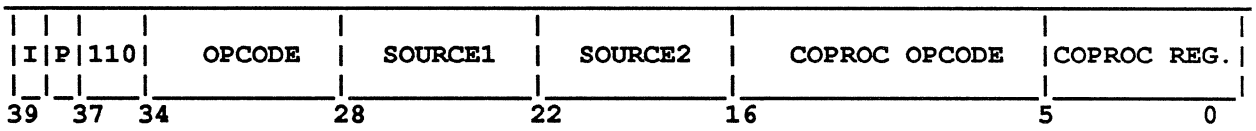
The coprocessor operations are: *read-coproc*, *write-coproc*, *load-coproc*, and *store-coproc* Three of the coprocessor instructions are available in all RR and RIL formats; they are: *write-coproc*, *load-coproc*, and *store-coproc*. No Data type check is performed. The *load-coproc* and *store-coproc* instructions use Source1 and Source2 or Immediate to calculate memory address like other memory instructions. They also follow the same trap rules as *load-data* and *store-data* instructions. The 11-bit Coproc Opcode Field specifies what operation the coprocessor (eg, floating point) hardware is to perform. The 6-bit Coproc Reg Field specifies a source or destination register in the coprocessor.

The *read-coproc* instruction is only available in the RR format. Source1 and Source2 are not used. No data type check is performed. See figure 18 on page 103.

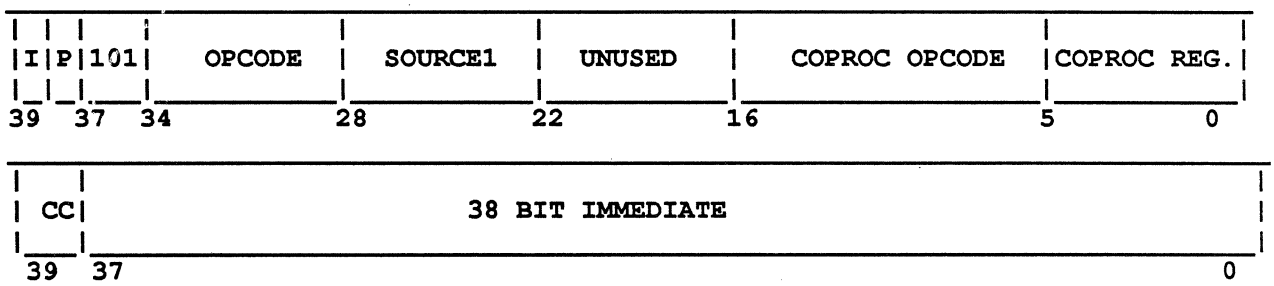
Figure 18. Coprocessor Operation Formats

write-coproc, load-coproc, store-coproc:

RR:

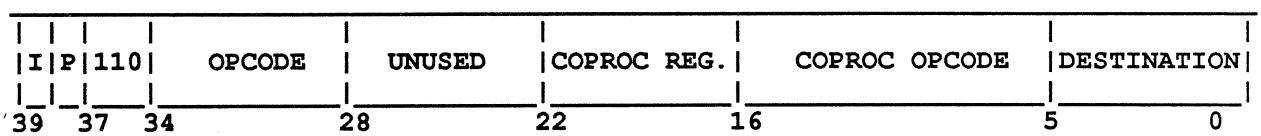


RIL:



read-Coproc:

RR:



Write-Coproc

Opcode: 57

Formats:

```
RR      coprocessor ←    source2|source1
RIL     coprocessor ← immediate|source1
```

Source Data Type Traps: None.

Coprocessor Trap: As determined by the coprocessor hardware.

Description: This instruction stores the data from the two sources into the coprocessor register specified by the Coproc Opcode and/or the Coproc Reg Field. The Coproc Opcode specifies what coprocessor operation to start or perform, and whether to use the coproc-reg as a source or destination. In a typical use, the sources load into the Coproc Reg. Another typical use starts a double word floating point operation using the Coproc Reg to point to one operand and the two processor registers (Source1 and Source2) as another operand which might be a double precision floating point number. Source2 or the immediate operand corresponds to the even word (MSW) of a double float sent to the coprocessor; Source1 is the odd word (LSW).

Load-Coproc

Opcode: 17

Formats:

```
RR      coprocessor ← [source1 + source2]
RIL     coprocessor ← [source1 + immediate]
```

Source Data Type Traps:

Coprocessor trap: As determined by the coprocessor hardware.

Memory Error Trap: If the data type of the word read is one of: *dtp-null, dtp-header-p, dtp-header-i or dtp-gc-forward* trap to the error trap handler.

Transport Trap: If the word read is a pointer to oldspace, and if transport traps are enabled for the page containing the word read, a transport trap will occur to evacuate the object.

Monitor Trap: If the data type of the word read is *dtp-monitor-forward*, take a transport trap as described above, otherwise take a monitor trap.

Indirect Trap: If the data type of the word read is one of: dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward, or dtp-element-forwarding - Take a transport trap as described above, otherwise take and indirect trap to follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This special load instruction reads two words from memory located at the specified address and stores the data read into the coprocessor register specified by the coproc-opcode and/or the coproc-reg field. The Coproc Opcode specifies what coprocessor operation to start or perform, and whether to use the Coproc Reg as a source or destination. The coprocessor may operate on 64 bit sources. In a typical use, the sources load into the Coproc Reg. Another typical use starts an operation using the Coproc Reg to define one coproc source and the just-loaded *memory data* as another source. Traps on data type of word read as specified above. Although this instruction transfers a double word to the coprocessor, only the addressed word is checked for traps. Except for this trap checking, the low order address bit is ignored.

Store-Coproc

Opcode: 37

Formats:

```
RR      [source1 + source2]    ← coprocessor
RIL     [source1 + immediate]  ← coprocessor
```

Source Data Type Traps:

Coprocessor trap: As determined by the coprocessor hardware.

Memory Error Trap: If the data type of the word read is one of: *dtp-header-p, dtp-header-i or dtp-gc-forward*, trap to the error trap handler.

Monitor Trap: If the data type of the word read is *dtp-monitor-forward*, take a transport trap as described above, otherwise take a monitor trap.

Indirect Trap: If the data type of the word read is one of: dtp-external-value-cell-pointer, dtp-one-q-forward, dtp-header-forward, or dtp-element-forwarding - Take a transport trap as described above, otherwise take and indirect trap to follow the forwarding pointer chain.

Bound location trap: If the data type of the word read is dtp-bound-location, take a bound location trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Logic variable trap: If the data type of the word read is dtp-logic-variable, take a logic variable trap, unless the conditions for a transport trap are present, in which case the transport trap takes priority.

Description: This special store instruction reads the contents of the memory location at the specified address and traps on the data type of the word read as specified above. If no error trap, it stores from the coprocessor into memory at the specified address.This instruction always stores a double word from the coprocessor into memory. Except for determining which word to use for error trap checking, this instruction ignores the low order address bit. The Coproc Opcode specifies what coprocessor operation to start or perform, what to write into memory, and whether to use the Coproc Reg as a source or another destination. Typically, this instruction writes the result of a previously started coprocessor operation into memory.

Store-Coproc

Opcode: 37

Formats:

```
RR      [source1 + source2]    ← coprocessor
RIL     [source1 + immediate]  ← coprocessor
```

Source Data Type Traps:

Coprocessor trap: As determined by the coprocessor hardware.

Description: This special store instruction stores a double word result from the coprocessor into memory. This instruction ignores the low order address bit. The Coproc Opcode specifies what coprocessor operation to start or perform, what to write into memory, and whether to use the Coproc Reg as a source or another destination. Typically, this instruction writes the result of a previously started coprocessor operation into memory. Store-coproc stores cdr-next as the cdr code of the even word and cdr-nil as the cdr-code of the odd word that is stored into memory.

Read-Coproc

Opcode: 77

Formats:

```
RIS     destination ← coprocessor
```

Source Data Type Traps: None.

Coprocessor Trap: As determined by the coprocessor hardware.

Description: The Coproc Opcode specifies what coprocessor operation to start or perform,what to write into the destination, and whether to use the Coproc Reg as a source or another destination. Typically, this instruction reads the result of a previously started coprocessor operation.

Read-Coproc

Opcode: 77

Formats:

```
RR      destination ← coprocessor
```

Source Data Type Traps: None.

Coprocessor Trap: As determined by the coprocessor hardware.

Description: The Coproc Opcode specifies what coprocessor operation to start or perform,what to write into the destination, and whether to use the Coproc Reg as a source or another destination. Typically, this instruction reads the result of a previously started coprocessor operation.

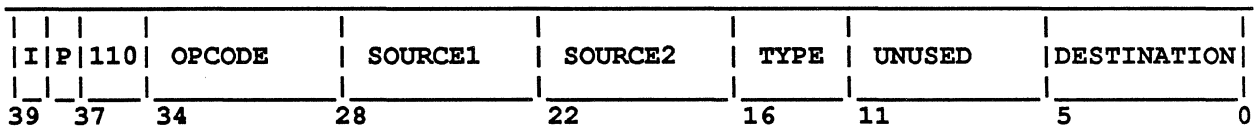
1.5.13 MC Register Operations

The MC Register operations are: *read-mc-reg* and *write-mc-reg*. These instructions communicate to the memory control logic section and internal MC registers such as the WBC. They use the sources similar to load and store instructions in that they calculate an address to put out on the address lines. However, instead of addressing the S-Cache, this address specifies control information to the memory control logic. For a decoding of that control information see the MC register descriptions on page 9. The data passes from or to the processor over the SBUS as it does in load or store instructions. The formats for read and write follow those for load and store instructions, respectively. See figure 19 on page 109.

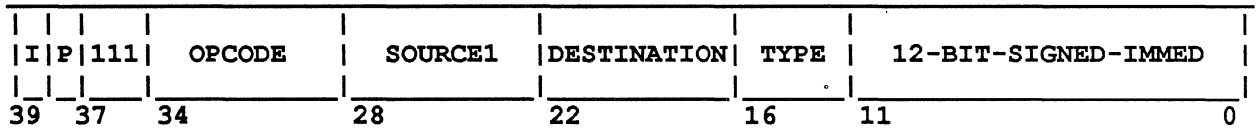
Figure 19. MC Reg Operation Formats

read-mc-reg:

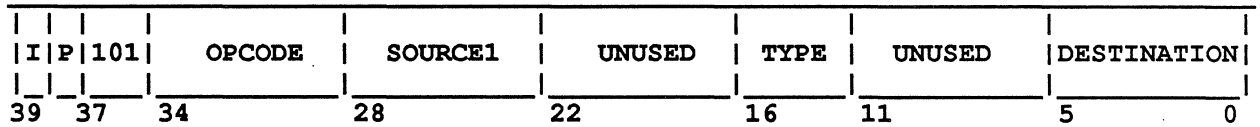
RR:

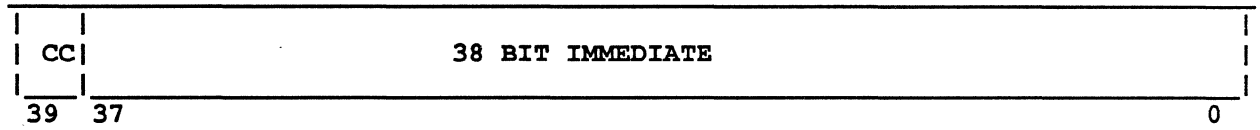


RIS:



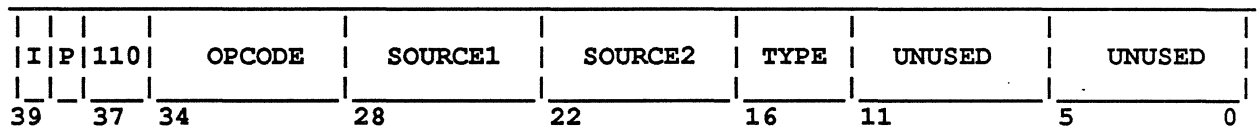
RIL:



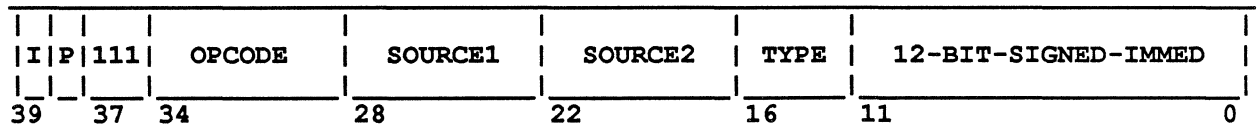


write-mc-reg:

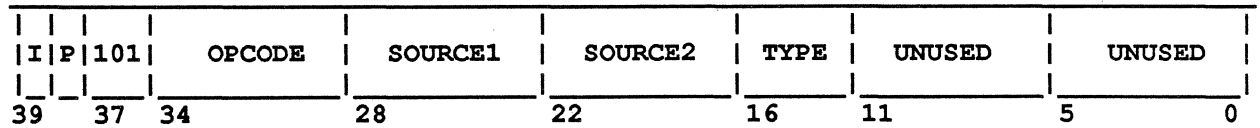
RR:

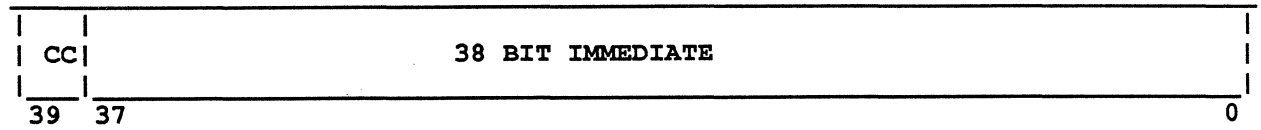


RIS:



RIL:





read-mc-reg

Opcode: 67

Formats:

```
RR      destination ← [source1 + source2]
RIS     destination ← [source1 + 12-bit-signed-immediate]
RIL     destination ← [38-bit-immediate + source1]
```

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified (and in the RR format tests source2 to be *dtp-fixnum*).

Traps: none

Description: Reads the contents of the memory control register specified by the computed address.

write-mc-reg

Opcode: 47

Formats:

RR	[source1 + mar] ← source2
RIS	[source1 + 12-bit-signed-immediate] ← source2
RIL	[38-bit-immediate + source1] ← source2

Source Data Type Traps: Specified by the type field, checks source1 to be of the data type specified.

Traps: None.

Description: Stores into a memory control register.

1.6 STACK GROUPS

A stack group provides a means for describing each process. See table 11 on page 112 for a list of the contents of a stack group.

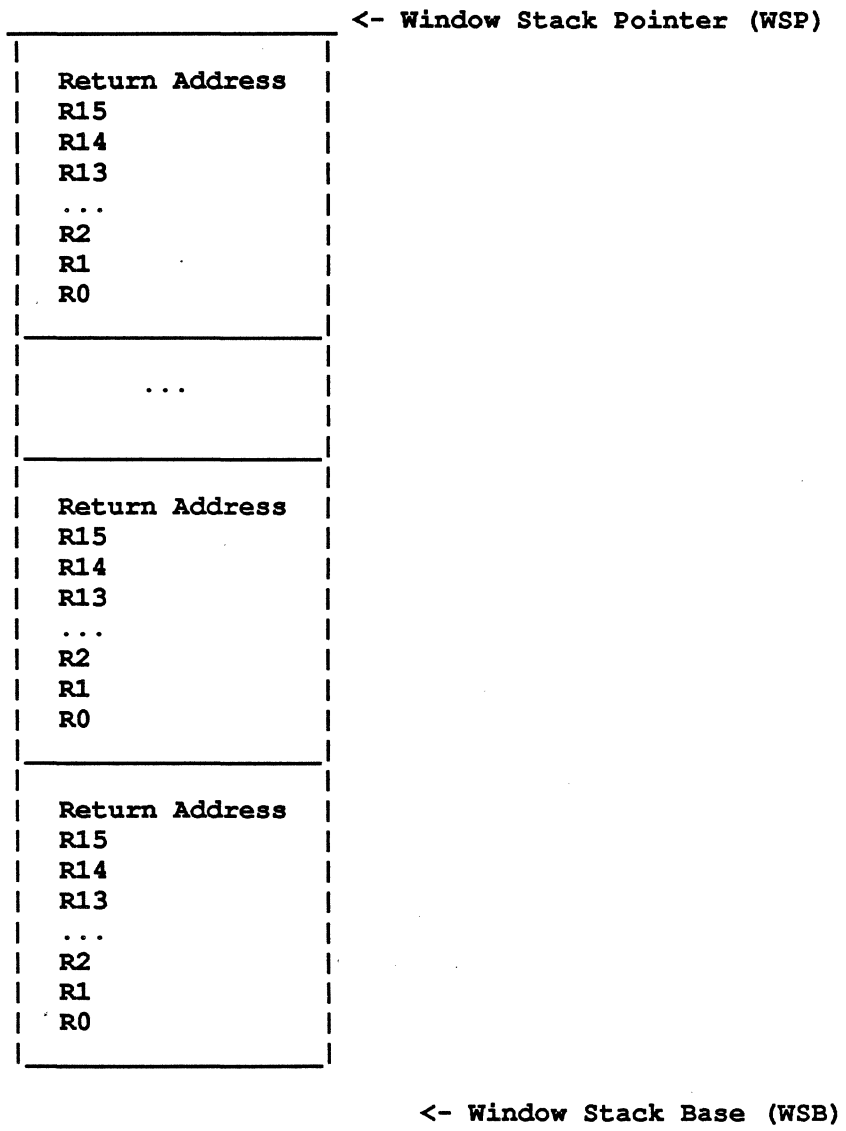
Table 11. Stack Group Registers

Stack Group State	
Structure	Abbr.
Window Stack	(WS)
Window Stack Pointer	(WSP)
Window Stack Base	(WSB)
Window Stack Limit	(WSL)
Data Stack	(DS)
Data Stack Pointer	(DSP)
Data Stack Base	(DSB)
Data Stack Limit	(DSL)
Binding Stack	(BS)
Binding Stack Pointer	(BSP)
Binding Stack Base	(BSB)
Binding Stack Limit	(BSL)
Program Counter	(PC)
Catch Block Pointer	(CBP)
List Block Pointer	(LBP)
List Block Base	(LBB)
List Block Length	(LBL)
Structure Block Pointer	(SBP)
Structure Block Base	(SBB)
Structure Block Length	(SBL)
Byte-Rotate Register	(BR)
Array Header Register	(AHR)
Array Length Register	(ALR)
Trap Result Register	(TR)
Number of Args Register	(N-ARGS)
Status/Control Register	(CR)
Memory Address Register	(MAR)
Coprocesor state	

1.6.1 Window Stack

The combination of the Window Stack and the Data Stack provides the functional equivalent of the I-Machine or L-Machine Control Stack. In L-Machine language, what is referred to as a frame (or Control Stack Frame) equates to a Sunstone Window and possibly includes a block on the Data Stack. The window stack (see figure 20 on page 113) has 17 locations for each window (frame); 16 register values, and the return address of the calling function. Most functions will require only a window, and will not have need of a block on the data stack. Each frame has a window associated with it, and some frames will have a block on the data stack. The Window Stack Pointer (WSP) points to the address of the last used location. The Window Stack Base (WSB) points to the first invalid location of the WS. The Window Stack Limit (WSL) points to the maximum address that has been allocated for the stack.

Figure 20. Window Stack



1.6.2 Data Stack

The Data Stack provides an allocation area for temporary data whose lifetime is associated with a function's lifetime. This allows less expensive allocation and deallocation than the general mechanism. Software implements this in the same manner as the L-Machine and I-Machine. In addition, when there is not enough room in the window extra needed space can be used on the data stack. The cases where a block of space will be needed on the data stack are:

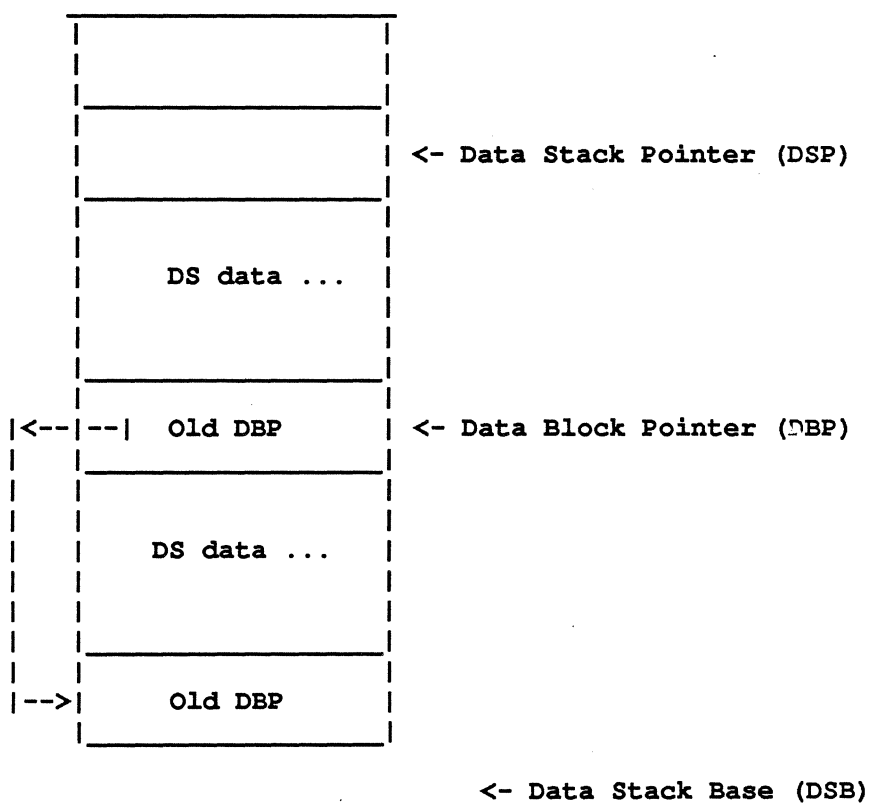
- too many args and locals (16 is the maximum in a window)

- any local that requires a locative to it
- catch or unwind-protect blocks
- any function that has an &rest argument

The Data Block Pointer (DBP) points to the current block on the data stack. The Data Stack Pointer (DSP) points the the first free location on the DS. See figure 21 on page 114.

A block on the data stack will use a one word header, the Old Data block Pointer. The compiler knows which functions will need to use the data stack and will generate entry and exit code to set up and tear down the data stack block for the function. Throw code is responsible for knowing when to pop the data stack. Through dynamic analysis, we expect about 10% of the functions will require use of the Data Stack. The DBP is saved on the data stack so that it can be restored on return.

Figure 21. Data Stack



How do each of the cases listed above use the data stack?

- *too many args* - An apply type call will be used whenever a function is called with more than 14 arguments. See page 117

- *any local that requires a locative to it* - for an example of this see the Lisp remq function. In this case, the compiler creates a location on the Data Stack for the argument.
- *catch and unwind protect blocks* - Similar to the I-Machine. A catch-throw catch block on the DS is 4 words, and the unwind protect catch block is 3 words. All of the catch blocks in the stack are linked together. The Catch Block Pointer (CBP) points to the most recent catch block.

```
CATCH-BLOCK:
catch-block-tag                     any object reference
catch-block-pc                      catch exit address
catch-block-binding-stack-pointer   has cdr-code of 0,
                                          means catch-block
catch-block-previous                 previous catch block


UNWIND-PROTECT:

catch-block-pc                      cleanup handler
catch-block-binding-stack-pointer   has cdr-code of 1,
                                         means unwind-protect
catch-block-previous                previous catch block
```

- *&rest in lambda list* - See section 1.7 on page 116. The suggested implementation preserves the current Symbolics incompatibility with Common Lisp. It might be reasonable to consider consing the &REST list instead of maintaining it on the stack. On this machine the only extra cost for that should be in garbage collection and consing.

1.6.3 Binding Stack

The binding stack looks like the I-Machine and L-Machine Binding Stack. There are two words per entry. The first word contains a locative to the memory cell that has been bound, and the second word is the contents of the bound cell. The cdr-code of the saved memory cell locative links the entries on the Binding Stack together. For each window with entries on the Binding Stack, the compiler will initially set the cdr-code of the memory cell locative to 0. All subsequent binds will set the cdr-code to 1. The block of code that performs the bind and the unbind should have the inhibit preempts bit in the instructions set.

Binding first performs a *load-bind* to get the contents pointed at by the locative. Then it does a *store-cdr-next* of the locative onto the Binding Stack, followed by a *store-cdr-next* of the value. Finally it increments the BSP by 2.

```
(bind pointer value) ; assuming pointer is a locative
        *load-bind       old-value ← [pointer]    ; disable preempt
        *store-cdr-next   [bsp,#2] ← old-value    ; disable preempt
        *store-cdr-next  [bsp,#1] ← pointer       ; disable preempt
        *add             bsp ← bsp,#2             ; disable preempt
         store-bind      [pointer] ← value
```

Unbinding follows a similar process. It first performs a *load-bind* of the bound value in order to read the old value. Next it performs a *load-bind* locative to get a pointer to the value cell. After getting the pointer, it performs a *store-bind* to store the old value at the location pointed to; then decrements the BSP by 2.

```
(unbind)
        *load-bind      old-value ← [bsp]          ; disable preempt
        *load-bind      pointer ← [bsp,#-1]        ; disable preempt
        *store-bind     [pointer] ← old-value      ; disable preempt
         sub            bsp ← bsp,#2
```

1.6.4 Stack Group Switching

The exact list of actions taken on a stack group switch, and their order, is very hardware dependent. Here are some of the actions, in order of execution:

- save registers
- swap binding stack - in descending order
- restore binding stack - in ascending order
- restore registers

1.7 FUNCTION CALLING

This section describes the Sunstone Function Calling Process and includes: Calling, Returning, Entry, Generic Functions, Message Passing, Lexical Closures.

1.7.1 Calling

There are basically four cases:

1. Calling a constant function with the expected number of arguments. A given function can have more than one expected number of arguments if there are &optional arguments.
2. APPLYing a constant function with the expected number of spread arguments.
3. FUNCALL other than (1).
4. APPLY other than (2).

On Sunstone all four are done with the *call* instruction, with help from trap handlers in some cases. The N-Args register is 5 bits wide, and the high bit(called Applybit) is 0 for normal calls and 1 for APPLY. The low bits are the number of spread arguments. If there is an apply argument it is always passed in A15, not in the next register after the spread arguments. Note that coding the N-Args this way for APPLY causes just the right effects in number-of-arguments checking, provided that the wrong number of arguments handler looks at the situation and does a pull-apply-args or push-apply-args when necessary. Tail recursion removal is done with a direct-branch instruction.

If the number of arguments passed for a compiled function is less than 15, the arguments are simply placed in the callee's registers in sequence, by storing them in A0 through A13 in the build window. If there are more than 14 arguments, the compiler pushes the extra arguments onto the data stack. The *call* instruction loads the number of arguments register and branches to the called function. In the general case the branch will be to the function's entry code, which will verify that the correct number of arguments is passed; in the case where it is known that the caller and callee agree as to the number and form of the arguments to be passed the *call* instruction will branch directly to the body of the function. We expect that functions that do not require a block on the data stack will be dynamically the most frequent.

Callers with more than 14 arguments will turn them into an APPLY. The caller makes a stack list of the arguments after the first 14, puts that stack list in A15, and calls with N-Args = applybit + 14. Upon return the caller has to pop the stack list off of the data stack. A callee with more than 14 spread arguments is altered by the compiler to have 14 spread arguments plus an apply argument, and includes explicit car'ing and cdr'ing of the apply argument to retrieve the other arguments. The reason only 14 spread arguments are allowed is because A15 is used for the apply argument, and A14 is used for generic functions and lexical closures, see section 1.7.3 on page 118.

If the function requires a data stack block the function's entry code builds the block immediately after checking for the correct number of arguments. Subroutines will probably handle the more complex cases of entry and return. Because these subroutine's access the routines registers, they will store a return address in a register (perhaps in the build window) and branch rather than use the *Call* instruction. The routine that is branched to will have to know that it was called in this manner and upon return will get the return address from the register and move the contents of the register into the PC.

In order to reduce the overhead of building up and taking down data stack blocks, the virtual memory system will be used to enforce the stack limits. The pages immediately above and below the legal range of the Data Stack will be non-existent, or have the fault-request bit set, in the PHT. The virtual memory system will be responsible for issuing stack underflow or overflow errors. This same mechanism will be applied to the Binding Stack.

While every compiled function will contain entry code to confirm that the caller and callee agree about the number of passed arguments, there will also be entry points that will allow a "pre-matched" caller to directly enter with the assurance that the correct number of arguments have been supplied in the expected places. In some cases, such as &optional arguments, there will be multiple entry points depending on the number of arguments supplied by the caller. When a function is recompiled all the old callers must be patched to point to the new compiled code.

For direct linked calls, the call instruction will load the N-Args register even though the callee never looks at it. For non direct linked calls, control passes to an entry sequence at the beginning of the function which does some testing, trapping, and dispatching based on the N-Args Register.

1.7.2 Entry

A compiled function starts with a sequence of instructions that are involved with ensuring that the caller and the callee agree on the number and location of arguments being passed. The following is a list of conditions that are needed by the entry code to perform its function:

- too few spread arguments N-Args < min-args
- too many spread arguments N-Args > max-args
- Maximum spread arguments N-Args = max-args
- apply argument supplied N-Args ≥ 16
- rest arguments wanted

where N-Args is the N-Args register setup by the caller, min-args is the minimum number of spread arguments the callee needs, and max-args is the maximum number of spread arguments the callee needs. Typically min-args = max-args, they are different when the callee uses &optional.

The most common and simplest case, is when no rest argument is wanted by the callee. In this case if min-args ≤ N-Args ≤ max-args we have the right number of arguments and only need to dispatch into a table to setup optional arguments not supplied. If N-Args < min-args or N-Args > max-args, take a wrong number of arguments trap. This trap will do a pull-apply-args if the apply bit is set in the N-Args register and the number of args passed is < max-args.

The pull-apply-args operation is used when there are fewer than the maximum number of spread arguments and the apply bit is set. It pulls some additional spread arguments out of the apply argument. This may result in a correct match up of arguments, or it may result in a wrong number of arguments error.

The push-apply-args operation is used when there are too many spread arguments, and a rest argument is wanted. It pushes some of the spread arguments back into the apply argument. After the arguments are pushed onto the apply list, the function has the right number of arguments, right where it expects them and execution of the function is started.

Push-apply-args and pull-apply-args are difficult routines to write code for due to a number of reasons. They can not be "called" because they need to deal with the current window, and a call would create a new current window and there would be no access to the old current window. The only way to reference registers in windows is explicitly in the instructions, it is not possible to use the contents of some register as the source of the register value.

1.7.3 Generic Functions

A generic function is a function whose implementation dispatches on the flavor of its first argument and selects a method that gets called as the body of the generic function. When the compiler compiles a call to a function, it generates code to set up the build window in a normal fashion, and to call the function in a normal fashion. If the generic function form looked like (generic-function object arg), the build window would look like this:

```
...
A1 arg
A0 object
```

and the compiled code would look like this:

```
move    A0 ← object
call    generic-function,N-Args ← 2,A1 ← arg
```

When the generic-function is defined its function cell will point to code that will move the generic function into some known location and then do a generic dispatch. We choose A14 as the location into which the generic function is moved as A15 is already defined for the apply argument and we don't want to have to shuffle arguments around to make room for the generic function, which is an argument of sorts. So the generic dispatch code will see windows that look like this:

```
R14 generic function
...
R1  arg
R0  object
```

The generic dispatch code will check for at least one argument, and pull apply args if necessary to get it; without at least one argument an error occurs. The argument is checked to insure that its data type is one of the instance data types. If its data is one of the instance data types, load its header, otherwise use the data type to index a 64-element table in the trap vector that points to the hash-mask fields of the flavor descriptions. Next, load the hash mask and the hash-table address of the flavor and perform the handler hash table search. If a key is found that matches the function, store its parameter mapping table in R14 (where the generic function is)(if the parameter mapping table is NIL, the generic function is supplied instead.) and terminate the search. If a key is found that is nil, terminate the search. If the method found has a data type of *dtp-pc* or *dtp-compiled-function* jump to the address, otherwise trap as an error.

1.7.4 Message Passing

When using flavors and methods, the form of (send object message arg) or (funcall object message arg) a RR format call will be used with the type check field ≠compiled-function. This will trap on data types of dtp-instance, and the trap routine will set up the arguments properly, with the message in A14, the pointer to the called object in A0 and the arguments in A1 thru A13. Then it will do a generic dispatch (see section 1.7.3 on page 118).

1.7.5 Lexical Closures

Lexical closures are called in the same way as a normal funcall is done, but the call instruction will trap due to the value being moved into the PC being something other than a dtp-compiled-function (this trap is caused by the type check in the call being ≠dtp-compiled-function). The trap routine dispatches on the type of the thing being loaded into the PC (this same trap is used for generic function funcall) and in the case of a lexical closure it will get the environment into the caller's a14 and the code into the trap

result register; it will then return-subvert which will cause the call instruction to try to load the contents of the trap result register into the PC; if the closure's code is just normal compiled code this will result in a normal call; if the code is not compiled then another trap will occur which may have to emulate the call instruction. Here is a sketch of the code:

```
call pc←rn ≠dtp-compiled-fuction  ;rn has dtp-lexical-closure in it
        ....


trap:   branch-type-next a1,service-lexical,#dtp-lexical-closure
        ....    ;generic-function, etc

service-lexical: load-car-cdr r14    ←[a1]    ;the environment
        load-cdr-finish trap-result ←[a1]     ;the code
        return-subvert scr          ←a0       ;restore scr
```

1.7.6 Return

A function that returns a single value simply stores that value in its R15 register. The caller retrieves the value by reading its A15 register. All functions must set the number of returned values in the N-Args register. If multiple values are returned, all returned values are put into the window, starting with R15, R14, R13 .... If more than 16 values are returned they should be consed, and the last value returned should be a pointer to this consed list. If 0 values are returned 0 is put into the N-Args register and NIL is stored in R15 so only receivers of multiple values need to check the N-Args Register. The n-args register contains the number of values being returned. The maximum number of returned values is 32.

If the caller is doing a call-for-return it needs to copy all arguments in the window that are being returned. If tail recursion removal is used this will not occur.

1.8 EXCEPTIONS

Exceptions, either interrupts or traps, temporarily redirect the instruction fetching mechanism off to exception handling code. Because the hardware behaves similarly to a call instruction, this code can act much like a typical function. However, since the exception may occur while instructions are setting up a build window, the handler code must restrict its use of window registers. The code can only use its own build window registers. Some handlers must also reside in wired physical memory.

Traps automatically receive three arguments as part of hardware support and interrupts receive one. Because the exception may change some internal processor state, the hardware provides a way to save the status conditions before the handler starts executing. Then, hardware can resume the processor state when the trap routine finishes. For this purpose, the exception handlers expect to receive the Status Control Register as an argument. The trap handlers expect Status Control Register as one of the three arguments passed; the interrupt handlers expect it as the only argument passed.

Since exception handlers may only use build window registers, the hardware must place exception

arguments in the build window of the handler routine, not the build window of the trapped routine as would a normal function call. Note that if a sufficiently complex handler needs to call a function, it can use the arguments it received in the build window as arguments to a new function call. Likewise, if the handler needs more than 16 registers, it could perform a function call before writing into any of the Build Window Registers.

1.8.1 *Interrupts*

Interrupts come from the memory control logic that attaches to the IBUS. The Sunstone processor reads one of the memory control interrupt registers with a *read-mc-reg* instruction. The interrupt routine software must clear the bit by writing a 1 as reading does not clear the bit. As in the I-Machine, software must service all asserted bits at a given interrupt level before returning from an interrupt routine.

An interrupt has one argument automatically saved in the build window for it, the Status Control Register. So when Sunstone enters an interrupt routine, enough machine state has already been saved to avoid the need to lock out all interrupts or traps. The software restores the Status Control Register during the *return* instruction at the end of the interrupt routine. Interrupt Level 1 is used to signal preemption. Preemption is the operation that stops one process and allows the scheduler to select another process to run. Because certain operations must be executed atomically across several instructions, preempt interrupts can be inhibited by a special bit in the instruction and/or Status Control Register while other interrupts continue to be allowed.

If at the time the Interrupt Level 1.Preempt Routine runs, preempts are disabled for software reasons, the routine sets a bit the Global Flags Register. Certain routines that might have inhibited preempts will check that bit using a trap-logtest instruction. The trap routine will cause a preempt if it is allowed.

Note that the trap routine itself will have to run with preempts disabled to prevent a race condition with the Level 1 preempt routine. Perhaps the the trap routine should simply check preemptability and then generate a level 1 interrupt if preempts are allowed.

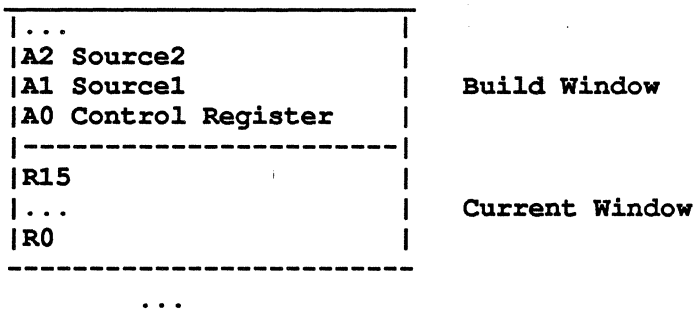
1.8.2 Traps

Getting traps to work efficiently on a register window machine like Sunstone presents an interesting problem. Because of the way that the sources and destination are embedded in the instruction, it would be difficult for the trap routine to emulate the instruction without special hardware assistance. This hardware assistance is provided by two mechanisms in Sunstone: the trap arguments and the subvert instruction return.

Whenever a trap occurs the trap routine is entered as if it had been called as a function. The Window Buffer Pointer is incremented and the PC of the instruction that caused the trap is pushed on top of the Return Address Buffer (a normal call pushes the PC of the next instruction onto the Return Address Buffer). Unlike a normal call, however, the Build Window of the instruction that caused the trap is not available to pass arguments to the trap routine, since it may have been in use to pass arguments to a

routine that the trapped routine was going to call. Therefore the trap routine must use its build window, rather than its normal register window. The hardware passes the trap arguments into the trap routine's Build Window, specifically into A0, A1 and A2. A0 receives the control register, and A1 & A2 receive the values of the Source1 and (if any) Source2 of the trapped instruction. See figure 22 on page 122. For instruction traps, the opcode is part of the trap vector address to facilitate efficient handling of the traps.

Figure 22. Windows After a Trap



There are three general kinds of operations that a trap routine can perform

1. modify the environment so that the instruction can execute
2. emulate the instruction and produce the desired result
3. generate an error

An example of the first type is a map miss; in that case the trap routine is responsible for refilling the map cache, or maybe swapping in a page. It then simply returns to retry the instruction.

The second case is more interesting. An example of this is an add that has bignum arguments. In this case, since the trap routine has the Source1 and Source2, its easy for it to perform the operation and produce the result. What is hard is putting that result into the proper place. This is done by an operation known as subverting the instruction. The trap handler must write its result into the trap-result register. The trap routine returns with a *return-subvert* instruction, which sets the Subvert Bit in the control register. See the *return-subvert* instruction section 1.5.5 54.

The third case, generating an error, is the easiest, since the debugger can do lots of work to figure out what to do about the problem without having any effect on the performance of the machine.

1.8.2.1 *Arithmetic Type Traps*

Arithmetic type traps are only used if the opcode is one of: *add, sub, or mult.* For *add* and *sub* the trap occurs when both sources have a numeric data type and the type check field is ≠hardware-arith, and both are not *dtp-fixnum.* However, if the FPU configuration register indicates that a floating point coprocessor is present, a trap will not occur if both sources are *dtp-single-float.* For *mult* the FPU

configuration register indicates if there is hardware available to perform either fixed point or floating point multiplies or both. The trap vector address includes information about the opcode, and the data type of the sources.

1.8.2.2 *Branch Type Traps*

Branch type traps are only used if there is a data type error. The trap occurs when both sources have a numeric data type and the cond field is one whose data type test is ≠ hardware-arith, and both are not *dtp-fixnum*. However, if the FPU configuration register indicates that floating point comparison is available, a trap will not occur if both sources are *dtp-single-float*. The trap vector address includes information about the opcode, the condition being tested and the data types of the two sources.

1.8.2.3 *Type Traps*

All type traps that don't qualify as an arithmetic type trap or a branch type trap, see 1.8.2.2 above, are type traps whose vector includes information about the opcode and the type being tested and a bit for the format to distinguish between RR format instructions which test both sources, and RIL and RIS format, which only test one source. This works very efficiently for all traps except possibly for store and *dpb* instructions. These instructions have three arguments, and we are only saving two on a trap. For some of the cases this still works fine, as for *store-array* (see section 1.10.4 on page 137), and for some it might take extra cycles to read the instruction from memory and emulate it. We expect this to be a very rare occurrence, and typically only on error traps.

1.8.2.4 *Load Traps and Store Traps*

There are four kinds of load traps and store traps: Indirect, Error, Monitor, Transport. The trap vector address includes information on the opcode that trapped and which of the traps occurred.

1.8.2.5 *Illegal Instruction Trap*

Illegal instruction traps are traps on illegal opcodes, or illegal data types. The opcode and the format, or the data type is part of the trap vector address. Illegal instructions can either be errors or can be interpreted as extensions to the instruction set.

1.8.2.6 *Miscellaneous Opcode Specific Traps*

There are several other traps associated with particular opcodes. These traps are listed in the table 12 on page 124. The trap vector address includes some bits from the opcode to identify the instruction that trapped, and some bits to specify the particular trap. *Load-array* and *store-array* instructions which trap due to packed array data will have a trap vector location with eight locations available for code to handle the trap, this makes handling of these traps more efficient by not having to perform another branch.

Table 12. Misc Opcode Specific Traps

MISC OPCODE SPECIFIC TRAPS	
OPCODE	TRAP
add	coprocessor overflow
sub	coprocessor overflow
mult	coprocessor overflow
ash	overflow
load-coproc	coprocessor
store-coproc	coprocessor
read-coproc	coprocessor
write-coproc	coprocessor
load-cdr-finish	cdr-reg = 3
load-array	array element prefix long reference out of range packed array
store-array	array element prefix long reference out of range packed array array element
store-rplacd	cdr-reg not cdr-normal

1.7.2.7 *General Traps*

General traps are traps that each have single unique trap addresses; the trap addresses do not depend on the opcode. Many of these traps can occur on any type of instruction, e.g. Instruction Map Miss can happen any instruction reference if the PC is not in VMA=PMA space. Many of these traps can only happen on LOADs and/or STOREs, e.g. Data Transport, Bound Location.

There are several general traps whose trap vector address is independent of the instruction executing. They are listed below:

- *Window Buffer Overflow* - This trap can happen on any CALL instruction, or it can happen on any instruction that traps or is interrupted. This trap happens whenever a new window is needed and none is available; the software must insure that the window buffer overflow limit is set to a level that will cause the trap to happen before a valid window is accidently overwritten. When an instruction that needs a window attempts to execute, if the new (incremented) value of the window buffer pointer is equal to the window buffer overflow limit the instruction will trap to the window buffer overflow trap; note that if the instruction was already trapping or being interrupted the window buffer overflow trap will take precedence. To state the occurrence of this trap in parallel terms to the underflow trap, Window Buffer overflow occurs during the call instruction if the build window number before the WSP is incremented equals the overflow limit.
- *Window Buffer Underflow* - Window Buffer underflow occurs during a return instruction if (prior to the decrement) the build window number (which is one greater than WBC bits 3:0) matches the underflow limit.
- *Instruction Completion* - Any instruction can get an instruction completion trap; the trap is caused by the take instruction completion trap bit being set in the status control register; if that bit is set, the trap occurs immediately. Note that the take instruction completion trap bit is set by an instruction completing execution when the trap on instruction completion bit is set, or by a call or return instruction completing execution when the trap on call or trap on return (respectively) bit is set.
- *Instruction Fetch Hardware Error* - This trap indicates that a hardware error (bus error, double bit ECC error, etc.) happened when attempting to fetch the instruction. Although it is implementation dependent, it is a goal that this trap will happen on exactly the instruction that got the error; it is a requirement that this trap not happen if a instruction that has been prefetched gets an error, if the prefetched instruction is never executed. The exception to this is if the word in error is the odd word and the even word is a single word instruction; the error will probably occur on the first instruction, even though it is actually in the word that has not yet been executed.
- *Data Fetch Hardware Error* - This trap can happen on a load or store instruction that gets a hardware error on reading or writing data. Note that while it is implementation dependent, it is likely that only errors on reading data will actually cause traps; errors on writes will probably be ignored. Some store instructions explictly do a read before a write, and if there is an error on the read it will cause the store instruction to trap.
- *Data Fetch Map Miss on Load Instruction* - If a load instruction references a virtual location that is not in the data cache and does not have a mapping in the map cache, this trap will start the map cache refill routine. Note that a word that is in the data cache but does not have a mapping in the map cache will not get a map miss trap on a load instruction, but it will get a map miss trap on a store. Virtual addresses in the VMA=PMA area never cause map miss traps.

1.8.2.7 *General Traps*

General traps are traps that each have single unique trap addresses; the trap addresses do not depend on the opcode. Many of these traps can occur on any type of instruction, e.g. Instruction Map Miss can happen any instruction reference if the PC is not in VMA=PMA space. Many of these traps can only happen on LOADs and/or STOREs, e.g. Data Transport, Bound Location.

There are several general traps whose trap vector address is independent of the instruction executing. They are listed below:

- *Window Buffer Overflow* - This trap can happen on any CALL instruction, or it can happen on any instruction that traps or is interrupted. This trap happens whenever a new window is needed and none is available; the software must insure that the window buffer overflow limit is set to a level that will cause the trap to happen before a valid window is accidently overwritten. When an instruction that needs a window attempts to execute, if the new (incremented) value of the window buffer pointer is equal to the window buffer overflow limit the instruction will trap to the window buffer overflow trap; note that if the instruction was already trapping or being interrupted the window buffer overflow trap will take precedence.
- *Window Buffer Underflow* - If a return or a return-subvert instruction would cause the window buffer pointer to be equal to the window buffer underflow limit, this trap is taken.
- *Instruction Completion* - Any instruction can get an instruction completion trap; the trap is caused by the take instruction completion trap bit being set in the status control register; if that bit is set, the trap occurs immediately. Note that the take instruction completion trap bit is set by an instruction completing execution when the trap on instruction completion bit is set, or by a call or return instruction completing execution when the trap on call or trap on return (respectively) bit is set.
- *Instruction Fetch Hardware Error* - This trap indicates that a hardware error (bus error, double bit ECC error, etc.) happened when attempting to fetch the instruction. Although it is implementation dependent, it is a goal that this trap will happen on exactly the instruction that got the error; it is a requirement that this trap not happen if a instruction that has been prefetched gets an error, if the prefetched instruction is never executed. The exception to this is if the word in error is the odd word and the even word is a single word instruction; the error will probably occur on the first instruction, even though it is actually in the word that has not yet been executed.
- *Data Fetch Hardware Error* - This trap can happen on a load or store instruction that gets a hardware error on reading or writing data. Note that while it is implementation dependent, it is likely that only errors on reading data will actually cause traps; errors on writes will probably be ignored. Some store instructions explictly do a read before a write, and if there is an error on the read it will cause the store instruction to trap.
- *Data Fetch Map Miss on Load Instruction* - If a load instruction references a virtual location that is not in the data cache and does not have a mapping in the map cache, this trap will start the map cache refill routine. Note that a word that is in the data cache but does not have a mapping in the map cache will not get a map miss trap on a load instruction, but it will get a map miss trap on a store. Virtual addresses in the VMA=PMA area never cause map miss traps.
- *Data Fetch Map Miss on Store* - If a store instruction references a virtual location that does not have a mapping in the map cache, this trap will start the map cache refill routine. This

routine is separate from the similar load trap because the trap routine's computation of the referenced address is different for the load and store instructions.

- *Instruction Fetch Map Miss* - If an instruction fetch misses in the cache and the map cache, this trap will happen on the instruction that would have been fetched.
- *Instruction Fetch Transport Trap* - If an instruction word (either a direct branch or the immediate data of an RIL format instruction) points to oldspace this trap will happen when an attempt is made to execute the instruction, if transport traps are enabled on the page that the instruction word was fetched from.
- *Data Fetch Transport Trap* - If the word read by a load instruction is a pointer type, and the pointer points to oldspace, and the page the word is on has transport traps enabled, and the instruction is one of load-data, load-data-iv, load-car-cdr, load-cdr-finish, load-array, load-coproc, load-bind or load-scavange, then this trap is taken.
- *Error Trap* - If the word read by a load or store instruction should generate an error, according to the memory operation table x, this trap is taken.
- *Monitor Trap* - If the word read by a load or store instruction should generate an monitor trap, according to the memory operation table x, this trap is taken.
- *Bound Location Trap* - If the word read by a load or store instruction should generate an bound location trap, according to the memory operation table x, this trap is taken.
- *Logic Variable Trap* - If the word read by a load or store instruction should generate an logic variable trap, according to the memory operation table x, this trap is taken.
- *Write Protect Trap* - If the page that is to be written by a store instruction has its write-protect bit set in the map cache, the write is not done and this trap is taken.
- *Page Modified Update Trap* - If the page that is to be written by a store instruction has its page-modified bit clear in the map cache, the write is not done and this trap is taken. The trap routine will set the bit and return to reexecute the store.
- *Ephemeral Reference Update Trap* - Ephemeral reference update traps are used to keep the 4 ephemeral reference level bits in the PHT entry for a page updated. Each of the four bits is used to indicate that there may exist a pointer on the page that points a word in a particular ephemeral level group. (Each ephemeral level group contains eight ephemeral levels.)

  A store instruction will get an ephemeral reference update trap if it attempts to store a word which is a pointer to an ephemeral level group which the page being stored into did not previously point to, as indicated by the corresponding map cache bit being zero for that page. The trap routine will set the bit to a one in the map cache entry and the PHT and then return to reexecute the store instruction.

  More precisely, if the word being stored is a pointer type, and bits <31:27> of the pointer are zero, bits <25:24> are used to select one of the four bits; if the selected bit is zero, a trap is taken. However, as a special optimization, if the address being stored into has bits <31:27> equal to zero and bits <25:21> of the address equal bits <25:21> of the stored data, the trap is inhibited.

1.8.2.8 *Trap Vector*

The trap vector address points into a section of memory which contains a table of direct-branch instructions. This table must reside in VMA=PMA address space. The words in this table point to the trap handler routines. In the case of packed array traps there is not a direct-branch instruction pointing to the routine as the code is in line starting at the vector address.

The trap vector address requires a trap address base for locating this table in memory. The trap-base register contains this address.

The offset for the trap vector address comes from further trap specification. Table 13 on page 128 lists each trap classes and the traps belonging to the class. For each trap, the table shows trap vector address (See figure 23 on page 127).

Figure 23. Trap Vector Format

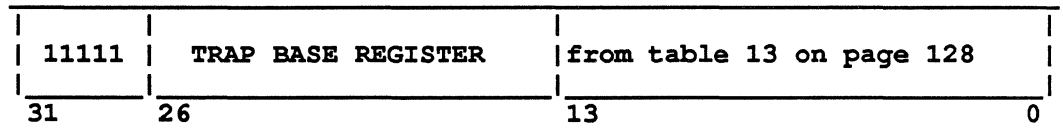


Table 13. Trap Vector Addresses

EXCEPTION VECTOR ADDRESS

Exception	13	12	11	10	9	8	7	6	5	4	3	2	1	0
trap-type,trap	0	0	Immediate<11:0>											
type	0	1	Type-Chk<4:0>				*	Op-code<5:0>						
branch-type-ex	1	0	TypChk<2:0>			Src1<2:0>			Src2<2:0>			0	0	0
arith-type-exc	1	1	0	Ah<1:0>		Src1<2:0>			Src2<2:0>			0	0	0
illegal-inst	1	1	1	0	0	Format<2:0>			Op-code<5:0>					
misc-opcode	1	1	1	0	1	Misc<3:0>				OT<1:0>		0	0	0
indirect	1	1	1	1	0	0	Op-code<4:0>					0	0	0
general	1	1	1	1	0	1	0	0	0	G-Traps<4:0>				
interrupts	1	1	1	1	1	0	0	0	0	0	0	Inter<2:0>		
reset	1	1	1	1	1	1	1	1	1	1	1	1	1	0
initialize	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Trap-Type and Trap instructions supply the lower 12 bits of the interrupt vector to be their immediate values.

Type trap vector uses the full opcode and the LSB of the format field to determine the instruction type and format. The Type-Check field is used as a condition field.

*Format<0> 0 RR Instruction 1 RIS or RIL

Branch-type trap vector uses the Type-Check(TypChk) field as the condition field itemized in Table 9 page 25. Also the data types of source-1(Src1) and source-2(Src2) which are numeric are used.

Arithmetic type trap vector uses source-1 and source-2 as in Branch-type trap and the Opcode to create the Ah field which indicates the type of arithmetic instruction:

Ah<1:0> 0 ADD 1 SUB 2 MULT

Misc-opcode trap vector uses the opcode to generate a Misc opcode field to indicate the instruction specific trap. The Operation Type field indicates the type of trap that occured for that instruction.

Misc<3:0>	00 Ash	01 Load-cdr-finish
	02 Load-array	03 Load-coproc
	05 Store-rplacd	06 Store-array
	07 Store-coproc	10 Add
	11 Sub	12 Mult
	13 Write-coproc	17 Read-coproc
OT<1:0> (Op-Type)	0 Array-Element, Coprocessor, Rplacd,Illegal-Cdr-3	
	1 Array-Long-Pre, Overflow	
	2 Array-Element	
	3 Array-Packed	

Table 12. Trap Vector Addresses

EXCEPTION VECTOR ADDRESS

Exception	13	12	11	10	9	8	7	6	5	4	3	2	1	0
trap-type,trap	0	0	Immediate<11:0>											
type	0	1	Type-Chk<4:0>				*	Op-code<5:0>						
branch-type-ex	1	0	TypChk<2:0>			Src1<2:0>		Src2<2:0>			0		0	0
arith-type-exc	1	1	0	Ah<1:0>		Src1<2:0>		Src2<2:0>			0		0	0
illegal-inst	1	1	1	0	0	Format<2:0>		Op-code<5:0>						
misc-opcode	1	1	1	0	1	Misc<3:0>				OT<1:0>		0	0	0
indirect	1	1	1	1	0	0	Op-code<4:0>					0	0	0
interrupts	1	1	1	1	1	1	1	0	Inter<2:0>			0	0	0
general	1	1	1	1	1	1	1	1	0	0	G-Trap<3:0>			
wb-overflow	1	1	1	1	1	1	1	1	1	1	1	1	0	0
reset	1	1	1	1	1	1	1	1	1	1	1	1	1	0
initialize	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Trap-Type and Trap instructions supply the lower 12 bits of the interrupt vector to be their immediate values.

Type trap vector uses the full opcode and the LSB of the format field to determine the instruction type and format. The Type-Check field is used as a condition field.

*Format<0> 0 RR Instruction 1 RIS or RIL

Branch-type trap vector uses the Type-Check(TypChk) field as the condition field itemized in Table 9 page 25. Also the data types of source-1(Src1) and source-2(Src2) which are numeric are used.

Arithmetic type trap vector uses source-1 and source-2 as in Branch-type trap and the Opcode to create the Ah field which indicates the type of arithmetic instruction:

Ah<1:0> 0 ADD 1 SUB 2 MULT

Misc-opcode trap vector uses the opcode to generate a Misc opcode field to indicate the instruction specific trap. The Operation Type field indicates the type of trap that occured for that instruction.

Misc<3:0>	00 Ash	01 Load-cdr-finish
	02 Load-array	03 Load-coproc
	05 Store-rplacd	06 Store-array
	07 Store-coproc	10 Add
	11 Sub	12 Mult
	13 Write-coproc	17 Read-coproc
OT<1:0>	0 Array-Element, Coprocessor, Rplacd,Illegal-Cdr-3	
(Op-Type)	1 Array-Long-Pre, Overflow	
	2 Array-Range	
	3 Array-Packed	

Table 12. Trap Vector Address Cont.

Indirect trap vector uses the opcode to indicate which Load or Store Instruction trapped.

General-traps are independent of the instruction that is executing only a code is given for the type of trap that occurs. They are prioritized by order.

G-Trap<3:0>
00 Inst-Completion
01 Inst-Map-Miss
02 Inst-Hardware-Err
03 Inst-Transport
04 WB-Underflow
05 Data-Map-Miss(Load)
06 Data-Map-Miss(Store)
07 Data-Hardware-Err
10 Error-Trap(Memory Cycle)
11 Data-Transport
12 Logic-variable
13 Bound-variable
14 Monitor-Trap
15 Write-Protect
16 Page-Mod-Update
17 Eph-Ref-Update

Interrupt-trap vector is generated by a three bit vector interrupt vector as described below.

Interrupt-Vector<2:0> 0 no interrupts or preempts
1 preempt
2-7 interrupts 2-7

Reset trap vector is issued when the reset signal is asserted.

Initialize trap vector is issued when the init signal is asserted.

1.7.2.9 Exception Priorities

Hardware determines the priority in which exceptions are handled. The following text and table 13 on page 130 list the exceptions that can occur in the priority that the hardware handles them. It also provides a cross reference to the trap or interrupt vector that can be used in conjunction with Table 12 to determine the vector address.

Initialize **Initialize-Exc**

This trap must be the highest priority because it forces the machine into a cleared state unconditionally.

Table 13. Trap Vector Address Cont.

Indirect trap vector uses the opcode to indicate which Load or Store Instruction trapped.

General-traps are independent of the instruction that is executing only a code is given for the type of trap that occurs. They are prioritized by order.

G-Traps<4:0> 00 WB-Overflow
20 Inst-Completion
21 Inst-Map-Miss
22 Inst-Hardware-Err
23 Inst-Transport
24 WB-Underflow
25 Data-Map-Miss(Load)
26 Data-Map-Miss(Store)
27 Data-Hardware-Err
30 Error-Trap(Memory Cycle)
31 Data-Transport
32 Logic-variable
33 Bound-variable
34 Monitor-Trap
35 Write-Protect
36 Page-Mod-Update
37 Eph-Ref-Update

Interrupt-trap vector is generated by a three bit vector interrupt vector as described below.

Interrupt-Vector<2:0> 0 no interrupts or preempts
1 preempt
2-7 interrupts 2-7

Reset trap vector is issued when the reset signal is asserted.

Initialize trap vector is issued when the init signal is asserted.

1.8.2.9 Exception Priorities

Hardware determines the priority in which exceptions are handled. The following text and table 14 on page 133 list the exceptions that can occur in the priority that the hardware handles them. It also provides a cross reference to the trap or interrupt vector that can be used in conjunction with Table 13 to determine the vector address.

Initialize **Initialize-Exc**

This trap must be the highest priority because it forces the machine into a cleared state unconditionally. It must be able to do that in the face of any other trap causing conditions, and also arrange to execute instructions without any further traps.

Reset	**Reset-Exc**

This acts as a non-maskable interrupt. It can be argued that this should be lower priority than window buffer overflow, but that doesn't matter because it could still interrupt the window buffer overflow routine. In most cases it will be possible to continue from a reset trap, but not always.

WB-Overflow	**General-Trap**

This must be higher priority than any other traps because the other traps could otherwise destroy the stack. It inhibits all interrupts until there is room in the window buffer.

Interrupts	**Interrupt-Exc**
Preempts	**Interrupt-Exc**

These two similar cases are high priority in order to keep latency low and allow interrupts out of trap loops.

Inst-Completion	**General-Trap**

From here down are traps that relate to trying to fetch and then execute the next instruction; this trap is a result of successfully executing the previous instruction, so it is higher priority.

The next few traps are related to fetching instructions; until we have a valid instruction and a valid set of registers, no traps related to trying to execute an instruction make sense.

Inst-Map-Miss	**General-Trap**

If we don't know the physical address to fetch at, nothing else matters.

Inst-Hardware-Error	**General-Trap**

Once we have a good address, if the data is bogus we shouldn't try to decode it.

Inst-Transport-Trap	**General-Trap**

It's important that this comes before illegal instruction, in case we decide that it is valid to put other pointer types into the instruction stream that aren't valid instructions (e.g. generic functions). We don't currently plan to do that, but it can't hurt to make it possible.

Illegal-Inst	**Illegal-Instruction**

Now that we have a real word that thinks it's an instruction, is it any good?

WB-Underflow	**General-Trap**

Now we have a good instruction, the next thing to check is if it's arguments are any good; but if the window buffer is underflowed, the registers aren't valid. It's not important for window buffer underflow

to be higher priority than this because this is the first time that the register contents matter, and any other trap will make the windows valid again. Actually, since this trap only happens on return instructions the priority probably isn't important.

```
Type-Type-Trap          Type-Trap
Type-Arith-Trap         Arithmetic-Trap
Type-Branch-Trap        Branch-Trap
```

Three variants of "the registers have bad stuff in them", which we must catch before we try to use those values for anything.

```
Array-Element           Misc-Opcode
```

This is similar to a type trap, again a case of bad stuff in the registers.

```
Array-Long-Pre          Misc-Opcode
```

If the array is long prefix, we haven't really got good header information, which the next traps depend on.

```
Array-Range             Misc-Opcode
```

If we are out of range we don't want to try to do the packed reference, or anything else.

```
Array-Packed            Misc-Opcode
```

If everything else about the array reference is okay, go and do the packing/unpacking.

```
Rplacd                  Misc-Opcode
```

This is similar to a type trap, since it depends on the contents of the cdr register from a previous load instruction. This must be higher priority than actually doing the memory reference.

```
Coproc-Trap             Misc-Opcode
Overflow                Misc-Opcode
```

These are based on the results of an otherwise valid operation needing special attention.

```
Data-Map-Miss(Load)     General-Trap
Data-Map-Miss(Store)    General-Trap
```

Now that we have a valid load or store operation to perform, these traps happen if we don't have a valid address mapping.

```
D-Hardware-Error        General-Trap
```

If we have a valid address to reference, but that references screws up somehow.

```
Error-Trap(Mem)         General-Trap
Data-Transport-Trap     General-Trap
Indirect-Trap           Indirect-Trap
Logic-Variable          General-Trap
```

```
Bound-Location          General-Trap
Monitor-Trap            General-Trap
```

These traps are dependent on the data that was read from a valid load or store instruction. Their priority is defined by the architecture.

```
Write-Protect           General-Trap
Page-Mod-Update         General-Trap
Eph-Ref-Update          General-Trap
```

Now that we've gotten all the way to attempting a valid store, the map needs to be fixed up. Write protect is the highest priority because the other two traps assume that a valid write will eventually complete, after they fix up the map entry; the order of page-mod-update and ephemeral reference update is arbitrary.

```
Trap-Inst-True          Trap-Type
```

This is actually a sort of type trap, and could be a higher priority since it is orthogonal to everything that came after the type traps.

Table 13. Exception Priorities

EXCEPTIONS PRIORITIZED					
Priority	Exception	Vector	A0	A1	A2
1.	Initialize	Initialize-Exc	SCR	-	-
2.	Reset	Reset-Exc	SCR	-	-
3.	WB-Overflow	General-Trap	SCR	-	-
4.	Interrupts	Interrupt-Exc	SCR	-	-
5.	Preempts	Interrupt-Exc	SCR	-	-
6.	Inst-Completion	General-Trap	SCR	-	-
7.	Inst-Map-Miss	General-Trap	SCR	-	-
8.	Inst-Hardware-Error	General-Trap	SCR	-	-
9.	Inst-Transport-Trap	General-Trap	SCR	-	-
10.	Illegal-Inst	Illegal-Instruction	SCR	-	-
11.	WB-Underflow	General-Trap	SCR	-	-
12.	Type-Type-Trap	Type-Trap	SCR	S1	S2/Imm
13.	Type-Arith-Trap	Arithmetic-Trap	SCR	S1	S2/Imm
14.	Type-Branch-Trap	Branch-Trap	SCR	S1	S2/Imm
15.	Array-Element	Misc-Opcode	SCR	S2*	adj-index*
16.	Array-Long-Pre	Misc-Opcode	SCR	S1/S2*	adj-index*
17.	Array-Range	Misc-Opcode	SCR	S1/S2*	adj-index*
18.	Array-Packed	Misc-Opcode	SCR	S1/S2*	adj-index*
19.	Rplacd	Misc-Opcode	SCR	S2**	Address**
20.	Coproc-Trap	Misc-Opcode	SCR	S1	S2/Imm
21.	Overflow	Misc-Opcode	SCR	S1	S2/Imm
22.	Data-Map-Miss(Load)	General-Trap	SCR	S1**	Address**
23.	Data-Map-Miss(Store)	General-Trap	SCR	S2**	Address**
24.	D-Hardware-Error	General-Trap	SCR	S1/S2**	Address**
25.	Error-Trap(Mem)	General-Trap	SCR	S1/S2**	Address**
26.	Data-Transport-Trap	General-Trap	SCR	S1**	Address**
27.	Indirect-Trap	Indirect-Trap	SCR	S1/S2**	Address**
28.	Logic-Variable	General-Trap	SCR	S1**	Address**
29.	Bound-Location	General-Trap	SCR	S1/S2**	Address**
30.	Monitor-Trap	General-Trap	SCR	S1/S2**	Address**
31.	Write-Protect	General-Trap	SCR	S2**	Address**
32.	Page-Mod-Update	General-Trap	SCR	S2**	Address**
33.	Eph-Ref-Update	General-Trap	SCR	S2**	Address**
34.	Trap-Inst-True	Trap-Type	SCR	S1	S2/Imm

* Load-Array stores S1 in A1 on array traps. Store-Array stores S2 (the value to be stored) in A1 on array traps. In RR-format Load-Array, the index is S2; in RR-format Store-Array, the index is S1; in RIS and RIL formats, the index is the immediate operand. All array traps store the adjusted index in A2, where the adjustment consists of adding the Array-Header byte offset field to the index, then logically shifting the sum right by the value of the Array-Header byte-pack field, and adding one to the shifted sum. The bits shifted out are saved in the position field of the byte-rotate register.

** Memory reference traps always put the address in A2. Load instructions put S1 in A1; Store instructions put S2 (the data to be stored) in A1.

Table 14. Exception Priorities

EXCEPTIONS PRIORITIZED		
Priority	Exception	Vector
1.	Initialize	Initialize-Exc
2.	Reset	Reset-Exc
3.	WB-Overflow	General-Trap
4.	Interrupts	Interrupt-Exc
5.	Preempts	Interrupt-Exc
6.	Inst-Completion	General-Trap
7.	Inst-Map-Miss	General-Trap
8.	Inst-Hardware-Error	General-Trap
9.	Inst-Transport-Trap	General-Trap
10.	Illegal-Inst	Illegal-Instruction
11.	WB-Underflow	General-Trap
12.	Type-Type-Trap	Type-Trap
13.	Type-Arith-Trap	Arithmetic-Trap
14.	Type-Branch-Trap	Branch-Trap
15.	Array-Element	Misc-Opcode
16.	Array-Long-Pre	Misc-Opcode
17.	Array-Range	Misc-Opcode
18.	Array-Packed	Misc-Opcode
19.	Rplacd	Misc-Opcode
20.	Coproc-Trap	Misc-Opcode
21.	Overflow	Misc-Opcode
22.	Data-Map-Miss(Load)	General-Trap
23.	Data-Map-Miss(Store)	General-Trap
24.	D-Hardware-Error	General-Trap
25.	Error-Trap(Mem)	General-Trap
26.	Data-Transport-Trap	General-Trap
27.	Indirect-Trap	Indirect-Trap
28.	Logic-Variable	General-Trap
29.	Bound-Location	General-Trap
30.	Monitor-Trap	General-Trap
31.	Write-Protect	General-Trap
32.	Page-Mod-Update	General-Trap
33.	Eph-Ref-Update	General-Trap
34.	Trap-Inst-True	Trap-Type

1.9 GARBAGE COLLECTION (GC)

The gc will operate much as in the I-Machine. Two registers will support this: the Zone Oldspace Register and the Ephemeral Oldspace Register.

The Zone Oldspace Register (ZOR) is identical in to the I-Machine's Zone Oldspace Register. The Zone Oldspace Register contains a bit map that specifies whether each zone of dynamic space (there are 29 zones) is newspace or oldspace. Bits 31 and 0 of the Zone Oldspace Register are typically 0. Bit 31 represents physical memory zones, and bit 0 represents the Ephemeral Zone. Since new/old space is a characteristic of virtual memory, bit 31 is set to 0 (VMA=PMA) and since there is a special Ephemeral Oldspace Register, bit 0 is set to 0 (this is the ephemeral space bit). A set bit indicates its corresponding zone is oldspace.

The Ephemeral Oldspace Register (EOR) is identical to the Ephemeral Oldspace Register in the I-Machine. The EOR contains a bit map that specifies, for each of the 32 ephemeral levels, which half of the level is newspace and which half is oldspace. A set bit indicates the upper half is oldspace, a reset bit indicates the lower half is oldspace.

Two traps are used to support GC: Transport Trap and Ephemeral Reference Update Trap (ER Update Trap). A memory read of a pointer to old space signals a Transport Trap when enabled. The EOR and ZOR are used as an efficient lookup in finding where oldspace exists. A memory write of a pointer to an ephemeral object signals an ER Update Trap when the ER bits from the PHT will no longer be accurate after the write. The trap routine updates the PHT.

Also in support of efficient GC there are four instructions: *load-scavenge*, *load-gc-copy*, *load-ephemeralp*, and *load-oldspacep*.

1.10 ARRAY REFERENCES.

Support for arrays requires special *load-array* and *store-array* instructions. (For more information on the *load-array* and *store-array* instructions see pages 78 and 93 respectively.) These instructions depend on two special registers: Array Header Register and Array Length Register. When these instructions execute, they use the array-header and array-length registers to detect various trap conditions and to determine whether the address calculation needs to be incremented. For loop optimization, Sunstone has its own version of array registers called array descriptors.

Typically, the instructions will also use the special MAR register for the base source, immediately following a load-header instruction. The non array register sequence for arrays looks like this:

```
*load-header  array-header ← [array-pointer]   ≠array
 load-array   destination  ← [MAR,index]       ≠array ;+1 to sources
```

The load-header instruction will follow invisible pointers, so that the actual header location pointer is

stored in the MAR register. These two instructions run with preempt inhibited to prevent the software in another process from changing the header location (eg, array-push-extend). When not using array descriptors, the load-array instruction will trap for long prefix arrays. Also, notice that the code does not explicitly load the Array Length Register; instead, the load-header instruction will load the Array Length Register as a side effect, when its destination is the Array Header Register.

1.9.1 Array Hardware Support

Sunstone provides hardware support for arrays in several ways. The following text lists some of the support provided.

- array header register
- array length register - automatically loaded during LOAD of Array-Header. Array-Length ← Array-Header<14:0>, if short; S-Data-Bus<71:40>, if long prefix.
- unique load/store array instructions - load/store-array adds one to the address calculation.
- load/store-array traps if array header has a long prefix and no long prefix inhibit
- store-array traps if the array element type is boolean, if the array element type is character and source2 isn't character or has bits set above the array element size (i.e., bit 20 is set for a 16-bit-string array), or if the array element type is fixnum and source2 isn't fixnum.
- Load/store-array traps if array header shows packed data or element type which differs from the type of the words in which elements are stored (character and boolean).
- unique trap address for packed array trap of load/store-array instructions with enough space in the trap vector to handle the trap (this is about 2 locations for *load-array*, and 5 for *store-array*).
- After load-array traps:

```
A0 ← SCR as in normal trap operation,
A1 ← array address (s1) as in normal trap operation,

A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
       where byte-spec = (byte 32. Array-Header-Register<29:27>)
       and byte-offset forced to 0 for short prefix arrays
```

- After store-array traps:

```
A0 ← SCR as in normal trap operation,
A1 ← data to store,

A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
       where byte-spec = (byte 32. Array-Header-Register<29:27>)
       and byte-offset forced to 0 for short prefix arrays"
```

- *Load-array* or *store-array* instructions load the byte rotate register with the size (actually size - 1 for FS) set by header bits <29:27>.

<29:27>	size
0	32
1	16

stored in the MAR register. These two instructions run with preempt inhibited to prevent the software in another process from changing the header location (eg, array-push-extend). When not using array descriptors, the load-array instruction will trap for long prefix arrays. Also, notice that the code does not explicitly load the Array Length Register; instead, the load-header instruction will load the Array Length Register as a side effect, when its destination is the Array Header Register.

1.10.1 Array Hardware Support

Sunstone provides hardware support for arrays in several ways. The following text lists some of the support provided.

- array header register
- array length register - automatically loaded during LOAD of Array-Header. Array-Length ← Array-Header<14:0>, if short; S-Data-Bus<71:40>, if long prefix.
- unique load/store array instructions - load/store-array adds one to the address calculation.
- load/store-array traps if array header has a long prefix and no long prefix inhibit
- store-array traps if the array element type is boolean, if the array element type is character and source2 isn't character or has bits set above the array element size (i.e., bit 20 is set for a 16-bit-string array), or if the array element type is fixnum and source2 isn't fixnum.
- load/store-array traps if array header shows packed data
- unique trap address for packed array trap of load/store-array instructions with enough space in the trap vector to handle the trap (this is about 2 locations for *load-array*, and 5 for *store-array*).
- After load-array traps:

```
A0 ← SCR as in normal trap operation,
A1 ← array address (s1) as in normal trap operation,

A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
       where byte-spec = (byte 32. Array-Header-Register<29:27>)
       and byte-offset forced to 0 for short prefix arrays
```

- After store-array traps:

```
A0 ← SCR as in normal trap operation,
A1 ← data to store,

A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
       where byte-spec = (byte 32. Array-Header-Register<29:27>)
       and byte-offset forced to 0 for short prefix arrays"
```

- *Load-array* or *store-array* instructions load the byte rotate register with the size (actually size - 1 for FS) set by header bits <29:27>.

<29:27>	size
0	32
1	16
2	8

```
3       4
4       2
5       1
```

Position (BB) is set by the product of the size and the least significant n bits of the sum of the index and the byte-offset field of the array-header register, where n is the value of the header bits <29:27>. In order to save on state for the ensuing subversion, non byte-pack load/store-array traps load the byte rotate register with size 32 and 0 bottom-bit.

- A special subvert mechanism for *load-array* instructions that does a ldb of the trap result register. It uses the value in the Byte Rotate Register, and merges in the proper data type as specified by the header register. It forces the result data type to character when the array element type is character. When the array element type is boolean, it forces the entire result to T or NIL depending on the lowest bit of the ldb'ed trap result register. For details on how array element type pertains to byte packing, refer to the table of valid array types below.

Table 15. Valid Array Types

Valid Array Types for Byte-Packing Values				
Array-byte packing	Fixnum	Character	Boolean	Object
0	art-fixnum	art-fat-string	xxx	art-q
1	art-16b	16-bit-string	xxx	xxx
2	art-8b	art-string	xxx	xxx
3	art-4b	xxx	xxx	xxx
4	art-2b	xxx	xxx	xxx
5	art-1b	xxx	art-boolean	xxx

1.10.2 Array Header Register

The Array Header Register contains the element type bits 31:30, byte pack bits 29:27; and long prefix bit 23 of the array header stored in memory. In addition, it contains a long prefix trap inhibit bit and byte offset bits. These read as 0 for a short prefix header and reflect bits<12,11:7> of a long prefix header encached as an array descriptor. These spare long prefix header bits must be set to 0 in the actual memory header location.

```
-------------------------------------------------------
|e-type|byte-pack|xxxx|lp|xxxx|lpi|byte-offset|xxxxxx|
-------------------------------------------------------
31   30 29     27      23       12 11        7

              x's mean not used
```

```
2       8
3       4
4       2
5       1
```

Position (BB) is set by the product of the size and the least significant n bits of the sum of the index and the byte-offset field of the array-header register, where n is the value of the header bits <29:27>. The hardware loads the byte rotate register even if the load/store-array instruction traps. When taking any one of the four array traps (element, range, long-prefix, and packed), it loads the byte rotate register with the value as specified above. For any other trap, or if the instruction does no trap, it loads the byte rotate register with size 32 and bottom bit 0.

- A special subvert mechanism for *load-array* instructions that does a ldb of the trap result register. It uses the value in the Byte Rotate Register, and merges in the proper data type as specified by the header register. It forces the result data type to character when the array element type is character. When the array element type is boolean, it forces the entire result to T or NIL depending on the lowest bit of the ldb'ed trap result register. For details on how array element type pertains to byte packing, refer to the table of valid array types below.

Table 14. Valid Array Types

Valid Array Types for Byte-Packing Values				
Array-byte packing	Fixnum	Character	Boolean	Object
0	art-fixnum	art-fat-string	xxx	art-q
1	art-16b	16-bit-string	xxx	xxx
2	art-8b	art-string	xxx	xxx
3	art-4b	xxx	xxx	xxx
4	art-2b	xxx	xxx	xxx
5	art-1b	xxx	art-boolean	xxx

1.9.2 Array Header Register

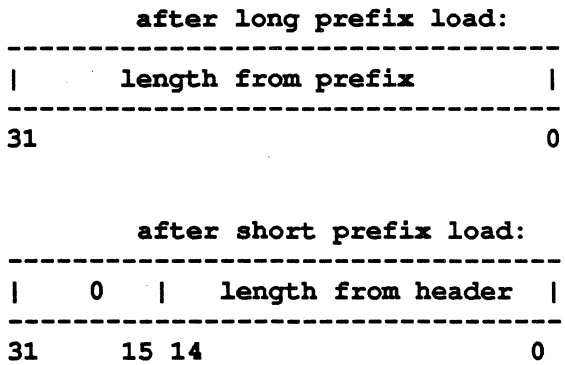
The Array Header Register contains the element type bits 31:30, byte pack bits 29:27; and long prefix bit 23 of the array header stored in memory. In addition, it contains a long prefix trap inhibit bit and byte offset bits. These read as 0 for a short prefix header and reflect bits<12,11:7> of a long prefix header encached as an array descriptor. These spare long prefix header bits must be set to 0 in the actual memory header location.

```
-------------------------------------------------
|e-type|byte-pack|xxxx|lp|xxxx|lpi|byte-offset|xxxxxx|
-------------------------------------------------
```


IMPORTANT

1.10.3 Array Length Register

The Array Length Register contains the index limit of the array. When the Array Header Register is loaded from memory, the Array Length Register automatically loads as well. Array length loads with the bottom 15 bits of the array header data read into the processor, if header indicates a short prefix; it loads with the second word of the array prefix, if header indicates a long prefix.This means that all non-trapping long prefix array headers, i.e., those that are stored in array descriptors, must align on even data words.



1.10.4 Array Descriptors

To provide a mechanism for referencing long-prefix arrays in a loop without trapping, Sunstone provides a method for encaching array information similar, but not identical, to the I-Machine array registers. As in the I-Machine, the software sets up the array descriptors outside the array reference loop, for use by generic inner loop code. However, instead of placing the array descriptors in the window buffer or on the data stack, Sunstone keeps a special array descriptor stack with a pointer kept in global registers; this allows for an infinite number of arrays to be optimized in this fashion. The generic inner loop code loads the array-header and array-length from the array-register section of memory using the locally stored array-register pointer.

During setup, the software sets an inhibit bit in the encached version of a long-prefix array header, so that long prefix arrays do not trap repeatedly. It also stores the byte offset of displaced arrays in other spare bits of the encached Array Header; non-displaced arrays contain a byte-offset of 0.

The array descriptor requires the following array-registers for each encached array:

array-header
array-length
array-object-reference
array-address-reg-VMA

A routine that adjusts an array needs to check to see if the array that its adjusting has encached array descriptors, or an array indirecting to it has encached array registers. Several different methods can be used to make this determination depending on the amount of searching this software wants to do. If encached, the array adjustment must redecode the array for array descriptor setup. The setup code will

use the array-address-reg-VMA to find the array address window register, and then reload the correct array-address into it. To avoid inconsistencies caused by array adjusting while in the middle of a reference, the code in the inner loop must inhibit preempts.

This is an example of what array references will look like, using array descriptors. Array-header is at array-descriptor-sp, and array-length is at array-descriptor-sp, #+1. Array-descriptor-sp, contains the base array-register address in a local register. The array-address local register will contain a locative to first element of the array minus one.

```
*load-data    array-header-register ← [array-descriptor-sp,]
                [array-length also loaded and no-preempt set]
load-array   result ← [array-address-reg+index]
```

1.10.5 Trap Conditions

Up to four conditions may cause an array trap during *load-array* or *store-array* instructions. Listed in order of precedence:

1. array element (see *store-array* section 1.5.11 on page 93).
2. prefix long (Array-Header bit<23> when not inhibited by Array-Header bit<12>)
3. index out of range (when index ≥ Array-Length[unsigned])
4. packed array (Array-Header bits<29:27> ≠ 0)

The array element trap only occurs on *store-array* and may fixup the data type of the value to be stored, or error. The prefix long trap must decode the array to find the appropriate array address. The index out of range is an error trap. The packed array trap involves special hardware to fixup the index and adjusts the byte rotate register to allow fast packed array trap routines.

When an array traps for packed array reasons, the A2 register trap argument expected by the trap routines contains an adjusted index. This index gives the real offset for the memory location index of the referenced array. In addition, the Byte Rotate Register contains the appropriate byte specifier used for ldb'ing during loads and dpb'ing during stores. This byte specifier refers to the n-bit byte specified by the low (log n 2) bits of index, where n represents the array byte-size indicated by the array-header. Additionally, the ldb'ing for a load occurs instead of a move during the subversion of the original *load-array* instruction.

Here's an example packed trap routine for load-array:

```
;;; on entry hardware has set up the following registers:

;;; A0 - SCR in normal trap operation
;;; A1 - array address (S1) as in normal trap operation
;;; A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
;;;           where byte-spec = (byte 32. Array-Header-Register<29:27>) and
;;;           byte-offset forced to 0 for short prefix arrays

load-data       trap-result ← [A1,A2]
return-subvert  SCR-24 ← A0
[when re-executing load-array, the subvert ldb's trap-result]
```

For store-array:

```
;;; A0 - SCR as in normal operation
;;; A1 - data to store
;;; A2 ← adjusted index = (1+ (ldb byte-spec (+ index byte-offset)))
;;;           where byte-spec = (byte 32. Array-Header-Register<29:27>) and
;;;           byte-offset forced to 0 for short prefix arrays
;;; mar - array
;;; the array address is in the MAR register because store-array is always
;;; preceded by a load-header, which leaves the real forwarded address in the
;;; MAR.


load-data       a3 ← [mar,a2]
dpb             a3 ← a1,a3      ; Byte-Rotate set up by trap
store-data      [mar,a2] ← a3
return-subvert  SCR-24 ← a0
```

1.11 STORE CONDITIONAL

In order to allow interprocess and interprocessor locking it must be possible to atomically modify a memory location. This operation is performed by the store-conditional instruction on the I-machine and by the following sequence of instructions on Sunstone:

```
      * load-data       temp ← [store-cond-address]
     !* write-mcreg     IBUS-Lock ← t
     !* load-raw        temp ← [MAR]
     !* branch-next     neq, temp, expected-data, @fail
     !* store-raw       [MAR] ← store-cond-data
        write-mc-reg    IBUS-Lock ← nil
        return          R15 ← t
fail:   write-mc-reg    IBUS-Lock ← nil
        return          R15 ← nil
```

In this sequence, the ! indicates that the inhibit-interrupt bit is set in the instruction; the * indicates that the inhibit-preempt bit is set.

The first *load-data* instruction is used to force any transport traps and/or invisible pointer following; once any traps have completed, the MAR contains the actual address of the data, and preempts are inhibited so it will not be possible for a GC flip to occur. The *write-mcreg* will turn on the IBUS-Lock bit, so that the following load-raw will not only read the data word into temp but will also lock the IBUS for the referenced address. The branch-next then checks if the location had the desired value; if it did, the new value is stored and T is returned after the IBUS-lock is turned off; otherwise, the IBUS-Lock is turned off and NIL is returned.

1.12 INSTRUCTION RESTRICTIONS

The hardware implementation forces the software to follow rules regarding the actual use of instructions specified by the architecture. Most of the restrictions that make these rules pertain to unallowable instruction sequences. A table at the end summarizes the sequence restrictions.

1.12.1 Load Instruction Restrictions

An instruction cannot immediately follow any load instruction, when it uses the destination of the load instruction as one of its sources. For any load instruction except *load-ephemeralp* or *load-oldspacep*, software must insert at least one other instruction after the load. After a *load-ephemeralp* or *load-oldspacep*, it must insert at least two other instructions. As an exception to this rule, the data for a store instruction can come from the register used as the destination of the previous load instruction.

An instruction that uses the SCR register after a load instruction has restrictions as listed below under SCR restrictions.

1.12.2 Special Register Restrictions

An instruction cannot use the following special registers for source 2: array-length, byte-rotate, status-control (SCR).

A store RR instruction cannot use a special register for source 1.

Software which writes non-zero values to unimplemented bit positions of special registers must not assume that the values returned by those registers for the next two instructions will be consistent. Because of the passaround paths provided in the processor, the unimplemented positions may return the values written if accessed in the next two cycles.

In addition, the processor restricts the use of instruction sequences using special registers as indicated below.

MAR - Cannot immediately follow an instruction that explicitly writes the MAR (ie, uses the MAR as a destination) with a MAR side-effecting instruction, specifically a *load-car-cdr*, *load-cdr*, *load-header*, or *load-structure* instruction, or a store RR instruction. The software must insert at least one other

instruction before the MAR side-effecting instruction, and at least three other instructions before a store RR.

N-Args - Cannot immediately follow an instruction that explicitly writes N-Args (ie, uses the N-Args as a destination), with an N-Args side-effecting instruction, specifically a call, return, or return-subvert instruction. The software must insert at least one other instruction before the N-Args side-effecting instruction.

SCR - Cannot immediately follow a load instruction with an instruction that uses the SCR as a source. The software must insert at least three other instructions before using the SCR.

1.12.3 MC Register Restrictions

An instruction cannot immediately follow any *read-mc-reg* instruction, when it uses the destination of the *read-mc-reg* instruction as one of its sources. Software must insert at least one other instruction after the *read-mc-reg*.

The software cannot follow a *write-mc-reg* instruction with a *read-mc-reg* instruction that references the mc register just written. Software must insert at least one other instruction before using the *read-mc-reg* instruction.

WBC - Cannot immediately follow an instruction that explicitly writes the WBC (ie, using the *write-mc-reg* instruction) with an instruction that uses a non-global or non-special register as a source. Since the processor requires two cycles before the desired change in window status takes place, software must insert at least two other instructions before an instruction can use the current or build window registers.

Return Address - Cannot immediately follow an instruction that explicitly writes the Return Address (ie, using the *write-mc-reg* instruction) with an instruction that implicitly uses the Return Address, specifically a *return* or *return-subvert* instruction. Software must insert at least two other instructions before using the *return* or *return-subvert*.

1.12.4 Instruction Sequence Restriction Table

The number in the table represents the number of unrelated instructions to insert.

Table 16. Instruction Sequence Restrictions

	AFTER THESE INSTRUCTIONS							
NEXT INSTRUCTION CONDITIONS	gc* loads	other** loads	dest. MAR	dest. N-Args	read-mc-reg	write-mc-reg	write WBC***	write RA****
using destination	2	1	-	-	1	-	-	-
using SCR	3	3	-	-	-	-	-	-
MAR side-effect load	-	-	1	-	-	-	-	-
store RR	-	-	3	-	-	-	-	-
call	-	-	-	1	-	-	-	-
return,return-subvert	-	-	-	1	-	-	-	2
read-mc-reg(same reg)	-	-	-	-	-	1	-	-
using window regs	-	-	-	-	-	-	2	-

* load-oldspacep or load-ephemeralp
** any load except for load-oldspacep and load-ephemeralp
*** write-mc-reg WBC
**** write-mc-reg Return Address

Appendix A
TABLE OF INSTRUCTION SIDE EFFECTS

Table 17. Instructions/Conditions Side Effecting Special Registers

Instructions/Conditions Side Effecting Special Registers		
Instruction and/or Condition	Register	Descriptions
call, jcall RIL format	PC	loads from 38-bit-immediate
return, return-subvert	PC	loads from Return Address
any branch	PC	if true, load PC offset from instruction
call, jcall	SCR - Complete	if bit 9 set, clear and set bit 12
return, return-subvert	SCR - Complete	if bit 10 set, clear and set bit 12
return-subvert	SCR - Subvert	sets subvert bit 8
any load	SCR - Cdr,Type	load cdr and type from memory
any trap or interrupt	SCR	clears bits 12:8 and ORs instruction bits 39:38 into 16:17
any interrupt	SCR - Interrupt	sets interrupt level
any completed instruction	SCR - Complete	if bit 11 set, clear and set bit 12
call, jcall, return, return-subvert	N-Args	may load from n-args inst. field
load-array	Byte-Rotate	loads according to packing information
load-cdr,load-car-cdr, load-header,load-structure	MAR	loads from calculated address
any load w/ dest. Array-Header	Array-Length	loads from Array-Header or SBUS word

Index

12-bit Signed-Immediate 25

6-bit-signed-immediate 22

Arithmetic Operations 33
Arithmetic Type Traps 122
Array descriptors 137
Array Hardware Support 135
Array Header Register 5, 136
Array Length Register 5, 137
Array References 134

Binding Stack 115
Bit and Byte Operations 43
Branch Type Traps 123
Build Window 3
Byte Rotate Register 5

Cache Control Register 12
Call Operations 49
Calling 116
Catch Block Pointer 115
Cdr Reg 7
Compiled functions 1
Cond 23
Condition 25
Conditional Operations 61
Coproc-Opcode 24
Current Window 3

Data Block Pointer 114
Data Stack Pointer 114
Data Stack 113
Data Type Checking 26
Data Type Setting 27
Data types 1, 21
  I-Machine differences 1
Destination 22
Diagnostic Register 14
Direct Branch Format 19
Direct Branch Operation 59

ECC Log Counter 14
Ephemeral Oldspace Register 11
Exceptions 120

FPU Configuration 11
Function Calling 116
Function Entry 118

Garbage Collection 134
General Traps 125
Generic Functions 118
Global Registers 8

High/Low 25

IBUS Lock 14
IBUS-Error-Address 14
IBUS-Error-Status 14
Illegal Instruction Trap 123
Inhibit Interrupt 8
Inhibit Interrupts 20
Inhibit Preempt 8
Inhibit Preemption 20
Instruction Format 15, 20
Instruction Sequencing 19
Instructions
  Add 34
  Add-no-overflow 35
  And 40
  Ash 44
  Branch 60
  Branch-next 63
  Branch-next-type 67
  Branch-take 64
  Branch-take-type 68
  Call 50
  Dpb 48
  Fields of 20
  Formats of 14
  Jcall 51
  Ldb 47
  Load-array 78
  Load-bind 83
  Load-car-cdr 74
  Load-cdr 80
  Load-cdr-finish 76
  Load-Coproc 105
  Load-data 72
  Load-data-iv 73
  Load-ephemeralp 88
  Load-gc-copy 85
  Load-header 82
  Load-oldspacep 87
  Load-raw 86
  Load-scavenge 84
  Load-structure 81
  Lsh 45
  Move 57
  Move-type 58
  Mult 38
  Or 41
  Read-Coproc 107
  Read-mc-reg 110
  Return 53
  Return-subvert 54
  Rot 46
  Store-38-bits 102
  Store-array 93
  Store-bind 95
  Store-cdr-3 99
  Store-cdr-next 97
  Store-cdr-nil 96
  Store-cdr-normal 98
  Store-cdr-reg 100
  Store-Coproc 106
  Store-data 90
  Store-data-iv 91
  Store-rplacd 92

Store-type-reg 101
Sub 36
Sub-no-overflow 37
Trap 65
Trap-type 69
Write-Coproc 104
Write-mc-reg 111
Xor 42
Internal Registers 4
Interrupt Level 8
Interrupt Registers 13
Interrupts 121

Load operations 70
Load Traps 123
Logical Operations 39

Map Cache Validbit 14
Map-Cache 13
Memory Address Register 6
Memory Control Registers 9
Memory Error Status Register 11
Message Passing 119
Metering Counter 12
Metering Modes 12
Microsecond Clock 11
Move Operations 56

N-Args 25
NIL 6
Number of Args Register 6

Opcode 21
Opcode Specific Traps 123
Opcodes 21
Overlapping Window Scheme 3

Page Offset 25
PC Address 21
PHT Hashbox Register 11
Physical address space 2
Program Counter 6

Register Immediate Long (RIL) Format 17
Register Immediate Short (RIS) Format 17
Register to Register (RR) Format 15
Registers 3
Return 120
Return Address Register 11
Return Operations 52

Slot Number Register 14
Source1 21
Source2 22
Special Registers 4
Stack Groups 112
Status Control 24 6
Status Control Register 6
Store Operations 89
Store Traps 123
Subvert Instruction 7

T 6

Table of instructions 30
Take Instruction Completion Trap 7
Timer 11
Trap Base Register 11
Trap on Call 7
Trap on Instruction Completion 7
Trap on Return 7
Trap Result Register 6
Trap Vector 126
Traps 121
Type Check 22
Type Reg 7
Type Traps 123

Virtual memory 2

Window Stack Limit 112
Window Buffer Control Register 11
Window Registers 3
Window Stack 112
Window Stack Base 112
Window Stack Pointer 112

Zero 6
Zone Oldspace Register 11