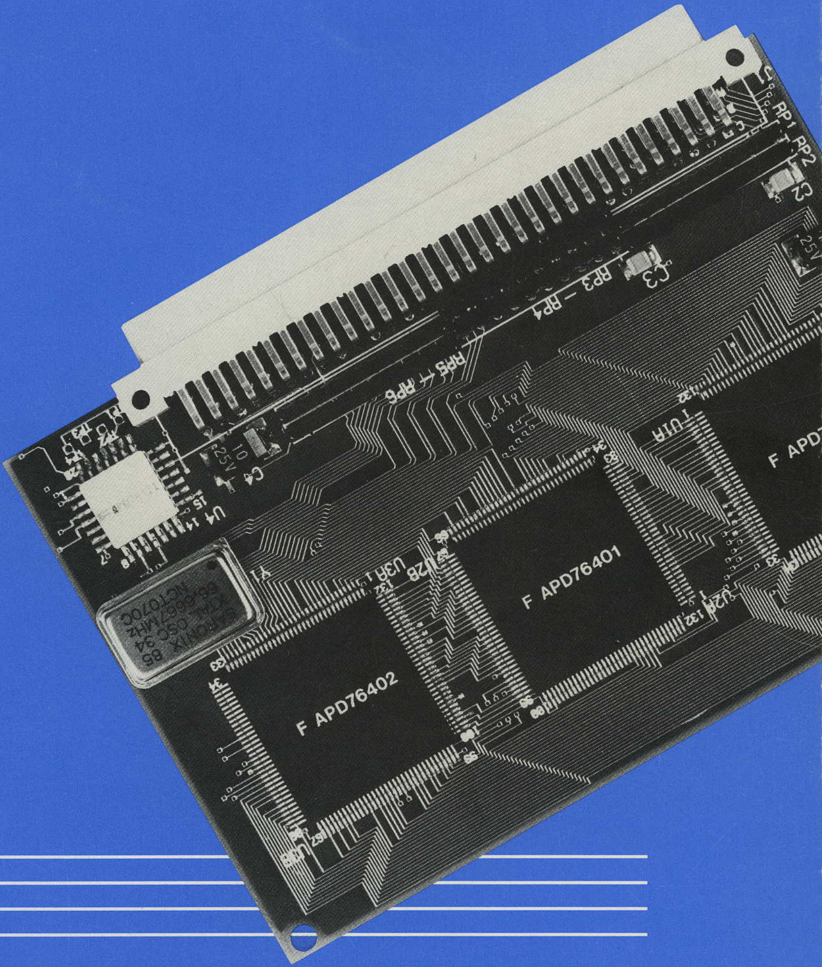
FAIRCHILD

A Schlumberger Company

CLIPPER™ 32-Bit Microprocessor Module

INSTRUCTION SET



CLIPPER™ 32-Bit Microprocessor Module

INSTRUCTION SET

ADVANCE
INFORMATION

CLIPPER™ 32-BIT MICROPROCESSOR MODULE INSTRUCTION SET

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

PREFACE

This document describes the CLIPPER instruction set. Component information (e.g., internal register description, programming model, exception processing, bus operation, etc.) is available in the CLIPPER Module Product Description.

CHAPTER ONE
INSTRUCTION FORMATS

This chapter discusses:
- **Instruction formats without addresses**
- **Instruction formats with addresses**

1.1 INTRODUCTION

The information encoded in each instruction specifies the operation to be performed, the type and number of operands to use, and the location of the operands. These operands can be located in a register or in memory. For example, the **loadb** instruction contains operands that reference memory and a register. If an operand is located in memory, the instruction must calculate the address of the operand according to the address mode specified in the instruction format.

The immediate and quick instructions use an operand encoded within the instruction for fast efficient operation.

All instructions are constructed in multiples of halfwords called parcels (see the general instruction format below). The size of instructions varies from one to four parcels.

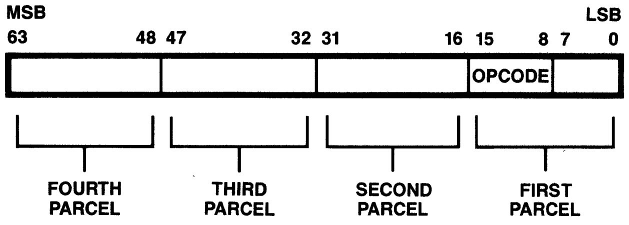


Figure 1-1 shows the instruction formats used in the CLIPPER architecture. Notice that the formats are divided into two main categories, non-memory referencing instructions (NO ADDRESS) and memory referencing instructions (WITH ADDRESS).

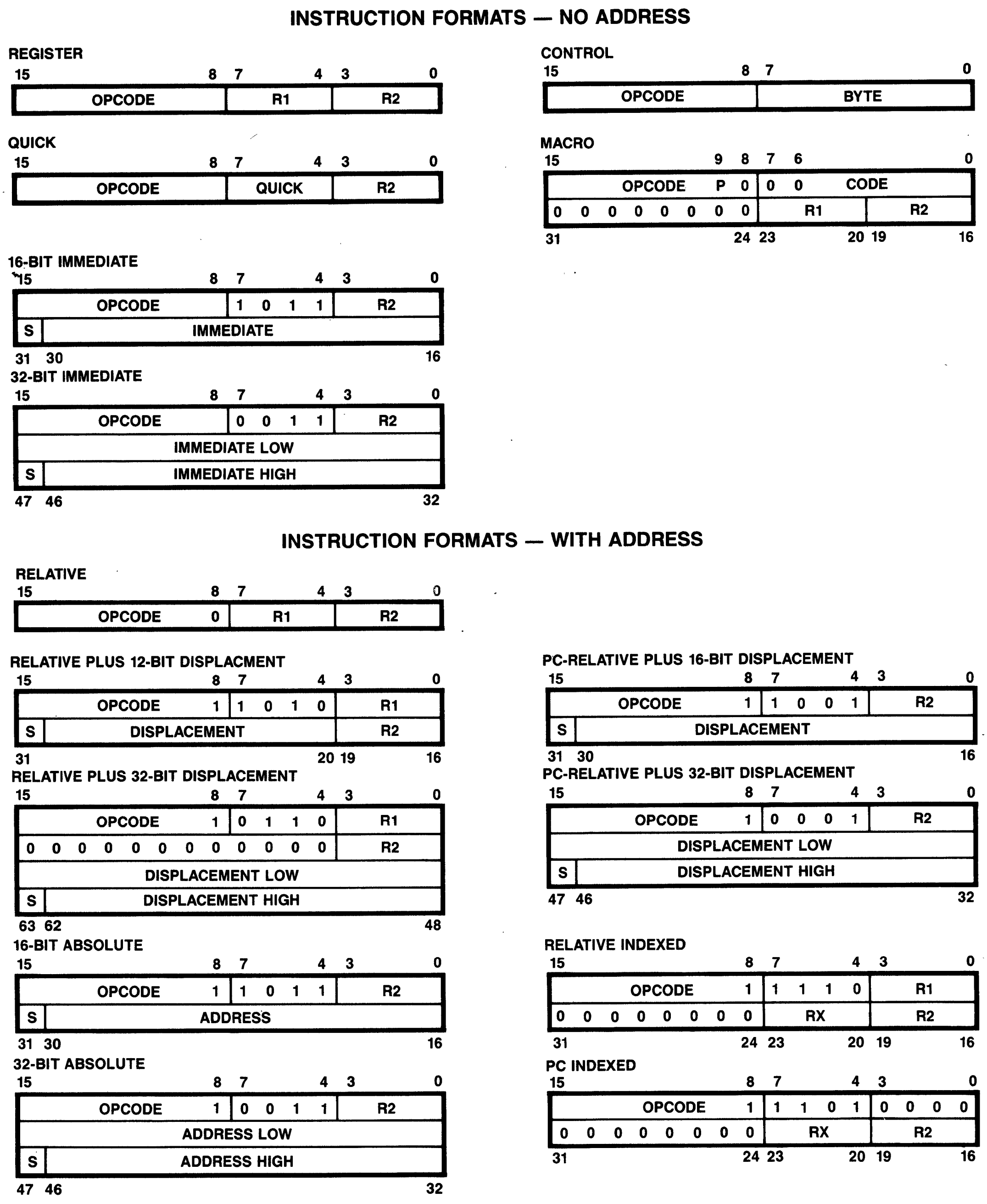
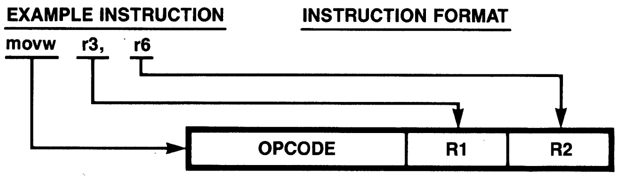


Figure 1-1 Instruction Formats

1.2 INSTRUCTION FORMATS — NO ADDRESS

1.2.1 REGISTER

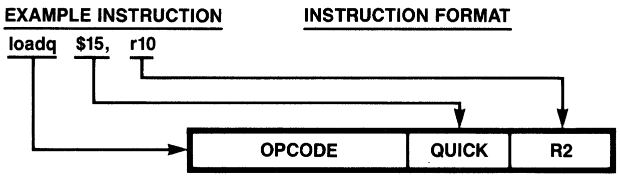
The Register format is used for most instructions that take just one or two register arguments.



The opcode specifies the interpretation of the R1 and R2 fields. Usually the R1 field contains the source operand register number, and R2 contains the destination operand register number. For example, in the **movsw** instruction, the R1 field contains the number of the single-precision floating-point register containing the source operand, and the R2 field contains the number of the general register in which to store the result.

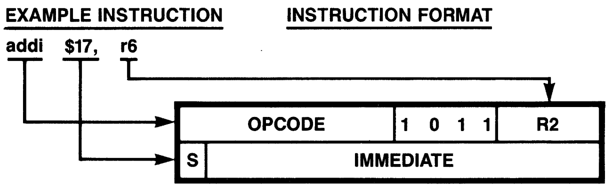
1.2.2 QUICK

The Quick format encodes constant, 4-bit unsigned source operands directly in the instruction. The quick value is always zero-filled at the left before use.



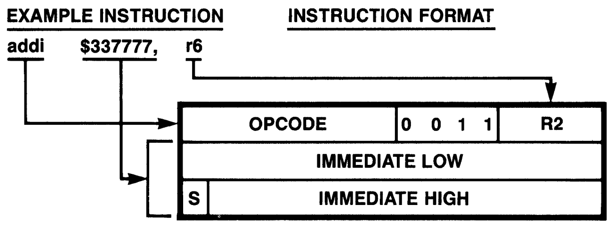
1.2.3 16-BIT IMMEDIATE

The 16-bit Immediate format encodes a 16-bit source operand constant directly in the instruction. The immediate value is always sign-extended before use.



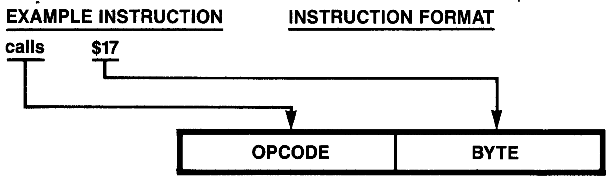
1.2.4 32-BIT IMMEDIATE

The 32-bit Immediate format encodes a constant, 32-bit source operand directly in the instruction.



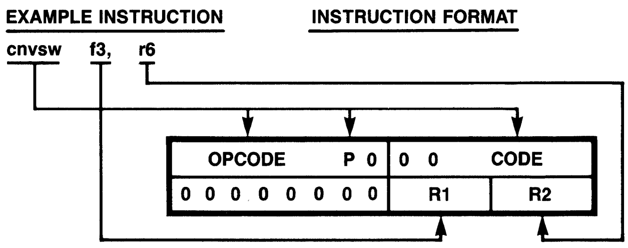
1.2.5 CONTROL

The Control format encodes up to 8 bits of a constant value that is used by a few special instructions. For example, the byte operand specifies the system call number in the **calls** instruction.



1.2.6 MACRO

The Macro format is used by those instructions that are implemented in MI ROM rather than directly in the hardware. The P bit in the opcode, bit 9 of the format instruction parcel, selects a privileged macro.



1.3 INSTRUCTION FORMATS — WITH ADDRESS

The rest of the instruction formats specify an address operand and a register operand. Several address formats, or modes, are provided to support typical high-level language operations. The address mode is selected first by the opcode (bit 8 of the first instruction parcel), and if necessary, by the AM field (bits 7:4 of the first instruction parcel). Displacements and absolute addresses are always sign extended.

The address modes used in the memory referencing instructions are summarized in Table 1-1 and explained on the following pages.

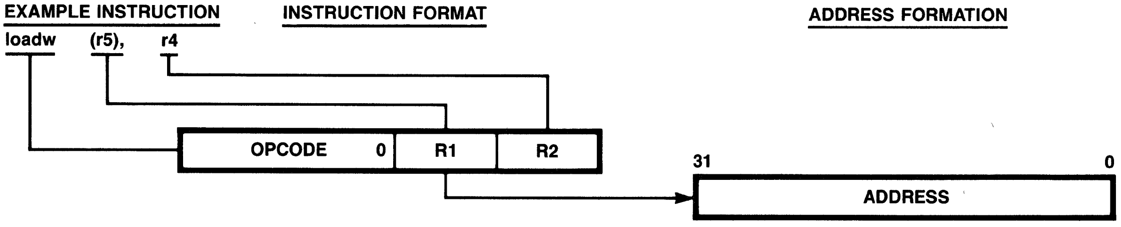
Table 1-1 Memory Addressing Modes

Memory Addressing Mode	Address Formation	Page
Relative	Address ← (R1)	1-7
Relative with 12-bit displacement	Address ← (R1) + 12-bit displacement	1-7
Relative with 32-bit displacement	Address ← (R1) + 32-bit displacement	1-8
16-bit Absolute	Address ← 16-bit displacement	1-8
32-bit Absolute	Address ← 32-bit displacement	1-9
PC Relative with 16-bit displacement	Address ← (PC) + 16-bit displacement	1-9
PC Relative with 32-bit displacement	Address ← (PC) + 32-bit displacement	1-10
Relative Indexed	Address ← (R1) + (RX)	1-10
PC Indexed	Address ← (PC) + (RX)	1-11

Notes:
All displacements are signed.
PC addresses the first parcel of the current instruction.
RX is any general register containing the index modifying the effect of the source register.

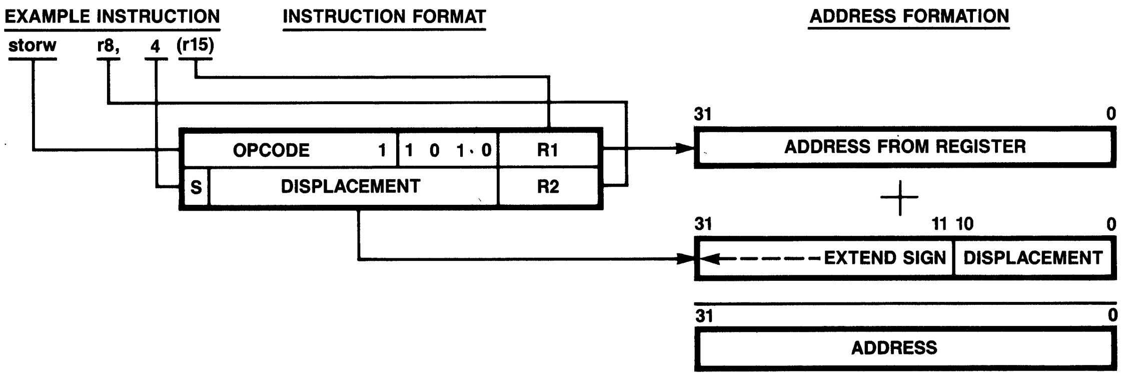
1.3.1 RELATIVE

The Relative format uses the address in a register (R1) to compute an address.



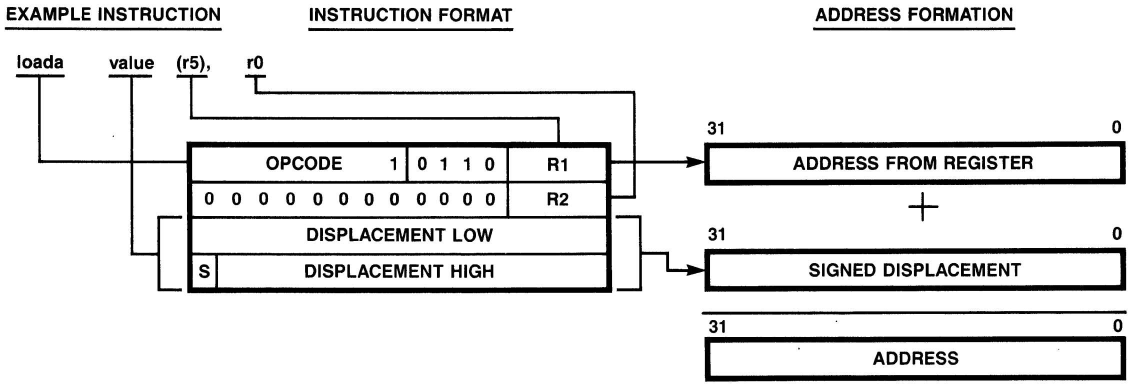
1.3.2 RELATIVE WITH 12-BIT DISPLACEMENT

The Relative Plus 12-bit Displacement format uses the address in a register (R1), plus a signed 12-bit displacement, to compute an address. The displacement is sign-extended to 32 bits before the address calculation.



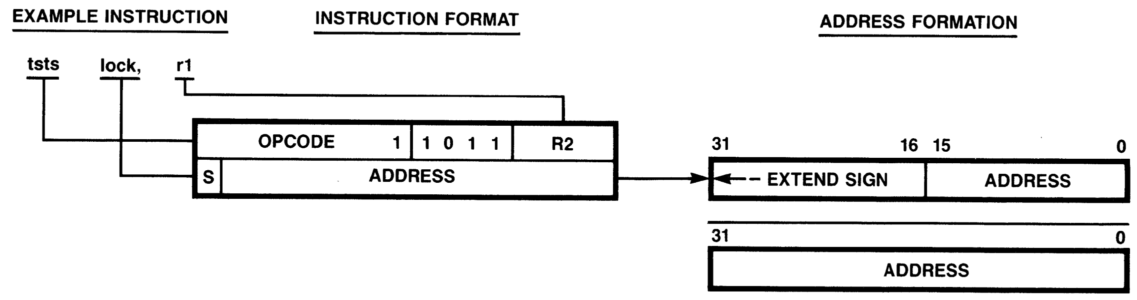
1.3.3 RELATIVE WITH 32-BIT DISPLACEMENT

The Relative Plus 32-bit Displacement format uses the address in a register (R1), plus a signed 32-bit displacement, to compute an address.



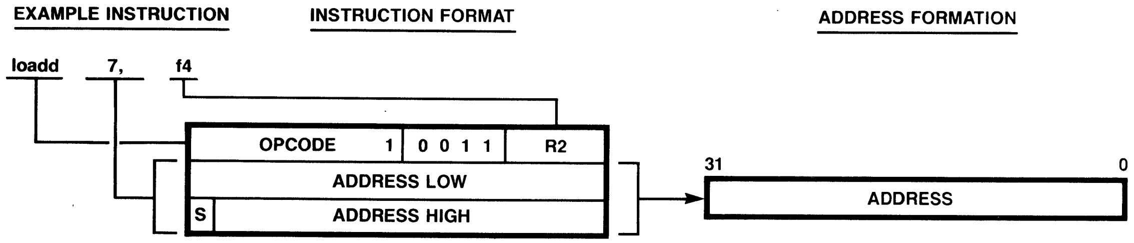
1.3.4 16-BIT ABSOLUTE

The 16-bit Absolute format uses the address in register (R2). The address is sign-extended to 32 bits before being used. Because the address field is signed, the range of address that can be accessed with this format is: 0xffff8000 ≤ address ≤ 0xffffffff.



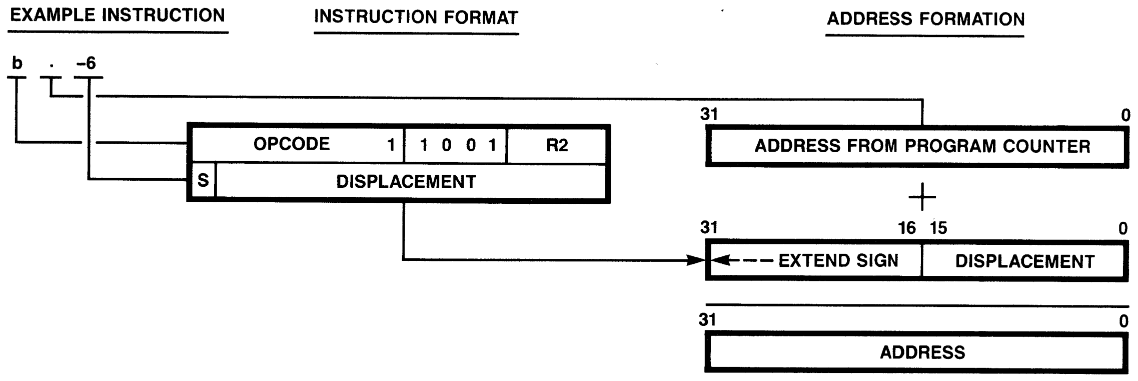
1.3.5 32-BIT ABSOLUTE

The 32-bit Absolute format uses the signed 32-bit displacement portion of the instruction as an address.



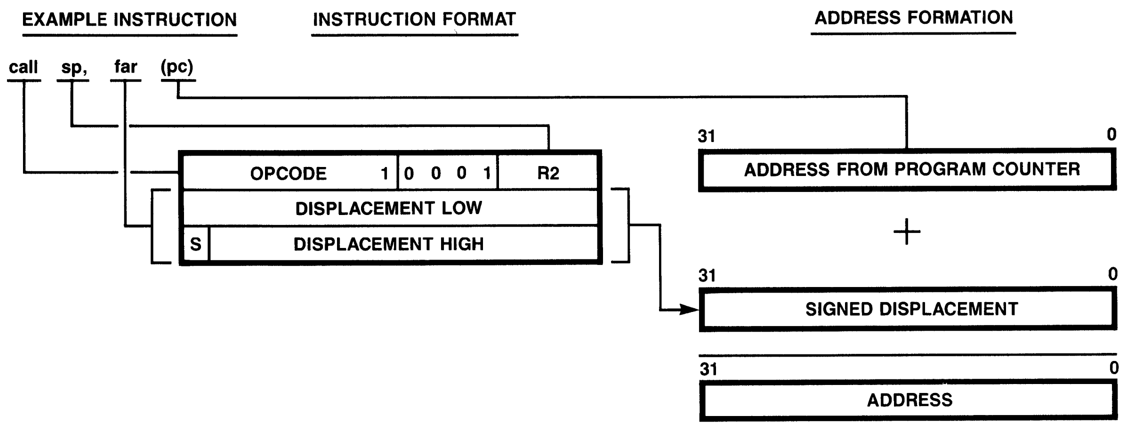
1.3.6 PC RELATIVE WITH 16-BIT DISPLACEMENT

The 16-bit PC Relative format adds a signed 16-bit displacement to the contents of the Program Counter (PC) to compute an address.



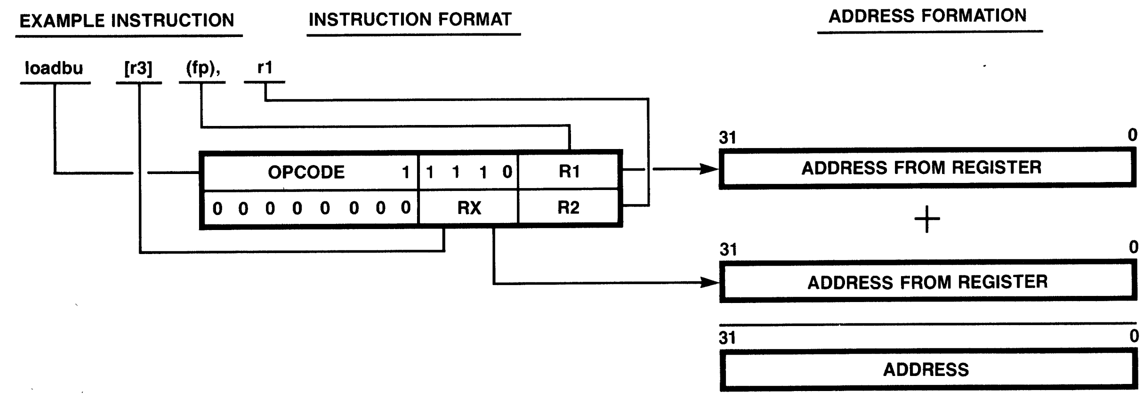
1.3.7 PC RELATIVE WITH 32-BIT DISPLACEMENT

The 32-bit PC Relative format adds a signed 32-bit displacement to the contents of the Program Counter (PC) to compute the address.



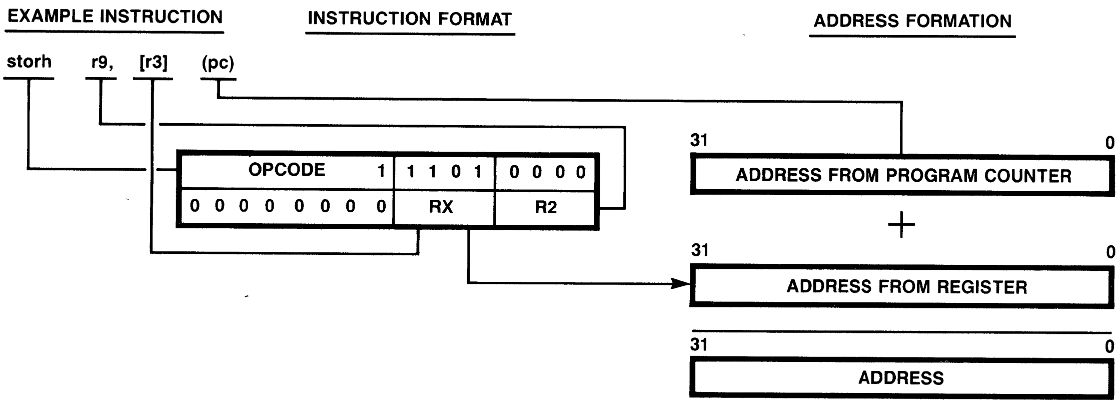
1.3.8 RELATIVE INDEXED

The Relative Indexed format uses the address in a register (R1), plus the contents of an index register (RX), to compute an address.



1.3.9 PC INDEXED

The PC Indexed format adds the contents of an index register (RX) to the contents of the PC to compute an address.



CHAPTER TWO
INSTRUCTION SET

This chapter contains detailed descriptions of the CLIPPER instructions. The instructions are listed in alphabetical order.

The CLIPPER instruction set contains 101 basic instructions and 67 macro instructions. This reduced instruction set is especially useful to high-level language optimizing compilers.

Memory access in a CLIPPER environment is through load/store instructions to minimize bus traffic and memory-dependent execution delays. Data operations are performed in registers.

There are two units in the CLIPPER CPU that execute instructions: the Integer Execution Unit (IEU) and the Floating-Point Execution Unit (FPU). The integer instructions (with the exception of integer multiplies and divides) are executed by the IEU. Floating-point instructions (and the integer multiplies and divides) are executed by the FPU.

The basic instructions are fetched from main memory, through the instruction cache, decoded and then executed, either by the IEU or by the FPU. The macro instructions are executed from the Macro Instruction ROM (MI ROM).

A macro instruction opcode is actually a reference to a sequence of instructions in the MI ROM. When a macro instruction is fetched, execution control is switched to the MI ROM and the macrocoded sequence of the macro instruction is executed.

The instructions are listed in Table 2-1.

Table 2-1 Alphabetical Listing of the Instructions

Mnemonic	Description	Page	Mnemonic	Description	Page
addd	Add Double Floating	2-7	movwp	Move Word to Processor Register	2-64
addi	Add Immediate	2-8	movws	Move Word to Single Floating	2-65
addq	Add Quick	2-9	muld	Multiply Double Floating	2-66
adds	Add Single Floating	2-10	muls	Multiply Single Floating	2-67
addw	Add Word	2-11	mulw	Multiply Word	2-68
addwc	Add Word with Carry	2-12	mulwu	Multiply Word Unsigned	2-69
andi	And Immediate	2-13	mulwux	Multiply Word Unsigned Extended	2-70
andw	And Word	2-14	mulwx	Multiply Word Extended	2-71
b*	Branch on Condition	2-15	negd	Negate Double Floating	2-72
bf*	Branch on Floating Exception	2-17	negs	Negate Single Floating	2-73
call	Call Subroutine	2-18	negw	Negate Word	2-74
calls	Call Supervisor	2-19	noop	No Operation	2-75
cmpc	Compare Characters	2-20	notq	Not Quick	2-76
cmpd	Compare Double Floating	2-21	notw	Not Word	2-77
cmpi	Compare Immediate	2-22	ori	Or Immediate	2-78
cmpq	Compare Quick	2-23	orw	Or Word	2-79
cmps	Compare Single Floating	2-24	popw	Pop Word	2-80
cmpw	Compare Word	2-25	pushw	Push Word	2-81
cnvds	Convert Double to Single Floating	2-26	restd*n*	Restore Registers f*n*: $0 \leq n \leq 7$	2-82
cnvdw	Convert Double to Word	2-27	restur	Restore User Registers	2-83
cnvrdw	Convert Rounding Double to Word	2-28	restw*n*	Restore Registers r*n*: $0 \leq n \leq 12$	2-84
cnvrsw	Convert Rounding Single to Word	2-29	ret	Return From Subroutine	2-85
cnvsd	Convert Single to Double Floating	2-30	reti	Return From Interrupt	2-86
cnvsw	Convert Single Floating to Word	2-31	roti	Rotate Immediate	2-87
cnvtdw	Convert Truncating Double to Word	2-32	rotl	Rotate Longword	2-88
cnvtsw	Convert Truncating Single to Word	2-33	rotli	Rotate Long Immediate	2-89
cnvwd	Convert Word to Double Floating	2-34	rotw	Rotate Word	2-90
cnvws	Convert Word to Single Floating	2-35	saved*n*	Save Registers f*n*: $0 \leq n \leq 7$	2-91
divd	Divide Double Floating	2-36	saveur	Save User Registers	2-92
divs	Divide Single Floating	2-37	savew*n*	Save Registers r*n*: $0 \leq n \leq 12$	2-93
divw	Divide Word	2-38	scalbd	Scale by Double Floating	2-94
divwu	Divide Word Unsigned	2-39	scalbs	Scale by Single Floating	2-95
initc	Initialize Characters	2-40	shai	Shift Arithmetic Immediate	2-96
loada	Load Address	2-41	shal	Shift Arithmetic Longword	2-97
loadb	Load Byte	2-42	shali	Shift Arithmetic Longword immediate	2-98
loadbu	Load Byte Unsigned	2-43	shaw	Shift Arithmetic Word	2-99
loadd	Load Double Floating	2-44	shli	Shift Logical Immediate	2-100
loadfs	Load Floating Status	2-45	shll	Shift Logical Longword	2-101
loadh	Load Halfword	2-46	shlli	Shift Logical Longword Immediate	2-102
loadhu	Load Halfword Unsigned	2-47	shlw	Shift Logical Word	2-103
loadi	Load Immediate	2-48	storb	Store Byte	2-104
loadq	Load Quick	2-49	stord	Store Double Floating	2-105
loads	Load Single Floating	2-50	storh	Store Halfword	2-106
loadw	Load Word	2-51	stors	Store Single Floating	2-107
modw	Modulus Word	2-52	storw	Store Word	2-108
modwu	Modulus Word Unsigned	2-53	subd	Subtract Double Floating	2-109
movc	Move Characters	2-54	subi	Subtract Immediate	2-110
movd	Move Double Floating	2-55	subq	Subtract Quick	2-111
movdl	Move Double Floating to Longword	2-56	subs	Subtract Single Floating	2-112
movld	Move Longword to Double	2-57	subw	Subtract Word	2-113
movpw	Move Processor Register to Word	2-58	subwc	Subtract Word with Carry	2-114
movs	Move Single Floating	2-59	trapfn	Trap On Floating Unordered	2-115
movsu	Move Supervisor to User	2-60	tsts	Test and Set	2-116
movsw	Move Single to Word	2-61	wait	Wait for Interrupt	2-117
movus	Move User to Supervisor	2-62	xori	Exclusive-OR Immediate	2-118
movw	Move Word	2-63	xorw	Exclusive-OR Word	2-119

The format of each instruction is described in detail in the following pages. Figure 2-1 below illustrates the information presented.

Instruction Name — mnemonic and descriptive name of the instruction.

Assembler Syntax — Bold means enter item exactly as shown. *Italics* means substitute correct value. Punctuation must be included as shown. See Table 2-2 for operands.

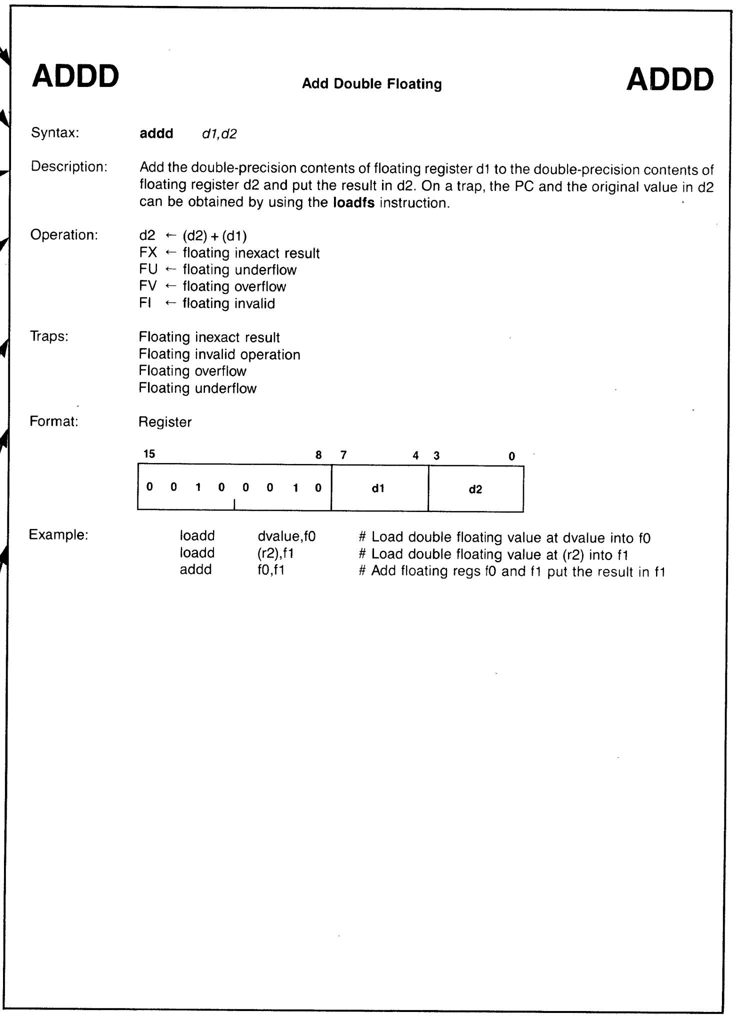
Description — the text that describes the operation of the instruction, functionally equivalent to Operation.

Operation — this is an equation describing the operation of the instruction. Tables 2-3 and 2-4 list operands used. When setting psw flags, the value of the expression to the right of the "←" replaces the flag.

Exceptions — conitions under which an exception may occur. Many traps have trap enables in the PSW or SSW. See Chapter 5 for details.

Insruction Format — the class, bit patterns, and fields of the instruction. See Table 2-5 for operand and operator descriptions.

Instruction Example — an example of the source statement entered. See Table 2-6 for a description of the terms used.



ADDD **Add Double Floating** **ADDD**

Syntax: **addd** *d1,d2*

Description: Add the double-precision contents of floating register d1 to the double-precision contents of floating register d2 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
d2 ← (d2) + (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15							8	7 … 4	3 … 0
0	0	1	0	0	0	1	0	d1	d2

Example:
```
loadd   dvalue,f0    # Load double floating value at dvalue into f0
loadd   (r2),f1      # Load double floating value at (r2) into f1
addd    f0,f1        # Add floating regs f0 and f1 put the result in f1
```

Figure 2-1 Instruction Description Example

Table 2-2 lists meanings of the operands used in the Syntax field of the instruction descriptions. The first character of the notation column specifies the operand's type and size:

b = byte	w = word	s = single-precision floating-point
h = halfword	l = longword	d = double-precision floating-point
		p = processor register

The second character of the notation column specifies the operand's field within the instruction and its location in the machine (immediate value, register, memory, etc.):

a = R1	q = quick	a = address
2 = R2	i = immediate	b = byte

Table 2-2 Syntax Field Operands

Notation	Meaning
ba	Byte address (32-bit address on byte boundary)
bb	Unsigned byte data (0 — 256 decimal)
da	Double floating address (32-bit address on doubleword boundary)
d1	Double floating register operand 1 (f0 — f7)
d2	Double floating register operand 2 (f0 — f7)
ha	Halfword address (32 bit address on halfword boundary)
l1	Longword register operand 1 (even register pair r0,r1 — r14,r15)
l2	Longword register operand 2 (even register pair r0,r1 — r14,r15)
p1	Processor register operand 1 (0 = psw, 1 = ssw)
sa	Single floating address (32-bit address on word boundary)
s1	Single floating register operand 1 (f0 — f7)
s2	Single floating register operand 2 (f0 — f7)
wa	Word address (32 bit address on word boundary)
wi	Signed 16-bit immediate data (sign extended to 32-bits) or signed 32-bit immediate data
wq	Unsigned 4-bit quick data (zero extended to 32-bits)
w1	General register operand 1 (r0 — r15)
w2	General register operand 2 (r0 — r15)

Tables 2-2 and 2-3 list the meanings of the operands used in the Operation field of each description. Table 2-4 lists the meanings of the operators used in the Operation field of each description.

Table 2-3 Operation Field Operands

Notation	Meaning
C	PSW carry out flag
FP dest	Original floating-point destination
FP PC	Floating-point program counter
N	PSW integer negative flag
n	Number (usually register number)
PC	Program counter
r0 — r15	General registers 0 through 15
V	PSW integer overflow flag
Z	PSW integer zero flag

Table 2-4 Operation Field Operators

Notation	Meaning
rot	Rotate operator
sha	Shift arithmetic operator
shl	Shift logical operator
+	Add operator
−	Subtract operator or negate unary operator
×	Multiply operator
÷	Divide operator
mod	Modulus operator
~	Complement operator
=	Equal operator
≠	Not equal operator
←	Assignment operator
&	AND logical operator
|	OR logical operator
⊕	Exclusive-OR logical operator
( )	Contents of operand within
[ ]	Separators used to indicate value inside in a unit
< >	Value inside symbols indicates bit(s) of a register
:	Indicates a range of values

Tables 2-2 and 2-5 list meanings of the operands used in the Format field of the instruction descriptions.

Table 2-5 Format Field Operands

Notation	Meaning
addr mode	A four bit code for the type of addressing used
cond	Condition code for branch conditions (see Tables 2-7 and 2-8)
high	Most-significant portion of an address or value
low	Least-significant portion of an address or value
R1	Register field 1 within the instruction format
R2	Register field 2 within the instruction format

Table 2-6 lists the assembler instruction operands used in the Example field of the instruction descriptions. Assembler instruction operands are generally given in source, destination order, independent of their positions in the machine representation.

Table 2-6 Example Field Operands

r0 : r15	General registers. Even general registers address longword operands. sp, fp, and ap are synonyms for r15, r14, and r13. Not to be confused with R1 or R2, which are register fields within an instruction.
f0 : f7	Floating registers. Each register may contain either a single or double floating value
psw, ssw	Processor registers 0 and 1
$*n*	Quick, byte or immediate value (decimal)
$O × *n*	Quick, byte or immediate valye (hexadecimal)
n	Absolute address
n(m)	Relative or relative with displacement address. *n* may be 0 or absent.
[r*x*](rn)	Relative indexed address
n(pc)	PC relative address
. ± *n*	PC relative address
[r*x*](pc)	PC indexed address
#	Indicates a comment field

ADDD — Add Double Floating — ADDD

Syntax: **addd** *d1,d2*

Description: Add the double-precision contents of floating register d1 to the double-precision contents of floating register d2 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:

```
d2 ← (d2) + (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid
```

Traps:

Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15							8	7 … 4	3 … 0
0	0	1	0	0	0	1	0	d1	d2

Example:

```
loadd    dvalue,f0    # Load double floating value at dvalue into f0
loadd    (r2),f1      # Load double floating value at (r2) into f1
addd     f0,f1        # Add floating regs f0 and f1 and put the result in f1
```

ADDI — Add Immediate — ADDI

Syntax: **addi** *wi,w2*

Description: Add the immediate value wi to the contents of general register w2 and put the result in w2. The operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have the same sign and the result has the opposite sign.

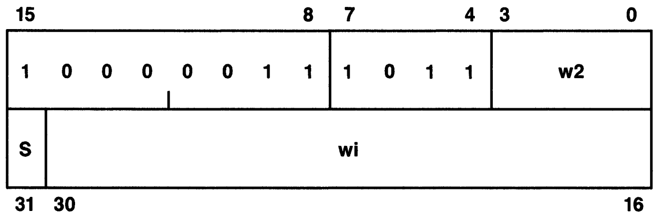
Operation:

$w2 \leftarrow (w2) + wi$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow$ integer overflow
$C \leftarrow$ integer carry out

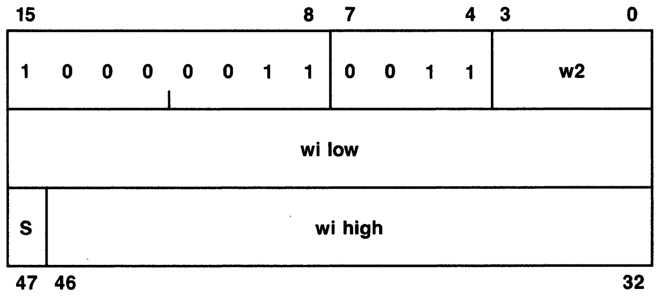
Traps: none

Format: Immediate

If $-2^{15} \le wi \le +2^{15} - 1$, this format is used:



If $+2^{15} \le wi \le +2^{31} - 1$ or $-2^{15} \le wi \le -2^{15} - 1$, this format is used:



Example:

```
addi    $180,r1       # Add 16-bit value to r1
addi    $99999,r0     # Add 32-bit value to r0
addi    $ – 1,r2      # Add 0xffff to r2
```

ADDQ — Add Quick — ADDQ

Syntax: **addq** *wq,w2*

Description: Add the unsigned quick value wq to the contents of general register w2 and put the result in w2. The contents of w2 may be a signed or unsigned integer. Overflow is set if the contents of w2 (which is treated as a signed integer) is positive and the result is negative.

Operation:

w2 ← (w2) + wq
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Quick

15							8	7 4	3 0
1	0	0	0	0	0	1	0	wq	w2

Example:

```
addq      $4,r1       # Add 4 to (r1)
```

Add Single Floating

Syntax: **adds** *s1,s2*

Description: Add the single-precision contents of floating register s1 to the single-precision contents of floating register s2 and put the result in s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction.

Operation:

s2 ← (s2) + (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:

Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15 8	7 4	3 0
0 0 1 0 0 0 0 0	s1	s2

Example:

```
loads   $value1,f0    # Load single floating value at $value1 into f0
loads   (r3),f2       # Load single floating value at (r3) into f2
adds    f2,f0         # Add floating regs f0 and f2 and put the result in f0
```

ADDW — Add Word — ADDW

Syntax: **addw** *w1,w2*

Description: Add the contents of general register w1 to the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have the same sign and the result has the opposite sign.

Operation:

w2 ← (w2) + (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register

15 – 8	7 – 4	3 – 0
1 0 0 0 0 0 0 0	w1	w2

Example:

```
addw    r12,r0    # Add contents of r12 and r0, put the result in r0
```

ADDWC Add Word With Carry ADDWC

Syntax: **addwc** *w1,w2*

Description: Add the contents of general register w1 and the carry flag to the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers.

Operation:
$w2 \leftarrow (w2) + (w1) + C$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register

15							8	7 … 4	3 … 0
1	0	0	1	0	0	0	0	w1	w2

Example: Assume (r0,r1) and (r4,r5) contain two longwords and the operation to be performed is:

$(r4,r5) \leftarrow (r4,r5) + (r0,r1).$

```
addw        r0,r4       # Add low words, get carry
addwc       r1,r5       # Add high words using carry
```

And Immediate

Syntax: **andi** *wi,w2*

Description: Bitwise AND the immediate value wi with the contents of general register w2 and put the result in w2.

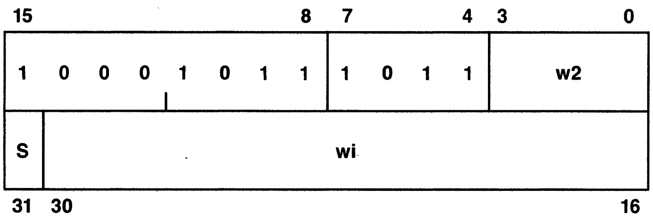
Operation:

```
w2 ← (w2) & wi
N  ← (w2<31>)
Z  ← (w2) = 0
V  ← 0
C  ← 0
```

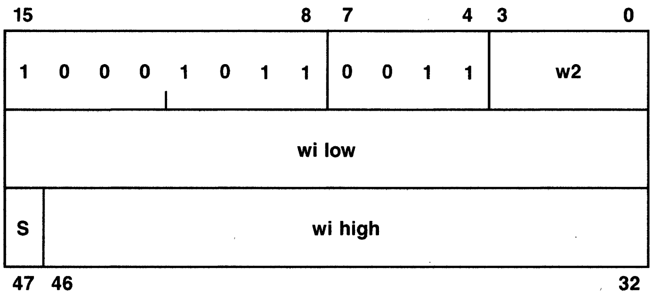
Traps: none

Format: Immediate

If $-2^{15} \le wi \le +2^{15} - 1$, this format is used:



If $+2^{15} \le wi \le +2^{31} - 1$ or $-2^{15} \le wi \le -2^{31} - 1$, this format is used:



Examples: Assume r8 contains 0x001100ff.

```
andi    $0xffff,r8    # AND 32-bit value with r8
```

The result put in r8 is 0x000000ff.

Assume r2 contains 0xffff0000.

```
andi    $023,r2       # AND 16-bit value with r2
```

The result put in r2 is 0x00000000.

ANDW — And Word — ANDW

Syntax: **andw** *w1,w2*

Description: Bitwise AND the contents of general register w1 with the contents of general register w2 and put the result in w2.

Operation:

```
w2 ← (w2) & (w1)
N  ← (w2<31>)
Z  ← (w2) = 0
V  ← 0
C  ← 0
```

Traps: none

Format: Register

15							8	7 … 4	3 … 0
1	0	0	0	1	0	0	0	w1	w2

Example: Assume r2 contains 0x7788ffff and r3 contains 0xffff0000.

```
andw    r2,r3    # AND (r2) with (r3), put result in r3
```

The result put in r3 is 0x77880000.

Branch On Condition

Syntax: **b*** *ha*

Description: If, for the selected condition cond, PSW flags C, V, Z, and, N match one of the patterns shown in Table 2-7, then the program branches to address ha; otherwise it does not branch. Cond is set by selecting one of the operands listed in Table 2-7.

When a choice of mnemonics is shown, use the mnemonics beginning with bc if the conditions to be tested were set by a compare instruction; otherwise use the mnemonics beginning with br.

Operation:

```
if selected condition,
  then PC ← ha
```

Traps: none

Format: Address

If addressing is relative, this format is used:

15–8	7–4	3–0
0 1 0 0 1 0 0 0	**R1**	**cond**

For all other addressing, this format is used:

15–8	7–4	3–0
0 1 0 0 1 0 0 1	**addr mode**	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63		

Examples: Assume (r3) contains 17, (r6) contains – 19.

```
movw    r3,r4
brgt    label1     # Branches

cmpw    r3,r6
bcge    label2     # Branches

cmpw    r3,r6
bcgeu   label3     # Doesn't branch
```

Table 2-7 Integer Branch Conditions

	PSW Flags		
cond	C V Z N	Name	Condition
0	X X X X	b	Branch always

	PSW Flags				
cond	C V Z N	Name	Compare R1:R2	Name	Result R2:0
1	X 0 0 0 X 1 0 1	bclt	Less Than	brgt	Greater Than
2	X 0 X 0 X 1 0 1	bcle	Less or Equal	brge	Greater or Equal
3	X X 1 0	bceq	Equal	breq	Equal
4	X 0 0 1 X 1 X 0	bcgt	Greater Than	brlt	Less Than
5	X 1 X 0 X 0 0 1 X X 1 0	bcge	Greater or Equal	brle	Less or Equal
6	X X 0 X X X 1 1	bcne	Not Equal	brne	Not Equal
7	0 X 0 X	bcltu	Less Than Unsigned	brgtu	Greater Than Unsigned
8	0 X X X	bcleu	Less or Equal Unsigned	brgeu	Greater Than or Equal Unsigned
9	1 X X X	bcgtu	Greater Than Unsigned	bcltu	Less Than Unsigned
A	1 X X X X X 1 X	bcgeu	Greater or Equal Unsigned	brleu	Less or Equal Unsigned

	PSW Flags		
cond	C V Z N	Name	Condition
8	0 X X X	bnc	Not Carry
9	1 X X X	bc	Carry
B	X 1 X X	bv	oVerflow
C	X 0 X X	bnv	Not oVerflow
D	X X 0 1	bn	Negative
E	X X X 0	bnn	Not Negative
F	X X 1 1	bfn	Floating uNordered

Legend:
X = don't care

The R2 field of the **branch on condition** instruction selects the conditions on which to branch. When a choice of mnemonics is shown, use the ones beginning with **bc** if the condition codes to be tested were set by a compare instruction. Use the mnemonics beginning with **br** if they were set by move or logical instructions (those instructions that set only N or Z).

Branch On Floating Exception

Syntax: **bf*** *ha*

Description: If the exceptions selected by the mnemonic are met, put ha in the PC. The cond field selects exceptions on which to branch as shown in Table 2-8.

Table 2-8 Floating Branch Conditions

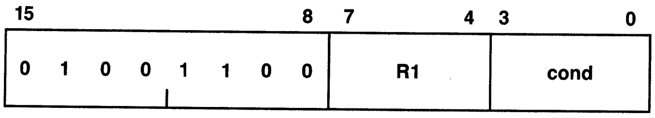
cond	Name	Exception
0	bfany	Floating ANY exception
1	bfbad	Floating BAD result
2–F		Reserved

Operation:
```
if selection condition,
    then PC ← ha
```

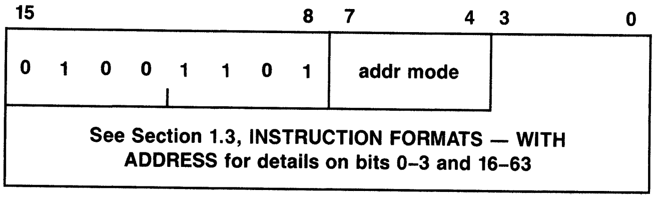
Traps: none

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:


15
8
7
4
3
0
0 1 0 0 1 1 0 1
addr mode
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63

Example: Assume (f0) contains 1.0, (f1) contains 0.0, and there are no traps enabled.

```
divd    f0,f1     # Divide by zero
bfdz    label1    # branches
```

Call Subroutine

Syntax: **call** *w2,ha*

Description: The return address (the address of the instruction following the **call** instruction) is pushed onto the stack. The stack pointer is contained in w2. Control is then transferred to the specified address. On an exception, the stack pointer has not been modified.

Operation:

w2 ← (w2)
(w2) ← (PC)
PC ← ha

Traps:

Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	0	0	0	1	0	0	R1	w2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	0	0	0	1	0	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63									

Example:

```
call      sp,print      # Call the print routine
```

CALLS — Call Supervisor — CALLS

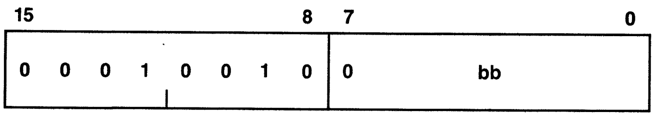
Syntax: **calls** *bb*

Description: Call the supervisor. Cause one of the 128 unique traps through the indicated supervisor trap vector bb. The current status registers (SSW, PSW and PC) are pushed on the supervisor's stack. The new PC and SSW are taken from the trap vector and the PSW is set to zero.

Operation: trap 400 + 8 × bb

Traps: none

Format: Control



Example: calls $6 # Invoke supervisor call # 6

Compare Characters

Syntax: **cmpc**

Description: Compare a string of bytes. The string length is in r0, the address of the first string is in r1, and the address of the second string is in r2. On an exception, the registers are updated so that restarting the instruction compares the remaining portions of the strings. The bytes are signed extended to 32 bits, then compared as words.

Operation:

```
while [(r0) ≠ 0] & [((r2)) = ((r1))],
      r0 ← (r0) − 1
      r1 ← (r1) + 1
      r2 ← (r2) + 1

if (r0) = 0,
  then N ← 0
       Z ← 1
       V ← 0
       C ← 0
  else temp ← (r2) − (r1)
       N ← (temp<31>)
       Z ← (temp) = 0
       V ← overflow
       C ← carry out
```

Traps: Page fault
Read protect fault
Memory fault

Format: Macro

15							8	7			4	3			0
1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example: Assume the string "ABCD" is at label **str1** and the string "ABXY" is at label **str2.**

```
loadq    $4,r0      # Load length into r0
loada    str1,r1    # Load addr of str1 into r1
loada    str2,r2    # Load addr of str2 into r2
cmpc                # Compare the strings
bcge     label1     # Does not branch, (r0) = 2,
                    # (r1) points to "C", str1 + 3
                    # (r2) points to "X", str2 + 3
```

Compare Double Floating

Syntax: **cmpd** *d1,d2*

Description: Compare the double-precision contents of floating register d1 with the double-precision contents of floating register d2. Only the condition codes are affected.

NOTE
Although +0.0 and −0.0 are represented differently, they compare equal.

Operation:
(d2) − (d1)
N ← [[(d2) − (d1)] ≤ 0] OR [(d2),(d1) unordered]
Z ← [[(d2) − (d1)] = 0] OR [(d2),(d1) unordered]
V ← 0
C ← 0

Traps: none

Format: Register

15 … 8	7 … 4	3 … 0
0 0 1 0 0 1 1 1	d1	d2

Example: cmpd f1,f2 # Compare contents of floating regs 1 & 2

CMPI Compare Immediate CMPI

Syntax: **cmpi** *wi,w2*

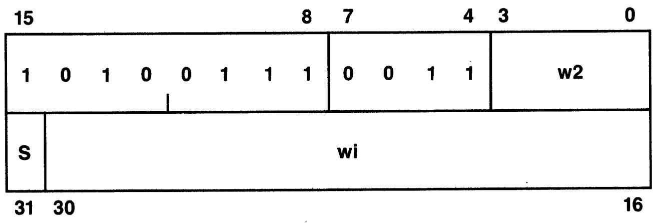
Description: Compare the immediate value wi with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – wi
N ← [(w2) – wi] < 0
Z ← [(w2) – wi] = 0
V ← integer overflow
C ← integer carry out

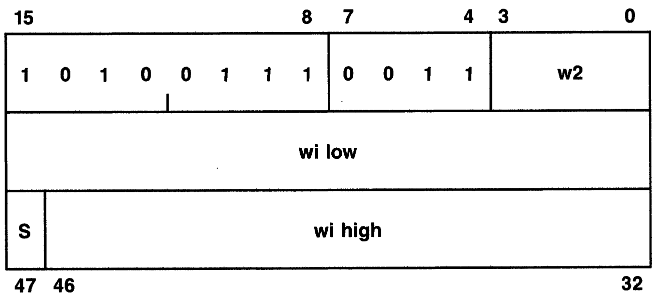
Traps: none

Format: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{31} - 1$, this format is used:



Example: cmpi $0x1f,r2 # Compare 0x1f to the contents of r2

CMPQ — Compare Quick — CMPQ

Syntax: **cmpq** *wq,w2*

Description: Compare the quick value wq with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – wq
N ← [(w2) – wq] < 0
Z ← [(w2) – wq] = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Quick

15							8	7 — 4	3 — 0
1	0	1	0	0	1	1	0	wq	w2

Example:
```
        cmpq      $0,r3       # Compare 0 to the contents of r3
```

CMPS — Compare Single Floating — CMPS

Syntax: **cmps** *s1,s2*

Description: Compare the single-precision contents of floating register s1 with the single-precision contents of floating register s2. Only the condition codes are affected.

NOTE
Although +0.0 and −0.0 are represented differently, they compare equal.

Operation:

```
(s2) – (s1)
N ← [[(d2) – (d1)] < 0] OR [(d2),(d1) unordered]
Z ← [[(d2) – (d1)] = 0] OR [(d2),(d1) unordered]
V ← 0
C ← 0
```

Traps: none

Format: Register

15							8	7 … 4	3 … 0
0	0	1	0	0	1	0	1	s1	s2

Example:

```
cmps	f0,f3	# Compare the contents of f0 to the contents of f3
```

CMPW — Compare Word — CMPW

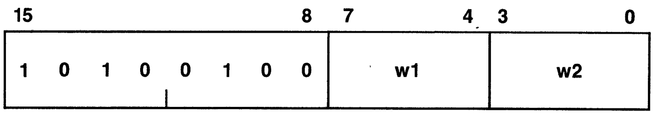
Syntax: **cmpw** *w1,w2*

Description: Compare the contents of general register w1 with the contents of general register w2. Only the condition codes are affected.

Operation:
(w2) – (w1)
N ← [(w2) – (w1)] < 0
Z ← [(w2) – (w1)] = 0
V ← integer overflow
C ← integer carry out

Traps: none

Format: Register



Example:

```
        cmpw        r0,r2        # Compare the contents of r0 to the contents of r2
```

Convert Double to Single Floating

Syntax: **cnvds** *d1,s2*

Description: Convert the double-precision contents of floating register d1 to single-precision and put the result in floating register s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction. The source and destination registers may be the same register.

Operation:
s2 ← (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	1	0	0	1
0	0	0	0	0	0	0	0	d1				s2			
31							24	23			20	19			16

Example:

```
cnvds    f0,f2    # Convert double in f0 to single, put in f2
cnvds    f1,f1    # Convert double in f1 to single
```

Convert Double Floating to Word

Syntax: **cnvdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to a signed integer using the IEEE rounding mode given in the PSW and put the result in general register w2.

Attempting to convert ±∞, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	1	0	0
0	0	0	0	0	0	0	0	d1				w2			
31							24	23			20	19			16

Example:
```
cnvdw   f1,r2    # Convert double in f1 to integer, put in r2
```

Convert Rounding Double Floating to Word

Syntax: **cnvrdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to the nearest signed integer by adding a properly signed 0.5, truncating toward 0.0, and putting the result in general register w2. This is the round operation in ISO Pascal, regardless of the IEEE rounding mode given in the PSW, and differs from the IEEE round-to-nearest mode.

Attempting to convert ±∞, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number for which the truncation portion of the operation is inexact causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	1	0	1

31							24	23			20	19			16
0	0	0	0	0	0	0	0	s1				s2			

Example:

```
        cnvrdw      f0,r2       # Convert double in f0 to integer, put in r2
```

CNVRSW Convert Rounding Single Floating to Word CNVRSW

Syntax: **cnvrsw** *s1,w2*

Description: Convert the single-precision contents of floating register s1 to the nearest signed integer by adding a properly signed 0.5, truncating toward 0.0, and putting the result in general register w2. This is the round operation in ISO Pascal, independent of the IEEE rounding mode in the PSW, and differs from the IEEE round to nearest mode.

Attempting to convert ±∞, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number for which the truncation portion of the operation is inexact causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	0	0	1
0	0	0	0	0	0	0	0	s1				w2			
31							24	23			20	19			16

Example: cnvrsw f1,r0 # Convert single in f1 to integer, put in r0

Convert Single to Double Floating

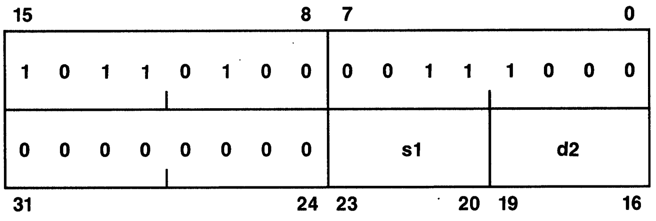
Syntax: **cnvsd** *s1,d2*

Description: Convert the single-precision contents of floating register s1 to double-precision and put the result in floating register d2. The source and destination registers may be the same register.

Operation: d2 ← (s1)
FI ← floating invalid

Traps: Floating invalid operation

Format: Macro



Example:

```
cnvsd      f0,f2       # Convert single in f2 to double, put in f2
```

Convert Single Floating to Word

Syntax: **cnvsw** *s1,w2*

Description: Convert the single-precision contents of floating register s2 to a signed integer using the IEEE rounding mode given in th PSW, and put the result in general register w2.

Attempting to convert ±∞, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	s1				w2			
31							24	23			20	19			16

Example:

```
cnvsw     f0,r3     # Convert single in f0 to integer, result in w2
```

CNVTDW Convert Truncating Double Floating to Word CNVTDW

Syntax: **cnvtdw** *d1,w2*

Description: Convert the double-precision contents of floating register d1 to a signed integer by truncating toward 0.0 and putting the result in general register w2. This is the INT operation in ANSI FORTRAN 77 and the trunc operation in ISO Pascal, regardless of the setting of the IEEE rounding mode in the PSW.

Attempting to convert ±∞, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:
w2 ← (d1)
FX ← floating inexact result
FI ← floating invalid

Traps:
Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	1	1	0
0	0	0	0	0	0	0	0	d1				w2			
31							24	23			20	19			16

Example: cnvtdw f1,r1 # Truncate double in f1 to integer, put in r2

Convert Truncating Single Floating to Word

Syntax: **cnvtsw** *s1,w2*

Description: Convert the single-precision contents of floating register s1 to a signed integer by truncating toward 0.0 and putting the result in general register w2. This is the INT operation in ANSI FORTRAN 77 and the trunc operation in ISO Pascal, regardless of the setting of the IEEE rounding mode in the PSW.

Attempting to convert $\pm\infty$, a NaN, or a number too large to fit in a word causes an invalid operation exception, and the stored result is undefined. Attempting to convert a number that is not an exact integer but is small enough for conversion causes a floating inexact result exception.

On a trap, the PC can be obtained with the **loadfs** instruction. The destination original value, normally returned by the **loadfs** instruction, is undefined.

Operation:

w2 ← (s1)
FX ← floating inexact result
FI ← floating invalid

Traps:

Floating invalid operation
Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	0	s1				w2			
31							24	23			20	19			16

Example:

```
        cnvtsw      f0,r2       # Truncate f0 to integer, store in r2
```

CNVWD Convert Word to Double Floating CNVWD

Syntax: **cnvwd** *w1,d2*

Description: Convert the contents of general register w1 to double-precision and put the result in floating register d2.

Operation: d2 ← (w1)

Traps: none

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	1	1	1
0	0	0	0	0	0	0	0	w1				d2			
31							24	23			20	19			16

Example: cnvwd r1,f0 # Convert integer in r1 to double, result in f0

CNVWS Convert Word to Single Floating CNVWS

Syntax: **cnvws** *w1,s2*

Description: Convert the contents of general register w1 to single-precision floating-point and put the result in floating register s2. If the conversion is not exact, the result is rounded according to the IEEE rounding mode in the PSW, and floating inexact result is signalled.

Operation: s2 ← (w1)
FX ← floating inexact result

Traps: Floating inexact result

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	0	0	1	1
0	0	0	0	0	0	0	0	w1				s2			
31							24	23			20	19			16

Example:

```
cnvws      r3,f2      # Convert integer in r3 to single, result in f2
```

Divide Double Floating

Syntax: **divd** *d1,d2*

Description: Divide the double-precision contents of floating register d2 by the double-precision contents of floating register d1 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:

d2 ← (d2) ÷ (d1)
FX ← floating inexact result
FU ← floating underflow
FD ← floating divide-by-zero
FV ← floating overflow
FI ← floating invalid

Traps:

Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow
Floating divide-by-zero

Format: Register

15 – 8	7 – 4	3 – 0
0 0 1 0 1 0 1 1	d1	d2

Example:

```
divd    f2,f0    # Divide double in f0 by double in f2, put result in f0
```

DIVS — Divide Single Floating — DIVS

Syntax: **divs** *s1,s2*

Description: Divide the single-precision contents of floating register s2 by the single-precision contents of floating register s1 and put the result in s2. On a trap, the PC and the original value in s2 can be obtained by using the **loadfs** instruction.

Operation:
(s2) ← (s2) ÷ (s1)
FX ← floating inexact result
FU ← floating underflow
FD ← floating divide-by-zero
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow
Floating divide-by-zero

Format: Register

15							8	7 4	3 0
0	0	1	0	1	0	0	1	s1	s2

Example:

```
divs     f1,f2     # Divide single in f2 by single in f1, put result in f2
```

Divide Word

Syntax: **divw** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the *quotient* in w2. The operands are treated as signed integers. The quotient is positive if the signs of the divisor and dividend are the same, and negative if they are different. Overflow is set if the largest negative number (-2^{31}) is divided by -1.

NOTE

For the same dividend and divisor, if the quotient returned by **divw** is multiplied by the divisor and then added to the remainder returned by **modw,** the result would be the original dividend.

Operation:

$w2 \leftarrow (w2) \div (w1)$
$N \leftarrow 0$
$Z \leftarrow 0$
$V \leftarrow$ integer overflow
$C \leftarrow 0$

Traps: Divide-by-zero

Format: Register

15 8	7 4	3 0
1 0 0 1 1 1 0 0	w1	w2

Example:

```
        divw     r10,r4       # Divide (r4) by (r10), put result in r4
```

DIVWU   Divide Word Unsigned   DIVWU

Syntax: **divwu** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the quotient in w2. The operands are treated as unsigned integers. Since the quotient is always positive, overflow cannot occur.

NOTE

For the same dividend and divisor, if the quotient returned by **divwu** and the remainder returned by **modwu** are multiplied, the result would be the original dividend.

Operation:

```
w2 ← (w2) ÷ (w1)
N  ← 0
Z  ← 0
V  ← 0
C  ← 0
```

Traps: Divide-by-zero

Format: Register

15							8	7 4	3 0
1	0	0	1	1	1	1	0	w1	w2

Example:

```
divwu     r0,r5     # Divide (r5) by (r0), put result in r5
```

Initialize Characters

Syntax: **initc**

Description: Initialize a byte, halfword, word, or single-precision floating string with a constant value. The length of the string (in bytes) is in the r0 and the address is in r1. The pattern to be stored is in r2. For word and single-precision floating strings, r2 contains the initial value. For halfword strings, both halfwords of r2 must contain the initial value. For byte strings, all four bytes of r2 must contain the desired value. If a trap occurs, the registers are updated so that restarting the instruction initializes the remaining portion of the string.

Operation:

```
while (r0) ≠ 0,
(r1) ← (r2<7:0>)
r0  ← (r0) – 1
r1  ← (r1) + 1
r2  ← (r2) ROT – 8
```

Traps: Page fault
Write protect fault
Memory fault

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example: Set the 17 bytes beginning at label **str** to 19.

```
loadi    $17,r0            # String length
loada    str,r1            # String address
loadi    $0x13131313,r2    # 19 decimal = 13 hex
initc                      # Initialize characters
```

LOADA — Load Address — LOADA

Syntax: **loada** *ba,w2*

Description: Load the memory address ba into general register w2.

Operation: w2 ← ba

Traps: none

Formats: Address

If addressing is relative, this format is used:

15–8	7–4	3–0
0 1 1 0 0 0 1 0	R1	w2

For all other addressing, this format is used:

15–8	7–4	3–0
0 1 1 0 0 0 1 1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63		

Examples:

loada	value,r0	# Address of value into r0
loada	8(sp),fp	# (sp) + 8 into fp
loada	addr(r4),r0	# (r4) + addr into r0
loada	[r1](r2),r3	# 3 operand addr that sets no flags

Load Byte

Syntax: **loadb** *ba,w2*

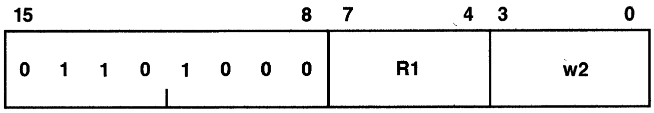
Description: Load the byte at memory address ba, sign-extended, into the least-significant byte of general register w2.

Operation: w2 ← (ba)

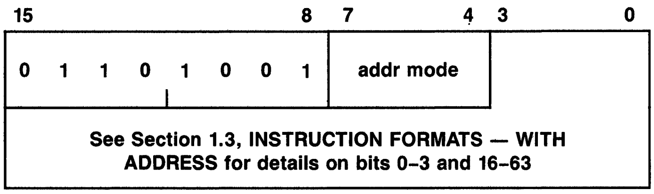
Traps:
Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63

Example: loadb 3(r7),r2 # Load byte at 3(r7) into r2

LOADBU — Load Byte Unsigned — LOADBU

Syntax: **loadbu** *ba,w2*

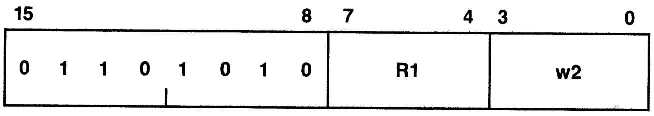
Description: Load the byte at memory address ba, zero-extended, into the least-significant byte of general register w2.

Operation: w2 ← (ba)

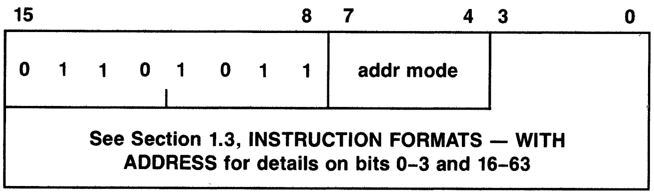
Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:



See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63

Example:

```
        loadbu      4(r5),r2        # Load byte at 4(r5) into r2
```

Load Double Floating

Syntax: **loadd** *da,d2*

Description: Load the double-precision floating-point value at memory addresses da and da + 1 into floating register d2.

Operation: d2 ← (da)

Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	1	0	0	1	1	0	R1	d2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	1	0	0	1	1	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63									

Examples:

```
loadd    addr,f2     # Load double floating value at addr into f2
loadd    4(r2),f2    # Load double floating value at address specified by 4(r2)
                     # into f2
```

LOADFS — Load Floating Status

Syntax: **loadfs** *w1,d2*

Description: Load the floating status following a floating trap. The PC of the offending floating instruction is put in general register w1. The original value of the destination register is put in floating register d2. This information allows a trap handler to determine the original operation and its operands.

Operation: w1 ← (FP PC)
d2 ← (FP dest)

Traps: none

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	0	w1				d2			
31							24	23			20	19			16

Example:

```
loadfs    r1,f2     # Load FP PC into r1, and original destination into f2
```

Load Halfword

Syntax: **loadh** *ha,w2*

Description: Load the halfword at memory address ha, sign-extended, into the least-significant halfword of general register w2.

Operation: w2 ← (ha)

Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	1	0	1	1	0	0	R1	w2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	1	0	1	1	0	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63									

Example:

```
loadh     (r15),r12     # Load the halfword at (r15) into r12
```

Load Halfword Unsigned

Syntax: **loadhu** *ha,w2*

Description: Load the halfword at memory address ha, zero-extended, into the least-significant halfword of general register w2.

Operation: w2 ← (ha)

Traps: Page fault
Read protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15 8	7 4	3 0
0 1 1 0 1 1 1 0	R1	w2

For all other addressing, this format is used:

15 8	7 4	3 0
0 1 1 0 1 1 1 1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63		

Example:

```
loadhu    new,r11    # Load the halfword at new into r11
```

Load Immediate

Syntax: **loadi** *wi,w2*

Description: Load the immediate value wi, sign-extended, into general register w2.

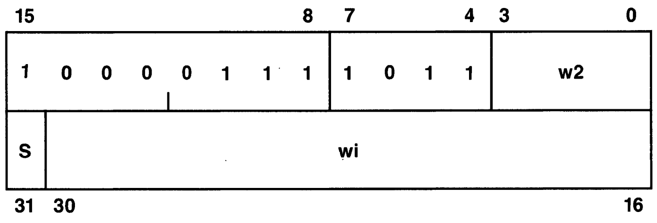
Operation:

$w2 \leftarrow wi$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

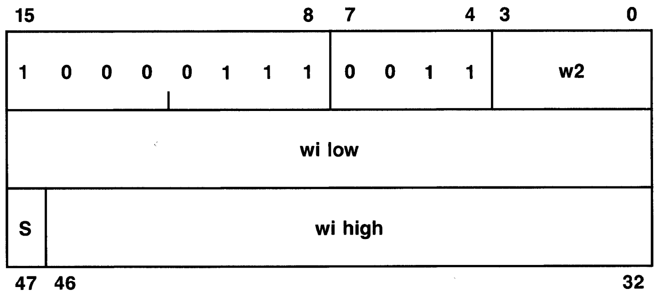
Traps: none

Format: Immediate

If $-2^{15} \le wi \le +2^{15} - 1$, this format is used:



If $+2^{15} \le wi \le +2^{31} - 1$ or $-2^{15} \le wi \le -2^{15} - 1$, this format is used:



Example:

```
        loadi       $21,r2      # Load 21 into r2
```

LOADQ Load Quick LOADQ

Syntax: **loadq** *wq,w2*

Description: Load the quick value wq, zero-extended, into general register w2.

Operation:

$w2 \leftarrow wq$
$N \leftarrow 0$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Quick

15							8	7 … 4	3 … 0
1	0	0	0	0	1	1	0	wq	w2

Example:

```
loadq     $0xf,r1     # Load f hex into r1
```

LOADS — Load Single Floating — LOADS

Syntax:	**loads** *sa,s2*
Description:	Load the single-precision floating-point value at memory address sa into floating register s2.
Operation:	s2 ← (sa)
Traps:	Page fault Read protect fault Memory fault
Format:	Address

If addressing is relative, this format is used:

15							8	7 — 4	3 — 0
0	1	1	0	0	1	0	0	**R1**	**s2**

For all other addressing, this format is used:

15							8	7 — 4	3 — 0
0	1	1	0	0	1	0	1	**addr mode**	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63									

Example:

```
loads    clock,f2     # Load single floating value at clock into f2
loads    (r3), f3     # Load single floating value at (r3) into f3
```

LOADW Load Word LOADW

Syntax: **loadw** *wa,w2*

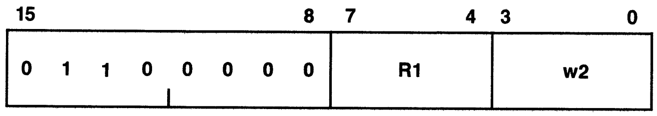
Description: Load the contents of memory address wa into general register w2.

Operation: w2 ← (wa)

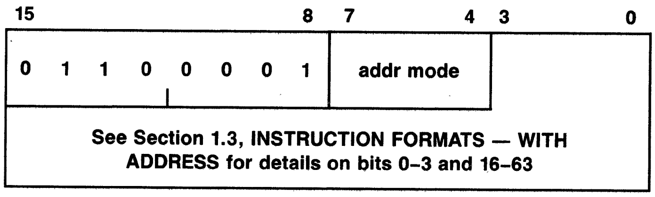
Traps: Page fault
Read protect fault
memory fault

Format: Address

If addressing is relative, this format is used:



For all other addressing, this format is used:


15
8
7
4
3
0
0 1 1 0 0 0 0 1
addr mode
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63

Example:

```
loadw    abc,r2         # Load word at abc into r2

loadw    $0xfca2,r1     # Load word at address fca2 hex into r1
```

MODW Modulus Word MODW

Syntax: **modw** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the *remainder* in w2. The operands are treated as signed integers. The quotient is positive if the signs of the divisor and dividend are the same, and negative if they are different. If the remainder is non-zero, it has the same sign as the dividend. (Overflow is set if the largest negative number (-2^{31}) is divided by -1.)

NOTE

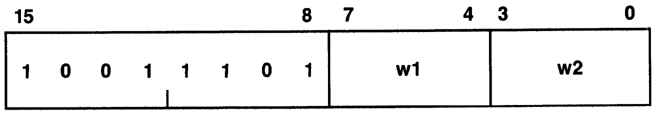
For the same dividend and divisor, if the quotient returned by **divw** is multiplied by the divisor and then added to the remainder returned by **modw,** the result would be the original dividend.

Operation:

```
w2 ← (w2) MOD (w1)
N  ← 0
Z  ← 0
V  ← integer overflow
C  ← 0
```

Traps: Divide-by-zero

Format: Register



Example:

```
loadi    $1234,r2    # Load 1234 into r2
loadq    $11,r1      # Load 11 into r1
modw     r1,r2       # Divide r2 by r1, save remainder in r2
```

MODWU — Modulus Word Unsigned — MODWU

Syntax: **modwu** *w1,w2*

Description: Divide the contents of general register w2 by the contents of general register w1 and put the remainder in w2. The operands are treated as unsigned integers. If the remainder is non-zero, then it is positive.

NOTE

For the same dividend and divisor, if the quotient returned by **divwu** and the remainder returned by **modwu** are multiplied, the result would be the original dividend.

Operation:
w2 ← (w2) MOD (w1)
N ← 0
Z ← 0
V ← 0
C ← 0

Traps: Divide-by-zero

Format: Register

15							8	7 4	3 0
1	0	0	1	1	1	1	1	w1	w2

Example:

```
loadi    $0xfffe,r5    # Load fffe hex into r5
loadq    $0x11,r0      # Load 11 hex into r0
modw     r0,r5         # Divide r5 by r0, save remainder in r5
```

Move Characters

Syntax: **movc**

Description: Move a string of bytes. The length of the strings is in r0, the address of the source string is in r1, and the address of the destination string is in r2. On an exception, the registers are updated so that restarting the instruction moves the remaining portion of the string.

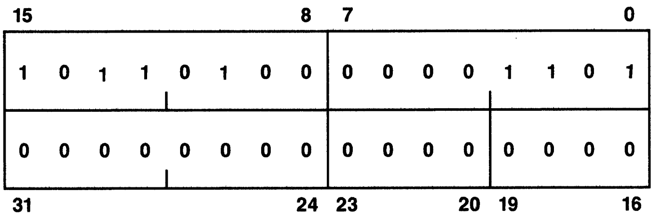
The source and destination strings may not overlap. The **initc** instruction should be used to initialize memory or the source string may be moved in blocks.

Operation:

```
while (r0) ≠ 0
      (r2) ← ((r1))
       r0 ← (r0) − 1
       r1 ← (r1) + 1
       r2 ← (r2) + 1
```

Traps: Page fault
Read protect fault
Write protect fault
Memory fault

Format: Macro



Example: Move a string of 124 characters at label **string1** to **string2.**

```
loadi     $124,r0       # Load string length into r0
loada     string1,r1    # Load address of source string into r1
loada     string2,r2    # Load address of destination string into r2
movc                    # Move the characters
```

MOVD — Move Double Floating — MOVD

Syntax: **movd** *d1,d2*

Description: Move the double-precision contents of floating register d1 to floating register d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation: d2 ← (d1)

Traps: none

Format: Register

15–8	7–4	3–0
0 0 1 0 0 1 1 0	**d1**	**d2**

Example:

```
        movd        f1,f2        # Move (f1) to f2
```

MOVDL — Move Double Floating to Longword

Syntax: **movdl** *d1,l2*

Description: Move the double-precision contents of floating register d1 to longword register pair l2 without conversion.

Operation: l2 ← (d1)

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	0	1	1	1	0	d1	l2

Example:

```
        movdl       f1,r2       # Move (f1) to reg pair (r2,r3)
```

MOVLD Move Longword to Double Floating MOVLD

Syntax: **movld** *l1,d2*

Description: Move the contents of longword register pair l1 to floating register d2 without conversion.

Operation: (d2) ← (l1)

Traps: none

Format: Register

15							8	7	4	3 0
0	0	1	0	1	1	1	1	l1	0	d2

Example:

```
        movld        r0,f2        # Move (r2,r3) to double floating reg f2
```

MOVPW — Move Processor Register to Word — MOVPW

Syntax: **movpw** *p1,w2*

Description: Move the contents of processor register p1 to general register w2. The p1 value is interpreted as follows:

p1	Name	Meaning
0	PSW	Program status word
1	SSW	Supervisor status word
2-15	—	(Reserved)

Operation: w2 ← (p1)

Traps: none

Format: Register

15–8	7–4	3–0
0 0 0 1 0 0 0 1	p1	w2

Examples:

```
movpw     psw,r2     # Move (PSW) to r2

movpw     ssw,r2     # Move (SSW) to r2
```

MOVS Move Single Floating MOVS

Syntax: **movs** *s1,s2*

Description: Move the single-precision contents of floating register s1 to floating register s2.

Operation: s2 ← (s1)

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	0	0	1	0	0	s1	s2

Example:

```
movs      f1,f2      # Move (f1) to f2
```

Move Supervisor to User (Privileged)

Syntax: **movsu** *w1,w2*

Description: Move the contents of supervisor general register w1 to user general register w2. A privileged instruction trap occurs if this instruction is executed in user mode.

Operation: $w2_{usr} \leftarrow (w1)_{sup}$

$N \leftarrow (w2<31>)_{usr}$

$Z \leftarrow (w2)_{usr} = 0$

$V \leftarrow 0$
$C \leftarrow 0$

Traps: Privileged instruction

Format: Macro

15							8	7							0
1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	w1				w2			
31							24	23			20	19			16

Example: movsu r0,r11 # Move (r0) to r11

MOVSW Move Single Floating to Word MOVSW

Syntax: **movsw** *s1,w2*

Description: Move the single-precision contents of floating register s1 to general register w2 without conversion.

Operation: w2 ← (s1)

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	0	1	1	0	0	s1	w2

Example:

```
movsw     f1,r2     # Move (f1) to r2
```

Move User to Supervisor (Privileged)

Syntax: **movus** *w1,w2*

Description: Move the contents of user general register w1 to supervisor general register w2.

Operation: $w2_{sup} \leftarrow (w1)_{usr}$

$N \leftarrow (w2<31>)_{sup}$

$Z \leftarrow (w2)_{sup} = 0$

$V \leftarrow 0$
$C \leftarrow 0$

Traps: Privileged instruction

Format: Macro

15							8	7							0
1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0

31							24	23 20	19 16
0	0	0	0	0	0	0	0	w1	w2

Example:

```
        movus     r1,r2      # Move (r1) to r2
```

MOVW — Move Word — MOVW

Syntax: **movw** *w1,w2*

Description: Move the contents of general register w1 to general register w2.

Operation:

$w2 \leftarrow (w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Register

15 — 8	7 — 4	3 — 0
1 0 0 0 0 1 0 0	w1	w2

Example:

```
        movw      r3,r0       # Move (r3) to r0
```

Move Word to Processor Register

Syntax: **movwp** *w2,p1*

Description: Move the contents of general register w2 to processor register p1. The p1 value is interpreted as follows:

p1	**Name**	**Meaning**
0	PSW	Program status word
1	SSW	Supervisor status word
2–15	—	(Reserved)

If p1 represents the PSW, then the condition codes are also set. Attempting to modify the SSW in user mode will cause a **noop.**

Operation:

p1 $\leftarrow$ (w2)
N $\leftarrow$ [p1 = psw] AND PSW$<$N$>$
Z $\leftarrow$ [p1 = psw] AND PSW$<$Z$>$
V $\leftarrow$ [p1 = psw] AND PSW$<$V$>$
C $\leftarrow$ [p1 = psw] AND PSW$<$C$>$

Traps: none

Format: Register

15							8	7 4	3 0
0	0	0	1	0	0	0	0	p1	w2

Examples:

```
movwp    r2,psw    # Move (r2) to PSW
```

The following example only works in supervisor mode:

```
movwp    r2,ssw    # Move (r2) to SSW
```

MOVWS   Move Word to Single Floating   MOVWS

Syntax: **movws** *w1,s2*

Description: Move the contents of general register w1 to floating register s2 without conversion.

Operation: s2 ← (w1)

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	0	1	1	0	1	w1	s2

Example:

```
movws    r1,f2    # Move (r1) to f2
```

MULD — Multiply Double Floating — MULD

Syntax: **muld** *d1,d2*

Description: Multiply the double-precision contents of floating register d2 by the double-precision contents of floating register d1 and put the result in d2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:

d2 ← (d2) × (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:

Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15							8	7 … 4	3 … 0
0	0	1	0	1	0	1	0	d1	d2

Example: Assume fpval1 contains 1239237.1234 and fpval2 contains 8989.44334.

```
loadd     fpval1,f1     # Load double floating value
loadd     fpval2,f2     # Load double floating value
muld      f1,f2         # Multiply the values
```

The result is 11140051905.62889.

MULS Multiply Single Floating MULS

Syntax: **muls** *s1,s2*

Description: Multiply the single-precision contents of floating register s2 by the single-precision contents of floating register s1 and put the result in s2. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:

s2 ← (s2) × (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:

Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15							8	7 4	3 0
0	0	1	0	1	0	0	0	s1	s2

Example:

```
          loadi     $0xf8ccd,r0     # Load r0 with value
          movws     r0,f0           # Transfer to f0
          loadi     $0xc27dd,r0     # Load r0 with value
          movws     r0,f1           # Transfer to f1
          muls      f0,f1           # Multiply the numbers
          stors     f1,place1       # Save the result
          .
          .
          .
place1:   .space
```

Multiply Word

Syntax: **mulw** *w1,w2*

Description: Multiply the contents of general register w2 by general register w1 and and put the least-significant word of the product in w2. The operands are treated as signed integers. Overflow is set if the product cannot be represented in one word.

Operation:
w2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: none

Format: Register

15 8	7 4	3 0
1 0 0 1 1 0 0 0	w1	w2

Example:

```
loadi      $999,r4       # Load 999 into r4
loadw      wdata,r2      # Get value at wdata
mulw       r2,r4         # Multiply the numbers
```

Multiply Word Unsigned

Syntax: **mulwu** *w1,w2*

Description: Multiply the contents of general register w2 by the contents of general register w1 and put the least-significant word of the product in w2. The operands are treated as unsigned integers. Overflow is set if the result cannot be represented in one word.

Operation:

w2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← integer overflow
C ← 0

Traps: none

Format: Register

15 8	7 4	3 0
1 0 0 1 1 0 1 0	w1	w2

Example:

```
loadhu    hwval,r0    # Load the value at hwval
loadq     $33,r1      # Load 33 into r1
mulwu     r0,r1       # Multiply the numbers
```

Multiply Word Unsigned Extended

Syntax: **mulwux** *w1,l2*

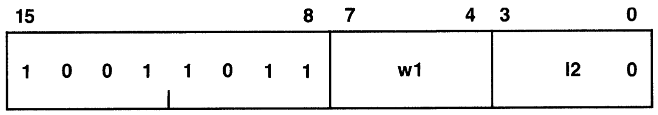
Description: Multiply the contents of general register w1 by the contents of general register w2 and put the product in longword register pair l2. The operands are treated as unsigned integers. Overflow is set if the product cannot be represented in one word.

Operation:

l2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← result requires a longword
C ← 0

Traps: none

Format: Register



Example:

```
loadhu    uhwval,r3    # Load the value at uhwval
loadi     $0xff,r6     # Load ff hex into r6
mulwux    r3,r6        # Multiply the numbers
```

MULWX Multiply Word-Extended MULWX

Syntax: **mulwx** *w1,l2*

Description: Multiply the contents of general register w1 by the contents of general register w2 and put the product in longword register pair l2. Overflow is set if the product cannot be represented in one word.

Operation:
l2 ← (w2) × (w1)
N ← 0
Z ← 0
V ← product requires a longword
C ← 0

Traps: none

Format: Register

15							8	7 4	3	0
1	0	0	1	1	0	0	1	w1	l2	0

Example:

```
        loadhu      wxval,r0        # Load the value at wxval
        loadi       $1023,r4        # Load 1023 into r4
        mulwx       r0,r4           # Multiply the numbers
```

Negate Double Floating

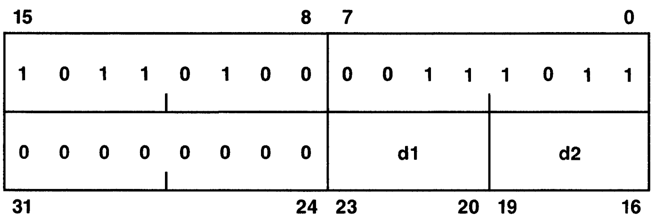
Syntax: **negd** *d1,d2*

Description: Negate the double-precision contents of floating register d1 and put the result in floating register d2. The sign of d1 is reversed so that ±0.0 and NaNs are handled properly.

Operation: d2 ← −(d1)

Traps: none

Format: Macro



Example:

```
        negd        f0,f0       # Negate the register
```

Negate Single Floating

Syntax: **negs** *s1,s2*

Description: Negate the single-precision contents of floating register s1 and put the result in floating register s2. The sign of s1 is reversed so that ±0.0 and NaNs are handled properly.

Operation: s2 ← −(s1)

Traps: none

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	1	0	1	0

31							24	23–20	19–16
0	0	0	0	0	0	0	0	s1	s2

Example:

```
        negs    f1,f2       # Negate the value in f1, store in f2
```

Negate Word

Syntax: **negw** *w1,w2*

Description: Two's complement the contents of general register w1 and put the result in general register w2. Overflow is set if w1 contains the largest negative number (-2^{31}). Carry is set if w1 does not equal 0.

Operation:
$w2 \leftarrow -(w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow$ integer overflow
$C \leftarrow$ borrow in

Traps: none

Format: Register

15 8	7 4	3 0
1 0 0 1 0 0 1 1	w1	w2

Example:

```
        negw        r1,r1        # Two's complement (r1)
```

NOOP — No Operation — NOOP

Syntax: **noop** *bb*

Description: No operation is performed. bb is ignored.

Operation: none

Traps: none

Format: Control

15							8	7 … 0
0	0	0	0	0	0	0	0	bb

Examples:

```
noop    $0    # No operation
noop          # Operand is optional
```

NOTQ — Not Quick — NOTQ

Syntax: **notq** *wq,w2*

Description: Zero fill the quick value wq on the left, take its one's complement and put the result in general register w2.

Operation:

```
w2 ← ~wq
N  ← 1
Z  ← 0
V  ← 0
C  ← 0
```

Traps: none

Format: Quick

15							8	7 … 4	3 … 0
1	0	1	0	1	1	1	0	wq	w2

Example:

```
notq    $4,r0    # Load -5 into r0
```

NOTW — Not Word — NOTW

Syntax: **notw** *w1,w2*

Description: Take the one's complement of the contents of general register w1 and put the result in general register w2.

Operation:
$w2 \leftarrow \sim(w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Register

15							8	7 – 4	3 – 0
1	0	1	0	1	1	0	0	w1	w2

Example:
```
        notw    r1,r2       # One's complement r1, put in r2
```

OR Immediate

Syntax: **ori** *wi,w2*

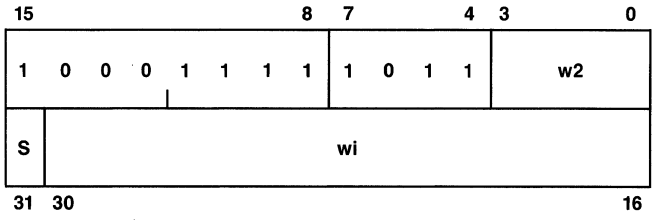
Description: Bitwise OR the contents of general register w2 with immediate value wi, and put the result in w2. A 16-bit immediate value is sign-extended first.

Operation:
$w2 \leftarrow (w2) \mid wi$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

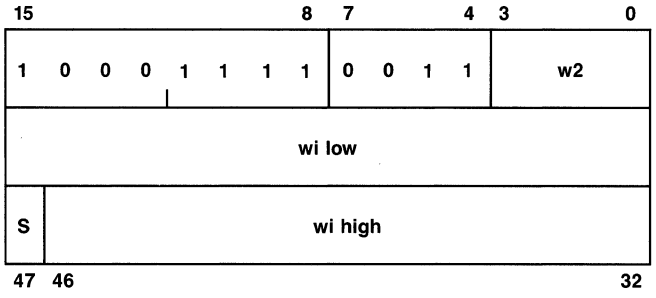
Traps: none

Formats: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Example: Assume r0 contains 0x00ff00ff.

```
ori     $0xff,r0     # Or 0xff with (r0)
```

The result put in r0 is 0x00ff00ff.

OR Word

Syntax: **orw** *w1,w2*

Description: Bitwise OR the contents of general register w2 with the contents of general register w1 and put the result in w2.

Operation:
$w2 \leftarrow (w2) \mid (w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Register

15							8	7 … 4	3 … 0
1	0	0	0	1	1	0	0	w1	w2

Example: Assume r0 contains 0x7770088f and r1 contains 0x001100ff.

```
orw     r1,r0     # Or (r0) with (r1)
```

The result put in r0 is 0x777108ff.

POPW — Pop Word — POPW

Syntax: **popw** *w1,w2*

Description: Pop the word on the top of the stack into general register w2. General register w1 contains the stack address. The stack grows from high to low addresses.

On a data page fault, the contents of w1 will have been incremented. The page fault handler must check for this case and decrement the stack pointer by 4 before restarting the instruction.

Operation:
w1 ← (w1) + 4
w2 ← ((w1) − 4

Traps:
Page fault
Read protect fault
Memory fault

Format: Register

15							8	7 4	3 0
0	0	0	1	0	1	1	0	w1	w2

Example: Assume sp = r15 = stack pointer.

```
        popw        sp,r0        # Pop word into r0
```

Push Word

Syntax: **pushw** *w2,w1*

Description: Push the contents of general register w2 onto the stack. General register w1 contains the stack address. The stack grows from high to low addresses.

On a data page fault, the contents of w1 will have been decremented. The page fault handler must check for this case and increment the stack pointer by 4 before restarting the instruction.

Operation:

$w1 \leftarrow (w1) - 4$
$(w1) \leftarrow (w2)$

Traps:

Page fault
Write protect fault
Memory fault

Format: Register

15 8	7 4	3 0
0 0 0 1 0 1 0 0	w1	w2

Example: Assume sp = r15 = stack pointer.

```
pushw     r2,sp     # Push (r2) onto stack
```

RESTD*n* Restore Double Floating Registers f*n* – f7 RESTD*n*

Syntax: **restd*n***

Description: This description represents the eight instructions **restd0** through **restd7**. The **restd*n*** instruction restores the double-precision, floating registers f*n* through f7 from the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting. A floating register containing a single-precision value will be restored properly by the appropriate **restd*n*** instruction, but will not appear in IEEE single-precision format while in memory.

Operation: dn : d7 ← ((r15)) : ((r15) + 8 × [7 – n])
r15 ← (r15) + 8 × [8 – n]

Traps: Page fault
Read protect fault
Memory fault

Format: Macro

15							8	7				3	2		0
1	0	1	1	0	1	0	0	0	0	1	0	1		n	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example:

```
restd3    # Restore floating registers f3 : f7
restd7    # Restore floating register f7
```

RESTUR Restore User Registers (Privileged) RESTUR

Syntax: **restur** *w1*

Description: Restore all the user registers. Restore the contents of all user registers r0 through r15 from supervisor memory addressed by supervisor general register w1. Register w1 may be the supervisor stack pointer, r15. On a data page fault, w1 is unchanged to permit restarting.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation: $r0 : r15_{usr} \leftarrow ((w1)) : ((w1) + 60)_{sup}$
$w1 \leftarrow (w1) + 64$

Traps: Privileged instruction
Page fault
Read protect fault
Memory fault

Format: Macro

15							8	7			4	3			0
1	0	1	1	0	1	1	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	w1				0	0	0	0
31							24	23			20	19			16

Example: restur r1 # Restore user's registers

Restore Registers r*n* - r14

Syntax: **restw*n***

Description: This description represents the 13 instructions **restw0** through **restw12**. The **restw*n*** instruction restores the general registers *rn* through r14 from the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting.

Operation: rn : r14 ← ((r15)) : ((r15) + 4 × (14 – n))
r15 ← (r15) + 4 × [15 – n]

Traps: Page fault
Read protect fault
memory fault

Format: Macro

15							8	7				3	2		0
0	0	1	1	0	1	0	0	0	0	0	1	0	n		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example: restw5 # Restore registers r5 : r14

Return from Subroutine

Syntax: **ret** *w2*

Description: Pop the word on top of the stack into the PC. General register w2 contains the stack address. This undoes the effect of the **call** instruction.

Operation: PC ← ((w2))
w2 ← (w2) + 4

Traps: Page fault
Read protect fault
Memory fault

Format: Register

15							8	7			4	3 0
0	0	0	1	0	0	1	1	0	0	0	0	w2

Example: Assume sp = r15 = stack pointer.

```
ret     sp      # Return to calling program
```

Return from Interrupt (Privileged)

Syntax: **reti** *w1*

Description: Return from interrupt or trap. Restore the contents of the SSW, PSW, and PC from the supervisor stack addressed by the contents of general register w1. The supervisor must assure that the stack references do not cause page or protect faults.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation:

$SSW \leftarrow ((w1))$
$PSW \leftarrow ((w1)+4)$
$PC \leftarrow ((w1)+8)$
$w1 \leftarrow (w1)+12$

Traps: Privileged instruction

Format: Macro

15							8	7							0
1	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	w1				0	0	0	0
31							24	23			20	19			16

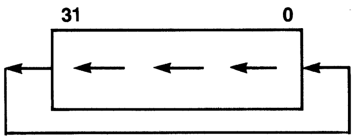
Example: Assume sp = r15 = stack pointer.

```
        reti        sp        # Return from interrupt
```

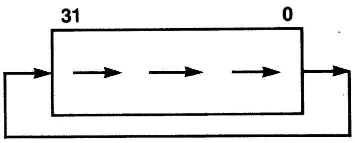
ROTI — Rotate Immediate — ROTI

Syntax: **roti** *wi,w2*

Description: Rotate the contents of general register w2 by the number of bits given in the 16-bit immediate value wi. A positive count rotates the contents of w2 to the left, moving bit 31 into bit 0:



A negative count rotates to the right, moving bit 0 into bit 31:

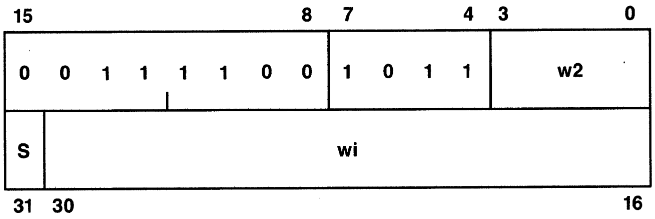


Operation:

```
w2 ← (w2) ROT wi
N  ← (w2<31>)
Z  ← (w2) = 0
V  ← 0
C  ← 0
```

Traps: none

Format: Immediate



Example: Assume r1 contains 0x3333ffff.

```
roti    $1,r1       # Rotate r1 left 1 place
```

The result put in r1 is 0x6667fffe.

Assume r1 contains 0x3333ffff.

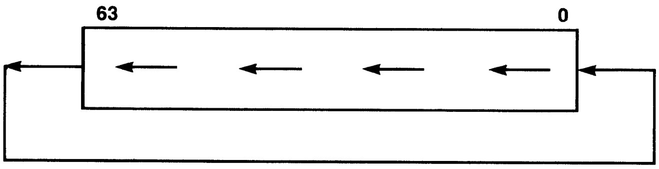
```
roti    $ – 1,r1    # Rotate r1 right 1 place
```

The result put in r1 is 0x9999ffff.

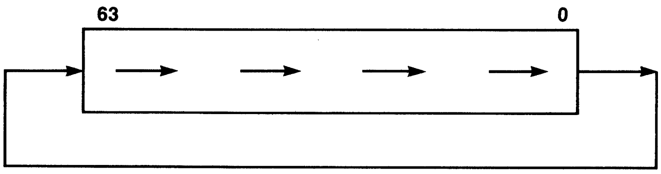
Rotate Longword

Syntax: **rotl** *w1,l2*

Description: Rotate the contents of longword register pair l2 by the number of bits given in general register w1. A positive count rotates the contents of l2 to the left, moving bit 63 into bit 0:



A negative count rotates the contents of w2 to the right, moving bit 0 into bit 63:



Operation:

l2 ← (l2) ROT (w1)
N ← (l2<63>)
Z ← (l2) = 0
V ← 0
C ← 0

Traps: none

Format: Register

15 … 8	7 … 4	3 … 0
0 0 1 1 0 1 0 1	w1	l2 0

Example: Assume r0 contains 2, r4,r5 contains 0x3333ffff 77770000.

```
        rotl      r0,r4       # Rotate r4,r5 left 2 place
```

The result put in r4,r5 is 0xcccfffffbbb80000.

Assume r0 contains −2, r4,r5 contains 0x3333ffff 77770000.

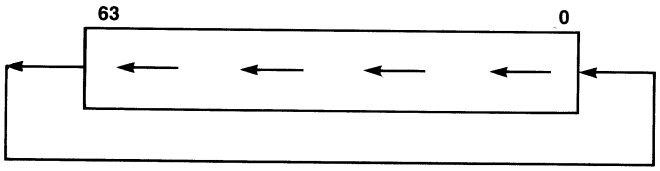
```
        rotl      r0,r4       # Rotate r4,r5 right 2 places
```

The result put in r4,r5 is 0x0cccfffffbbb8000.

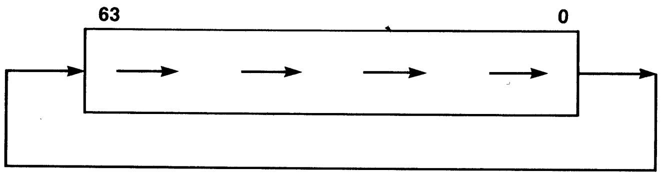
Rotate Long Immediate

Syntax: **rotli** *wi,l2*

Description: Rotate the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. A positive count rotates the contents of l2 to the left, moving bit 63 into bit 0:



A negative count rotates the contents of l2 to the right, moving bit 0 into bit 63:



Operation:

```
l2 ← (l2) ROT wi
N ← (l2<63>)
Z ← (l2)=0
V ← 0
C ← 0
```

Traps: none

Formats:

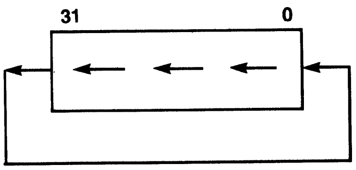
15							8	7			4	3			0
0	0	1	1	1	1	0	1	1	0	1	1		l2		0
S							wi								
31	30														16

Example: rotli $8,r2 # Rotate left 1 byte

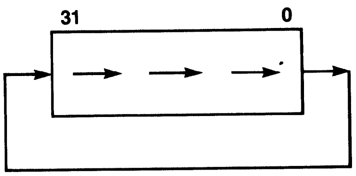
Rotate Word

Syntax: **rotw** *w1,w2*

Description: Rotate the contents of general register w2 by the number of bits given in w1. A positive count rotates the contents of general register w2 to the left, moving bit 31 into bit 0:



A negative count rotates right, moving bit 0 into bit 31:



Operation:
w2 ← (w2) ROT (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	1	0	1	0	0	w1	w2

Example: Assume r0 contains 8 and r2 contains 0x000f0071.

```
rotw      r0,r2      # Rotate r2 left 8 bits
```

The result put in r2 is 0x0f007100.

SAVED*n* Save Registers f*n* : f7 SAVED*n*

Syntax: **saved***n*

Description: This description represents the eight instructions **saved**0 through **saved**7. The **saved***n* instructions save the double-precision floating registers dn through d7 on the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting. A floating register that contains a single-precision value will be restored properly by the appropriate **restd***n* instruction, but it will not appear in IEEE single-precision format while in memory.

Operation: (r15) – 8 × [8 – n] : (r15) ← (dn) : (d7)
r15 ← (r15) – 8 × [8 – n]

Traps: Page fault
Write protect fault
Memory fault

Format: Macro

15							8	7				3	2		0
1	0	1	1	0	1	0	0	0	0	1	0	0	n		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23							16

Example: saved4 # Save floating registers f4 : f7

SAVEUR — Save User Registers (Privileged) — SAVEUR

Syntax: **saveur** *w1*

Description: Save the contents of user general registers r0 through r15 in supervisor memory addressed by supervisor general register w1. Register w1 may be the supervisor stack pointer, r15. On a data page fault, w1 is unchanged to permit restarting.

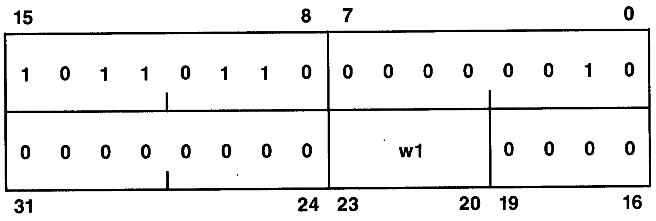
A privileged instruction trap occurs if this instruction is executed outside of supervisor mode.

Operation: $(w1) - 4 : (w1) - 64_{sup} \leftarrow (r15) : (r0)_{usr}$

$w1 \leftarrow (w1) - 64$

Traps: Privileged instruction
Page fault
Write protect fault
Memory fault

Format: Macro



Example: saveur r1 # Save user's registers r1 : r15

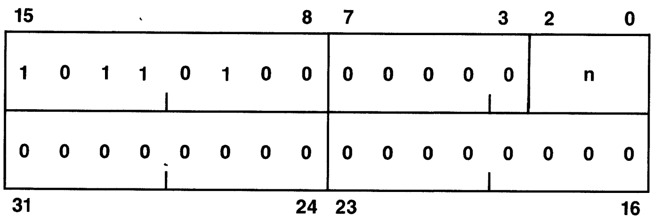
Syntax: **savew*n***

Description: This description represents the 13 instructions **savew**0, through **savew**12. The **savew*n*** instruction saves the general registers rn through r14 on the stack. The stack pointer is assumed to be in register r15. On a data page fault, the stack pointer is unchanged to permit restarting.

Operation: (r15) – 4 × [15 – n] : (r15) ← (rn) : (r14)
r15 ← (r15) – 4 × [15 – n]

Traps: Page fault
Write protect fault
Memory fault

Format: Macro



Example: savew6 # Save registers r6 : r14

ADVANCE INFORMATION

SCALBD — Scale By, Double Floating — SCALBD

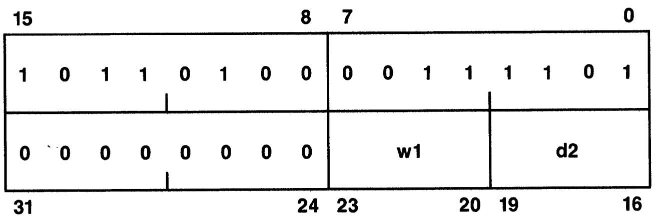
Syntax: **scalbd** *w1,d2*

Description: Multiply the double-precision contents of floating register d2 by two raised to the integer contents of general register w1 and put the result in d2. In the normal case, just the exponent is modified; a multiply operation is not performed. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
$d2 \leftarrow (d2) \times 2^{(w1)}$
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro



Example:

```
scalbd     r1,f0     # Scale the floating number
```

SCALBS — Scale By, Single Floating — SCALBS

Syntax: **scalbs** *w1,s2*

Description: Multiply the contents of single-precision floating register s2 by two raised to the integer contents of general register w1 and put the result in s2. In the normal case, just the exponent is modified; a multiply operation is not performed. On a trap, the PC and the original value in d2 can be obtained by using the **loadfs** instruction.

Operation:
$s2 \leftarrow (s2) \times 2^{(w1)}$
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Macro

15							8	7							0
1	0	1	1	0	1	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	w1				s2			
31							24	23			20	19			16

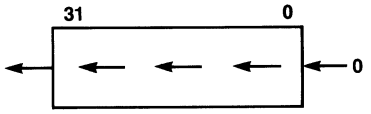
Example:

```
        scalbs      r1,f0       # Scale the double floating value
```

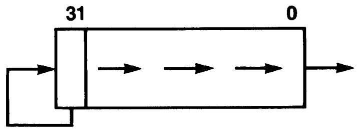
SHAI — Shift Arithmetic Immediate — SHAI

Syntax: **shai** *wi,w2*

Description: Shift arithmetically the contents of general register w2 by the number of bits given by the 16-bit immediate value wi. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of general register w2 left, bringing zeros into bit 0 and shifting the bit 31 out:



A negative count shifts right, bringing in copies of bit 31:

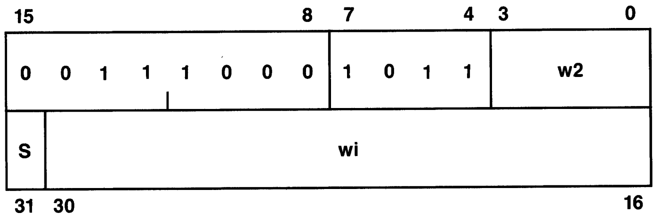


Operation:

w2 ← (w2) SHA wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Immediate



Example: Assume r2 contains 0xffff0000.

```
        shaw        $8,r2        # Shift r2 left 8 places
```

The result put in r2 is 0xff000000.

Assume r2 contains 0xffff0000.

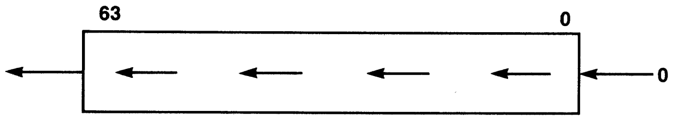
```
        shaw        $ – 8,r2     # Shift r2 right 8 places
```

The result put in r2 is 0xffffff00.

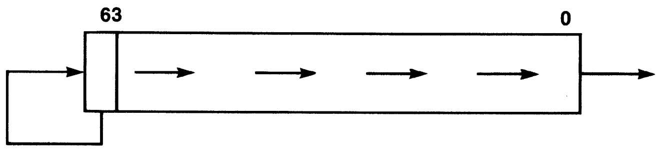
SHAL — Shift Arithmetic Longword — SHAL

Syntax: **shal** *w1,l2*

Description: Shift arithmetically the contents of longword register pair l2 by the number of bits given in w1. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



A negative count shifts right, bringing in copies of bit 63:



Operation:

```
l2 ← (l2) SHA (w1)
N ← (l2<63>)
Z ← (l2)=0
V ← integer overflow
C ← 0
```

Traps: none

Format: Register

15–8	7–4	3–0
0 0 1 1 0 0 0 1	w1	l2 0

Example: Assume r1 contains 5, r6,r7 contains 0x1111ffff00ff00ff.

```
shal     r1,r6     # Shift r6,r7 left 5 places
```

The result put in r6,r7 is 0x223fffe01fe01fe0.

Assume r1 = −5, r6,r7 = 0xffff111100ff00ff.

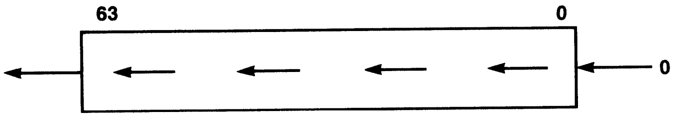
```
shal     r1,r6     # Shift r6,r7 right 5 places
```

The result put in r6,r7 is 0xfffff8888 8807f807.

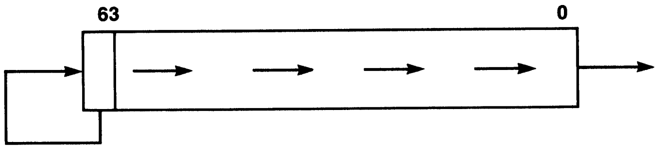
SHALI — Shift Arithmetic Longword Immediate — SHALI

Syntax: **shali** *wi,l2*

Description: Shift arithmetically the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



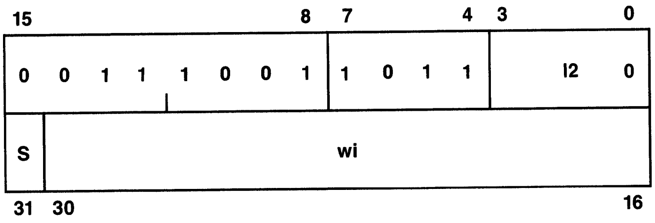
A negative count shifts right, bringing in copies of bit 63:



Operation:
l2 ← (l2) SHA wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Immediate



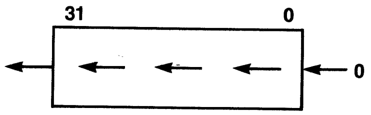
Example: Assume r8 contains 0x1110ffff.

```
shali       $-4,r8       # Shift r8 right 4 bits
```

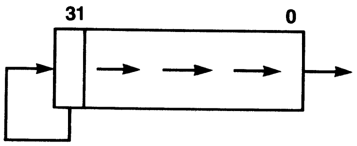
The result put in r8 is 0x01110ff.

Syntax: **shaw** *w1,w2*

Description: Shift arithmetically the contents of general register w2 by the number of bits given in general register w1. Overflow is set at the end of the operation if the sign of the result changes at any time during the operation. A positive count shifts the contents of general register w2 to the left, bringing zeros into bit 0:



A negative count shifts right, bringing in copies of bit 31:



Operation:

w2 ← (w2) SHA (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← 0

Traps: none

Format: Register

15							8	7 … 4	3 … 0
0	0	1	1	0	0	0	0	w1	w2

Example: Assume r10 contains 8, r2 contains 0xffff0000.

```
shaw      r10,r2      # Shift r2 left 8 places
```

The result put in r2 is 0xff000000.

Assume r10 contains – 8, r2 contains 0xffff0000.

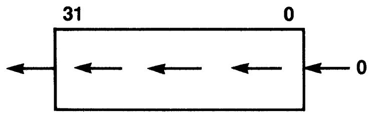
```
shaw      r10,r2      # Shift r2 right 8 places
```

The result put in r2 is 0xffffff00.

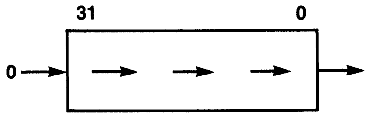
Shift Logical Immediate

Syntax: **shli** *wi,w2*

Description: Shift logically the contents of general register w2 by the number of bits given in the 16-bit immediate value wi. A positive count shifts the contents of w2 out the left; bringing zeros into bit 0:



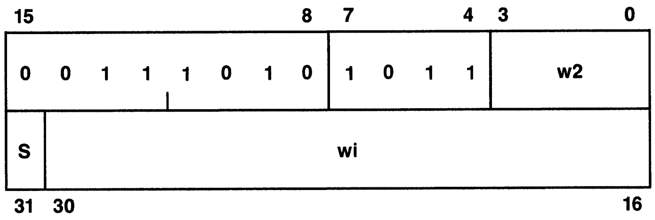
A negative count shifts the contents of w2 out the right; bringing zeros into bit 31.



Operation:
w2 ← (w2) SHL wi
N ← (w2<31>)
Z ← (w2) = 0
V ← 0
C ← 0

Traps: none

Format: Immediate



Example: Assume r4 contains 0xffff0000.

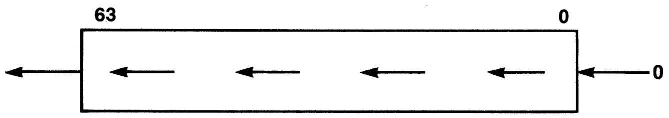
```
shli    $-8,r4    # Shift r4 right 8 places
```

The result put in r4 is 0x00ffff00.

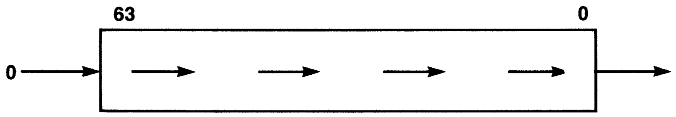
Shift Logical Longword

Syntax: **shll** *w1,l2*

Description: Shift logically the contents of longword register pair l2 by the number of bits given in general register w1. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



A negative count shifts right, bringing zeros into bit 63:



Operation:
l2 ← (l2) SHL (w1)
N ← (l2<63>)
Z ← (l2) = 0
V ← 0
C ← 0

Traps: none

Format: Register

15							8	7 4	3 0
0	0	1	1	0	0	1	1	w1	l2 0

Examples: Assume r1 contains 16 and r2,r3 contains 0x123456780000ffff.

```
        sh11        r1,r2        # Shift (r2,r3) by (r1)
```

The result put in r2,r3 is 0x56780000ffff0000.

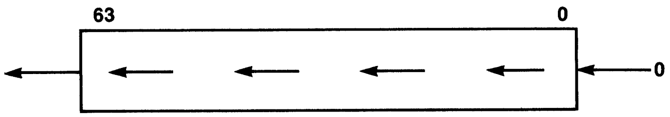
Assume r1 contains – 16 and r2,r3 contains 0x123456780000ffff.

```
        sh11        r1,r2        # Shift (r2,r3) by (r1)
```

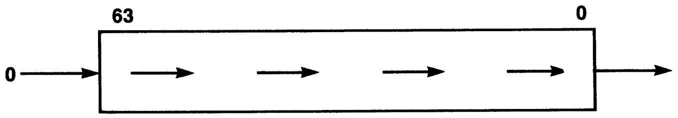
The result put in r2,r3 is 0x0000000012345678.

Syntax: **shlli** *wi,l2*

Description: Shift logically the contents of longword register pair l2 by the number of bits given in the 16-bit immediate value wi. A positive count shifts the contents of l2 to the left, bringing zeros into bit 0:



A negative count shifts right, bringing zeros into bit 63:

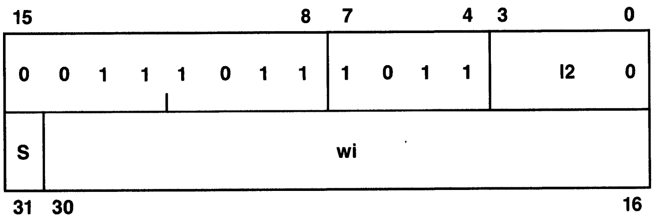


Operation:

```
l2 ← (l2) SHL wi
N ← (l2<63>)
V ← (l2)=0
Z ← 0
C ← 0
```

Traps: none

Format: Immediate



Example: Assume r2,r3 contains 0x7777ffff7777ffff.

```
sh11i    $8,r2     # Shift (r2,r3) left 8 places
```

The result put in r2,r3 is 0x77ffff7777ffff00.

Assume r2,r3 contains 0x7777ffff7777ffff.

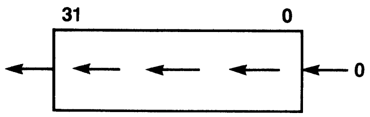
```
sh11i    $-8,r2    # Shift (r2,r3) right 8
```

The result put in r2,r3 is 0x007777ffff7777ff.

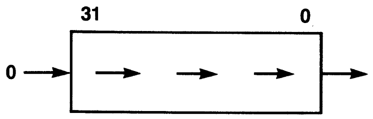
Shift Logical Word

Syntax: **shlw** *w1,w2*

Description: Shift logically the contents of general register w2 by the number of bits given in general register w1. A positive count shifts the contents of general register w2 to the left, bringing zeros into bit 0:



A negative count shifts right, bringing zeros into bit 31:



Operation:

$w2 \leftarrow (w2)\ \mathrm{SHL}\ (w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Register

15							8	7 … 4	3 … 0
0	0	1	1	0	0	1	0	w1	w2

Examples: Assume r1 contains 4 and r0 contains 0xff.

```
shlw     r1,r0     # Shift (r0) by (r1)
```

The result put in r0 is 0x00000ff0.

Assume r1 contains −4 and r0 contains 0xff.

```
shlw     r1,r0     # Shift (r0) by (r1)
```

The result put in r0 is 0x0000000f.

STORB — Store Byte — STORB

Syntax: **storb** *w2,ba*

Description: Store the least-significant byte of the contents of general register w2 into memory address ba.

Operation: ba ← (w2)

Traps:
Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 … 4	3 … 0
0	1	1	1	1	0	0	0	R1	w2

For all other addressing, this format is used:

15							8	7 … 4	3 … 0
0	1	1	1	1	0	0	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63									

Example:

```
storb     r2,label1     # Store (r2) at label1
```

STORD Store Double Floating STORD

Syntax: **stord** *d2,da*

Description: Store the double-precision contents of floating register d2 into memory addresses da.

Operation: da ← (d2)

Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	1	1	0	1	1	0	R1	w2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	1	1	0	1	1	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63									

Example: stord f0,fpsave # Store (f0) at fpsave

STORH — Store Halfword

Syntax: **storh** *w2,ha*

Description: Store the least-significant halfword of the contents of general register w2 into memory address ha.

Operation: ha ← (w2)

Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	1	1	1	1	0	0	R1	w2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	1	1	1	1	0	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63									

Example:

```
        storh     r12,hwsav1      # Store (r12) at hwsav1
```

STORS Store Single Floating STORS

Syntax: **stors** *s2,sa*

Description: Store the single-precision contents of floating register s2 into memory address sa.

Operation: sa ← (s2)

Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15 8	7 4	3 0
0 1 1 1 0 1 0 0	**R1**	**w2**

For all other addressing, this format is used:

15 8	7 4	3 0
0 1 1 1 0 1 0 1	**addr mode**	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0–3 and 16–63		

Example: stors f0,sfbuff # Store (f0) at sfbuff

Store Word

Syntax: **storw** *w2,wa*

Description: Store the contents of general register w2 into memory address wa.

Operation: wa ← (w2)

Traps: Page fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15							8	7 4	3 0
0	1	1	1	0	0	0	0	R1	w2

For all other addressing, this format is used:

15							8	7 4	3 0
0	1	1	1	0	0	0	1	addr mode	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63									

Example:

```
        storw       r0,waddr        # Store (r0) at waddr
```

SUBD — Subtract Double Floating — SUBD

Syntax: **subd** *d1,d2*

Description: Subtract the double-precision contents of floating register d1 from the double-precision contents of floating register d2 and put the result in d2.

Operation:
d2 ← (d2) – (d1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15 8	7 4	3 0
0 0 1 0 0 0 1 1	d1	d2

Example:

```
subd    f1,f0    # Subtract (f1) from (f0)
```

Subtract Immediate

Syntax: **subi** *wi,w2*

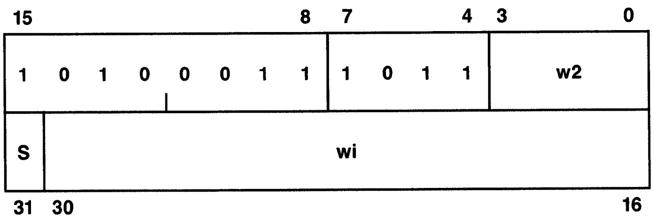
Description: Subtract the immediate value wi from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the input operands (which are treated as signed integers) have different signs and the sign of the result has the same sign as wi.

Operation:
w2 ← (w2) – wi
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

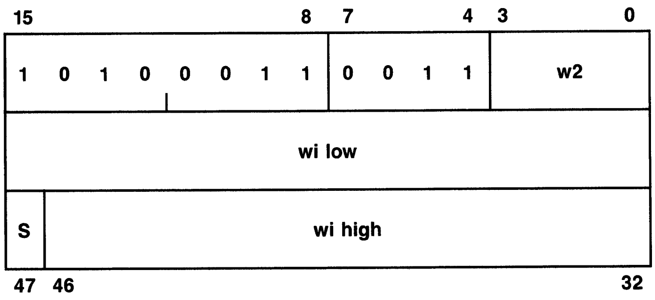
Traps: none

Format: Immediate

If $-2^{15} \leq wi \leq +2^{15}-1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31}-1$ or $-2^{15} \leq wi \leq -2^{15}-1$, this format is used:



Example: subi $0x18,r2 # Subtract 18 hex from (r2)

Subtract Quick

Syntax: **subq** *wq,w2*

Description: Subtract the quick value wq from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the input operands (which are treated as signed integers) have different signs and the result is positive.

Operation:
w2 ← (w2) – wq
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Quick

15							8	7 4	3 0
1	0	1	0	0	0	1	0	wq	w2

Example: subq $10,r3 # Subtract 10 from (r3)

Subtract Single Floating

Syntax: **subs** *s1,s2*

Description: Subtract the single-precision contents of floating register s1 from the single-precision contents of floating register s2 and put the result in d2.

Operation:
s2 ← (s2) – (s1)
FX ← floating inexact result
FU ← floating underflow
FV ← floating overflow
FI ← floating invalid

Traps:
Floating inexact result
Floating invalid operation
Floating overflow
Floating underflow

Format: Register

15 8	7 4	3 0
0 0 1 0 0 0 0 1	s1	s2

Example:

```
        subs      f1,f2        # Subtract (f1) from (f2)
```

Subtract Word

Syntax: **subw** *w1,w2*

Description: Subtract the contents of general register w1 from the contents of general register w2 and put the result in w2. Operands may be signed or unsigned integers. Overflow is set if the operands (which are treated as signed integers) have different signs and the sign of the result is the same as the subtrahend (w1).

Operation:

w2 ← (w2) – (w1)
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Register

15							8	7 4	3 0
1	0	1	0	0	0	0	0	w1	w2

Example:

```
subw     r0,r2     # Subtract (r0) from (r2)
```

SUBWC — Subtract Word with Carry — SUBWC

Syntax: **subwc** *w1,w2*

Description: Subtract the contents of general register w1 and the carry condition code from the contents of general register w2 and put the result in w2.

Operation:

w2 ← (w2) − (w1) − C
N ← (w2<31>)
Z ← (w2) = 0
V ← integer overflow
C ← borrow in

Traps: none

Format: Register

15 … 8	7 … 4	3 … 0
1 0 0 1 0 0 0 1	w1	w2

Example:

```
        subwc       r1,r2       # Subtract (r1) + C from (r2)
```

Syntax: **trapf*n***

Description: Causes an illegal instruction trap if a floating unordered condition exists. The IEEE draft standard specifies that the six predicates =, ≠, >, ≥, <, and ≤ shall cause floating invalid exceptions on unordered comparisons. The **trapf*n*** instruction, put before a branch instruction, supports the IEEE predicates. The supervisor trap handler must interpret an illegal instruction trap from a **trapf*n*** instruction as if it were a floating invalid operation trap.

Operation: if PSW <ZN> indicates unordered,
 then illegal instruction trap

Traps: Illegal instruction

Format: Macro

15							8	7			4	3			0
1	0	1	1	0	1	0	0	0	0	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example: The following sequence branches to **addr** if the IEEE ≤ predicate is satisfied.

```
cmps      f3,f5      # Compare single floating values
trapfn               # Checks for floating unordered
bclt      addr       # branches
```

Test and Set

Syntax: **tsts** *wa,w2*

Description: Test and set a software lock. Load the contents of memory address wa into general register w2 and set bit 31 in wa (the lock). The operation is indivisible and can be used in a multiprocessor configuration. The lock has been acquired if (w2<31>) = 0 after the instruction has executed.

This instruction may not refer to Boot ROM space or I/O space (System Tag 4 or 5).

Operation: w2 ← (wa)
(wa<31>) ← 1

Traps: Page fault
Read protect fault
Write protect fault
Memory fault

Format: Address

If addressing is relative, this format is used:

15–8	7–4	3–0
0 1 1 1 0 0 1 0	**R1**	**w2**

For all other addressing, this format is used:

15–8	7–4	3–0
0 1 1 1 0 0 1 1	**addr mode**	
See Section 1.3, INSTRUCTION FORMATS — WITH ADDRESS for details on bits 0-3 and 16-63		

Example: tsts addr,r2 # Set software lock

WAIT **Wait for Interrupt (Privileged)** WAIT

Syntax: **wait**

Description: Wait for an interrupt. When an enabled interrupt occurs, the instruction terminates and the interrupt is taken. The interrupt routine may then decide to whether or not to continue the interrupted instruction stream at the instruction following the **wait** instruction.

A privileged instruction trap occurs if this instruction is executed in user mode.

Operation:

```
while no interrupt pending
do nothing
```

Traps: Privileged instruction

Format: Macro

15							8	7							0
1	0	1	1	0	1	1	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31							24	23			20	19			16

Example:

```
wait        # Wait for interrupt
```

XORI — Exclusive-OR Immediate — XORI

Syntax: **xori** *wi,w2*

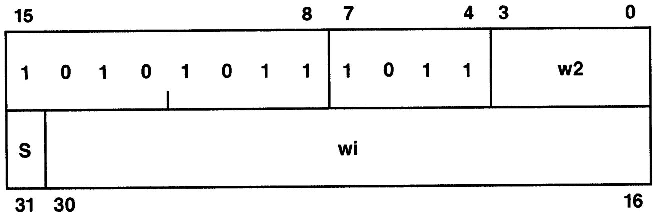
Description: Bitwise exclusive-OR the contents of general register w2 with the immediate value wi and put the result in w2.

Operation:
$w2 \leftarrow (w2) \oplus wi$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

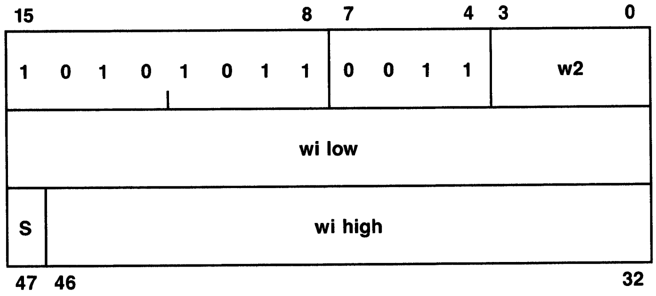
Traps: none

Formats: Immediate

If $-2^{15} \leq wi \leq +2^{15} - 1$, this format is used:



If $+2^{15} \leq wi \leq +2^{31} - 1$ or $-2^{15} \leq wi \leq -2^{15} - 1$, this format is used:



Examples: Assume r0 contains 0xff00.

```
xori    $0xa07f601f,r0    # XOR 0x7f601f with (r0)
```

The result put in r0 is 0xa07f9f1f.

Assume r2 contains 0xc0ffee12.

```
xori    $0xfe,r2          # XOR 0xfe with (r2)
```

The result put in r2 is 0xc0ff10.

XORW — Exclusive-OR Word — XORW

Syntax: **xorw** *w1,w2*

Description: Bitwise exclusive-OR the contents of general register w1 with the contents of general register w2 and put the result in w2.

Operation:

$w2 \leftarrow (w2) \oplus (w1)$
$N \leftarrow (w2<31>)$
$Z \leftarrow (w2) = 0$
$V \leftarrow 0$
$C \leftarrow 0$

Traps: none

Format: Register

15							8	7 … 4	3 … 0
1	0	1	0	1	0	0	0	w1	w2

Example: Assume r1 contains 0xffff0000 and r2 contains 0x00ff00ff.

```
xorw     r1,r2      # XOR regs, result in r2
```

The result put in r2 is 0xff0000ff.

APPENDIX A

ASCII CHARACTER SET

This appendix lists:
- **The ASCII character set**
- **A description of the non-printing ASCII characters**

Table A-1 ASCII Character Set

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	
000	00	NUL (^@)	032	20	SP	064	40	@	096	60		
001	01	SOH (^A)	033	21	!	065	41	A	097	61	a	
002	02	STX (^B)	034	22	"	066	42	B	098	62	b	
003	03	ETX (^C)	035	23	#	067	43	C	099	63	c	
004	04	EOT (^D)	036	24	$	068	44	D	100	64	d	
005	05	ENQ (^E)	037	25	%	069	45	E	101	65	e	
006	06	ACK (^F)	038	26	&	070	46	F	102	66	f	
007	07	BEL (^G)	039	27	'	071	47	G	103	67	g	
008	08	BS (^H)	040	28	(	072	48	H	104	68	h	
009	09	HT (^I)	041	29	)	073	49	I	105	69	i	
010	0A	LF (^J)	042	2A	*	074	4A	J	106	6A	j	
011	0B	VT (^K)	043	2B	+	075	4B	K	107	6B	k	
012	0C	FF (^L)	044	2C	,	076	4C	L	108	6C	l	
013	0D	CR (^M)	045	2D	-	077	4D	M	109	6D	m	
014	0E	SO (^N)	046	2E	.	078	4E	N	110	6E	n	
015	0F	SI (^O)	047	2F	/	079	4F	O	111	6F	o	
016	10	DLE (^P)	048	30	0	080	50	P	112	70	p	
017	11	DC1 (^Q)	049	31	1	081	51	Q	113	71	q	
018	12	DC2 (^R)	050	32	2	082	52	R	114	72	r	
019	13	DC3 (^S)	051	33	3	083	53	S	115	73	s	
020	14	DC4 (^T)	052	34	4	084	54	T	116	74	t	
021	15	NAK (^U)	053	35	5	085	55	U	117	75	u	
022	16	SYN (^V)	054	36	6	086	56	V	118	76	v	
023	17	ETB (^W)	055	37	7	087	57	W	119	77	w	
024	18	CAN (^X)	056	38	8	088	58	X	120	78	x	
025	19	EM (^Y)	057	39	9	089	59	Y	121	79	y	
026	1A	SUB (^Z)	058	3A	:	090	5A	Z	122	7A	z	
027	1B	ESC (^[)	059	3B	;	091	5B	[	123	7B	{	
028	1C	FS (^\)	060	3C	<	092	5C	\	124	7C	|	
029	1D	GS (^])	061	3D	=	093	5D	]	125	7D	}	ALT
030	1E	RS (^^)	062	3E	>	094	5E	^	126	7E	~	ESC
031	1F	US (^_)	063	3F	?	095	5F	_	127	7F		DEL

Table A-2 Nonprintable Characters

Char	Definition	Char	Definition
NUL	Null	DC2	Device Control 2 (TAPE)
SOH	Start of message	DC3	Device Control 3 (X-OFF)
STX	Start of Text	DC4	Device Control 4
ETX	End of Text	NAK	Negative Acknowledge
EOT	End of Transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of Transmission Block
ACK	Acknowledge	CAN	Cancel
BEL	Bell	EM	End of Medium
BS	Back Space	SUB	Substitute
HT	Horizontal Tab	ESC	Escape
LF	Line Feed	FS	File Separator
VT	Vertical Tab	GS	Group Separator
FF	Form Feed	RS	Record Separator
CR	Carriage Return	US	Unit Separator
SO	Shift Out	SP	Space
SI	Shift In	ALT	Alt mode
DLE	Data Link Escape	ESC	Escape prefix
DC1	Device Control 1 (X-ON)	DEL	Delete, Rubout

APPENDIX B
INSTRUCTION SUMMARY

This appendix summaries the instruction set. For a detailed description of each instruction, see Chapter 2.

Table B-1 Functional Instruction Set

LOAD/STORE INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Load Address	loada	*ba,w2*	62,63	Address	1-4	w2 ← ba	. . . . .	. . .	
Load Byte	loadb	*ba,w2*	68,69	Address	1-4	w2 ← (ba)	. . . . .	. . . .	P,R,M,
Load Byte Unsigned	loadbu	*ba,w2*	6a,6b	Address	1-4	w2 ← (ba)	. . . . .	. . . .	P,R,M,
Load Double Floating	loadd	*da,d2*	66,67	Address	1-4	d2 ← (da)	. . . . .	. . . .	P,R,M,
Load Floating Status	loadfs	*w1,d2*	b4 3f	Macro	2	w1 ← (FP PC), d2 ← (FP dest)	. . . . .	. . . .	
Load Halfword	loadh	*ha,w2*	6c,6d	Address	1-4	w2 ← (ha)	. . . . .	. . . .	P,R,M,
Load Halfword Unsigned	loadhu	*ha,w2*	6e,6f	Address	1-4	w2 ← (ha)	. . . . .	. . . .	P,R,M,
Load Immediate	loadi	*wi,w2*	87	Immediate	2,3	w2 ← wi	. . . . .	00**	
Load Quick	loadq	*wq,w2*	86	Quick	1	w2 ← wq	. . . . .	00*0	
Load Single Floating	loads	*sa,s2*	64,65	Address	1-4	s2 ← (sa)	. . . . .	. . . .	P,W,M,
Load Word	loadw	*wa,w2*	60,61	Address	1-4	w2 ← (wa)	. . . . .	. . . .	P,R,M,
Store Byte	storb	*w2,ba*	78,79	Address	1-4	ba ← (w2)	. . . . .	. . . .	P,W,M,
Store Double Floating	stord	*d2,da*	76,77	Address	1-4	da ← (d2)	. . . . .	. . . .	P,W,M,
Store Halfword	storh	*w2,ha*	7c,7d	Address	1-4	ha ← (w2)	. . . . .	. . . .	P,W,M,
Store Single Floating	stors	*s2,sa*	74,75	Address	1-4	sa ← (s2)	. . . . .	. . . .	P,W,M,
Store Word	storw	*w2,wa*	70,71	Address	1-4	wa ← (w2)	. . . . .	. . . .	P,W,M,

DATA MOVEMENT INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Move Double Floating	movd	*d1,d2*	26	Register	1	d2 ← (d1)	. . . . .	. . . .	
Move Double Floating to Longword	movdl	*d1,l2*	2e	Register	1	l2 ← (d1)	. . . . .	. . . .	
Move Longword to Double Floating	movld	*l1,d2*	2f	Register	1	d2 ← (l1)	. . . . .	. . . .	
Move Processor Register to Word	movpw	*p1,w2*	11	Register	1	w2 ← (p1)	. . . . .	. . . .	
Move Single Floating	movs	*s1,s2*	24	Register	1	s2 ← (s1)	. . . . .	. . . .	
Move Supervisor to User (priviledge)	movsu	*w1,w2*	b6 01	Macro	2	w2 ← (w1)	. . . . .	00**	S
Move Single Floating to Word	movsw	*s1,w2*	2c	Register	1	w2 ← (s1)	. . . . .	. . . .	
Move User to Supervisor (priviledge)	movus	*w1,w2*	b6 00	Macro	2	w2 ← (w1)	. . . . .	00**	S
Move Word	movw	*w1,w2*	84	Register	1	w2 ← (w1)	. . . . .	00**	
Move Word to Processor Register	movwp	*w2,p1*	10	Register	1	p1 ← (w2)	. . . . .	****	
Move Word to Single Floating	movws	*w1,s2*	2d	Register	1	s2 ← (w1)	. . . . .	. . . .	

Legend: PSW Flags Field	Traps Field
. = Flag not affected by instruction	D = Divide-by-zero
* = Flag set according to operation	I = Illegal instruction
0 = Flag set to 0	M = Memory fault
1 = Flag set to 1	P = Page fault
	R = Read protect fault
	S = Supervisor only (priviledged) instruction
	W = Write protect fault

Table B-1 Functional Instruction Set (Continued)

ARITHMETIC INSTRUCTIONS

							PSW Flags		
							FFFFF		
Instruction Name	**Syntax**		**Opcode**	**Format**	**Parcels**	**Operation**	**IVDUX**	**CVZN**	**Traps**
Add Double Floating	addd	*d1,d2*	22	Register	1	d2 ← (d2) + (d1)	**.**	....	
Add Immediate	addi	*wi,w2*	83	Immediate	2,3	w2 ← (w2) + wi	.....	****	
Add Quick	addq	*wq,w2*	82	Quick	1	w2 ← (w2) + wq	.....	****	
Add Single Floating	adds	s1,s2	20	Register	1	s2 ← (s2) + (s1)	**.**	....	
Add Word	addw	*w1,w2*	80	Register	1	w2 ← (w2) + (w1)	.....	****	
Add Word with Carry	addwc	*w1,w2*	90	Register	1	w2 ← (w2) + (w1) + C	.....	****	
Subtract Double Floating	subd	*d1,d2*	23	Register	1	d2 ← (d2) – (d1)	**.**	....	
Subtract Immediate	subi	*wi,w2*	a3	Immediate	2,3	w2 ← (w2) – wi	.....	****	
Subtract Quick	subq	*wq,w2*	a2	Quick	1	w2 ← (w2) – wq	.....	****	
Subtract Single Floating	subs	s1,s2	21	Register	1	s2 ← (s2) – (s1)	**.**	....	
Subtract Word	subw	*w1,w2*	a0	Register	1	w2 ← (w2) – (w1)	.....	****	
Subtract Word with Carry	subwc	*w1,w2*	91	Register	1	w2 ← (w2) – (w1) – C	.....	****	
Multiply Double Floating	muld	*d1,d2*	2a	Register	1	d2 ← (d2) × (d1)	*****	....	
Multiply Single Floating	muls	*s1,s2*	28	Register	1	s2 ← (s2) × (s1)	*****	....	
Multiply Word	mulw	*w1,w2*	98	Register	1	w2 ← (w2) × (w1)	.....	0*00	
Multiply Word Unsigned	mulwu	*w1,w2*	9a	Register	1	w2 ← (w2) × (w1)	.....	0*00	
Multiply Word Unsigned Extended	mulwux	*w1,l2*	9b	Register	1	l2 ← (w2) × (w1)	.....	0*00	
Multiply Word Extended	mulwx	*w1,l2*	99	Register	1	l2 ← (w2) × (w1)	.....	0*00	
Divide Double Floating	divd	*d1,d2*	2b	Register	1	d2 ← (d2) ÷ (d1)	*****	....	
Divide Single Floating	divs	*s1,s2*	29	Register	1	s2 ← (s2) ÷ (s1)	*****	....	
Divide Word	divw	*w1,w2*	9c	Register	1	w2 ← (w2) ÷ (w1)	.....	**00	D
Divide Word Unsigned	divwu	*w1,w2*	9e	Register	1	w2 ← (w2) ÷ (w1)	.....	0000	D
Negate Double Floating	negd	*d1,d2*	b4 3b	Macro	2	d2 ← (d1)	.....	....	
Negate Single Floating	negs	*s1,s2*	b4 3a	Macro	2	s2 ← (s1)	.....	....	
Negate Word	negw	*w1,w2*	93	Register	1	w2 ← (w1)	.....	****	
Modulus Word	modw	*w1,w2*	9d	Register	1	w2 ← (w2) MOD (w1)	.....	0*00	D
Modulus Word Unsigned	modwu	w1,w2	9f	Register	1	w2 ← (w2) MOD (w1)	.....	0000	D
Scale by, Double Floating	scalbd	*w1,d2*	b4 3d	Macro	2	d2 ← (d2) × 2(w1)	**.**	....	
Scale by, Single Floating	scalbs	*w1,s2*	b4 3c	Macro	2	s2 ← (s2) × 2(w1)	**.**	....	

Legend: PSW Flags Field	Traps Field
. = Flag not affected by instruction	D = Divide-by-zero
* = Flag set according to operation	I = Illegal instruction
0 = Flag set to 0	M = Memory fault
1 = Flag set to 1	P = Page fault
	R = Read protect fault
	S = Supervisor only (priviledged) instruction
	W = Write protect fault

Table B-1 Functional Instruction Set (Continued)

LOGICAL INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
And Immediate	andi	*wi,w2*	8b	Immediate	2,3	w2 ← (w2) & wi	.....	00**	
And Word	andw	*w1,w2*	88	Register	1	w2 ← (w2) & (w1)	.....	00**	
Or Immediate	ori	*wi,w2*	8f	Immediate	2,3	w2 ← (w2) | wi	.....	00**	
Or Word	orw	*w1,w2*	8c	Register	1	w2 ← (w2) |(w1)	.....	00**	
Exclusive-OR Immediate	xori	*wi,w2*	ab	Immediate	2,3	w2 ← (w2) ⊕ wi	.....	00**	
Exclusive-OR Word	xorw	*w1,w2*	a8	Register	1	w2 ← (w2) ⊕ (w1)	.....	00**	
Not Word	notw	*w1,w2*	ac	Register	1	w2 ← ~ (w1)	.....	00**	
Not Quick	notq	*wq,w2*	ae	Quick	1	w2 ← ~ wq	.....	0001	

SHIFT/ROTATE INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Shift Arithmetic Immediate	shai	*wi,w2*	38	Immediate	2	w2 ← (w2) SHA wi	.....	0***	
Shift Arithmetic Longword	shal	*w1,l2*	31	Register	1	l2 ← (l2) SHA (w1)	.....	0***	
Shift Arithmetic Longword Immediate	shali	*wi,l2*	39	Immediate	2	l2 ← (l2) SHA wi	.....	0***	
Shift Arithmetic Word	shaw	*w1,w2*	30	Register	1	w2 ← (w2) SHA (w1)	.....	0***	
Shift Logical Immediate	shli	*wi,w2*	3a	Immediate	2	w2 ← (w2) SHL wi	.....	00**	
Shift Logical Longword	shll	*w1,l2*	33	Register	1	l2 ← (l2) SHL (w1)	.....	00**	
Shift Logical Longword Immediate	shlli	*wi,l2*	3b	Immediate	2	l2 ← (l2) SHL wi	.....	00**	
Shift Logical Word	shlw	*w1,w2*	32	Register	1	w2 ← (w2) SHL (w1)	.....	00**	
Rotate Immediate	roti	*wi,w2*	3c	Immediate	2	w2 ← (w2) ROT wi	.....	00**	
Rotate Longword	rotl	*w1,l2*	35	Register	1	l2 ← (l2) ROT (w1)	.....	00**	
Rotate Longword Immediate	rotli	*wi,l2*	3d	Immediate	2	l2 ← (l2) ROT wi	.....	00**	
Rotate Word	rotw	*w1,w2*	34	Register	1	w2 ← (w2) ROT (w1)	.....	00**	

CONVERSION INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Convert Double Floating to Single	cnvds	*d1,s2*	b4 39	Macro	2	s2 ← (d1)	**.**	....	
Convert Double Floating to Word	cnvdw	*d1,w2*	b4 34	Macro	2	w2 ← (d1)	*....	....	
Convert Rounding Double to Word	cnvrdw	*d1,w2*	b4 35	Macro	2	w2 ← (d1)	*....	....	
Convert Rounding Single to Word	cnvrsw	*s1,w2*	b4 31	Macro	2	w2 ← (s1)	*....	....	
Convert Single Floating to Double	cnvsd	*s1,d2*	b4 38	Macro	2	d2 ← (s1)	*....	....	
Convert Single Floating to Word	cnvsw	*s1,w2*	b4 30	Macro	2	w2 ← (s1)	*....	....	
Convert Truncating Double to word	cnvtdw	*d1,w2*	b4 36	Macro	2	w2 ← (d1)	*....	....	
Convert Truncating Single to word	cnvtsw	*s1,w2*	b4 32	Macro	2	w2 ← (s1)	*....	....	
Convert Word to Double Floating	cnvwd	*w1,d2*	b4 37	Macro	2	d2 ← (w1)	.....	....	
Convert Word to Single Floating	cnvws	*w1,s2*	b4 33	Macro	2	s2 ← (w1)	....*	....	

Legend: PSW Flags Field	Traps Field
. = Flag not affected by instruction	D = Divide-by-zero
* = Flag set according to operation	I = Illegal instruction
0 = Flag set to 0	M = Memory fault
1 = Flag set to 1	P = Page fault
	R = Read protect fault
	S = Supervisor only (priviledged) instruction
	W = Write protect fault

Table B-1 Functional Instruction Set (Continued)

COMPARE AND TEST INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Compare Double Floating	cmpd	*d1,d2*	27	Register	1	(d2) – (d1)	.....	00**	
Compare Immediate	cmpi	*wi,w2*	a7	Immediate	2,3	(w2) – wi	.....	****	
Compare Quick	cmpq	*wq,w2*	a6	Quick	1	(w2) – wq	.....	****	
Compare Single Floating	cmps	*s1,s2*	25	Register	1	(s2) – (s1)	.....	00**	
Compare Word	cmpw	*w1,w2*	a4	Register	1	(w2) – (w1)	.....	****	
Test and Set	tsts	*wa,w2*	72,73	Address	1	w2 ← (wa), wa<31> ← 1	.....	....	P,R,W,M

CHARACTER STRING INSTRUCTIONS

Instruction Name	Syntax	Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Compare Characters r0 = length, r1 = string1, r2 = string2	cmpc	b4 0f	Macro	2	WHILE [(r0) = 0] & [((r2)) = ((r1))], r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) + 1	.....	****	P,R,M
Initialize Characters r0 = length, r1 = string1, r2 = string2	initc	b4 0e	Macro	2	WHILE (r1) ← (r2<7:0>) r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) ROT – 8	.....	....	P,W,M
Move Characters r0 = length, r1 = string1, r2 = string2	movc	b4 0d	Macro	2	WHILE (r0) = 0, (r2) ← ((r1)) r0 ← (r0) – 1, r1 ← (r1) + 1 r2 ← (r2) + 1	.....	....	P,R,W,M,

STACK MANIPULATION INSTRUCTIONS

Instruction Name	Syntax		Opcode	Format	Parcels	Operation	PSW Flags FFFFF IVDUX	CVZN	Traps
Pop Word	popw	*w1,w2*	16	Register	1	w1 ← (w1) + 4 w2 ← ((w2) – 4)	.....	....	P,R,M
Push Word	pushw	*w2,w1*	14	Register	1	w1 ← (w1) – 4 (w1) ← (w2)	.....	....	P,W,M
Restore Registers f*n*: f7	restd*n*		b4 28 . . . b4 2F	Macro	2	f*n* : f7 ← ((r15)) : ((r15) + 8 × [7-*n*]) r15 ← (r15) + 8 × [8-*n*]	.....	....	P,R,M,
Restore User Registers (priviledged)	restur	*w1*	b6 03	Macro	2	r0 : r15 ← ((w1)) : ((w1) + 60)	.....	....	P,R,M,S
Restore Registers r*n*: r14	restw*n*		b4 10 . . . b4 1C	Macro	2	r*n* : r14 ← ((r15)) : ((r15) + 4 × [14-*n*]), r15 ← (r15) + 4 × [15-*n*]	.....	....	
Save Registers f*n*: f7	saved*n*		b4 20 . . . b4 27	Macro	2	(r15) – 8 × [8-*n*] : (r15) – 8 ← (f*n*) : (f7) r15 ← (r15) – 8 × [8-*n*]	.....	....	P,W,M
Save User Registers (priviledged)	saveur	*w1*	b6 02	Macro	2	(w1) – 4 : (w1) – 64 ← (r15) : (r0)	.....	....	P,W,M,S
Save Registers r*n*: r14	savew*n*		b4 00 . . . b4 0C	Macro	2	(r15) – 4 × [15-*n*] : (r15) – 4 ← (r*n*) : ← (r14), r15 ← (r15)-8 × [8-*n*]	.....	....	P,W,M

Legend: PSW Flags Field
. = Flag not affected by instruction
* = Flag set according to operation
0 = Flag set to 0
1 = Flag set to 1

Traps Field
D = Divide-by-zero
I = Illegal instruction
M = Memory fault
P = Page fault
R = Read protect fault
S = Supervisor only (priviledged) instruction
W = Write protect fault

Table B-1 Functional Instruction Set (Continued)

CONTROL INSTRUCTIONS

Instruction Name	**Syntax**		**Opcode**	**Format**	**Parcels**	**Operation**	**PSW Flags FFFFF IVDUX**	**PSW Flags CVZN**	**Traps**
Branch Conditional	b*	*ha*	48,49	Address	1-4	IF cond, PC ← ha	.....	....	
Branch Floating Exception	bf*	*ha*	4c,4d	Address	1-4	IF cond, PC ← ha	.....	....	
Call Subroutine	call	*w2,ha*	44,45	Address	1-4	w2 ← (w2) − 4, (w2) ← (PC), PC ← ha	.....	....	P,W,M,
Call Supervisor	calls	*bb*	12	Control	1	trap 400 + 8 × bb<7:0>	.....	....	
No Operation	noop	*bb*	00	Control	1	none	.....	....	
Return From Subroutine	ret	*w2*	13	Register	1	PC ← ((w2)), w2 ← (w2) + 4	.....	....	P,R,M
Return From Interrupt (priviledged)	reti	*w1*	b6 04	Macro	2	Restore SSW, PSW and PC	.....	...	S
Trap on Floating Unordered	trapf*n*		b4 3e	Macro	2	IF PSW<ZN> indicated unordered, illegal instruction trap	.....	....	I
Wait for Interrupt (priviledged)	wait		b6 05	Macro	2	Wait for interrupt	.....	....	S

Legend: PSW Flags Field	Traps Field
. = Flag not affected by instruction	D = Divide-by-zero
* = Flag set according to operation	I = Illegal instruction
0 = Flag set to 0	M = Memory fault
1 = Flag set to 1	P = Page fault
	R = Read protect fault
	S = Supervisor only (priviledged) instruction
	W = Write protect fault

Table B-2 Instruction Opcode/Mnemonic Summary

MSB	LSB 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	noop															
1	movwp	movpw	calls	ret	pushw		popw									
2	adds	subs	addd	subd	movs	cmps	movd	cmpd	muls	divs	muld	divd	movsw	movws	movdl	movld
3	shaw	shal	shlw	shll	rotw	rotl			shai	shali	shli	shlli	roti	rotli		
4					call				b* see Table 2-7				bf* see Table 2-8			
5																
6	loadw		loada		loads		loadd		loadb		loadbu		loadh		loadhu	
7	storw		tsts		stors		stord		storb				storh			
8	addw		addq	addi	movw		loadq	loadi	andw			andi	orw			ori
9	addwc	subwc		negw					mulw	mulwx	mulwu	mulwux	divw	modw	divwu	modwu
A	subw		subq	subi	cmpw		cmpq	cmpi	xorw			xori	notw		notq	
B					macros see Table B-3		priviledged macros see Table B-4									
C																
D																
E																
F																

Table B-3 Macro Instruction Code Field (opcode B4)

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	savew0	savew1	savew2	savew3	savew4	savew5	savew6	savew7	savew8	savew9	savew10	savew11	savew12	movc	initc	cmpc
1	restw0	restw1	restw2	restw3	restw4	restw5	restw6	restw7	restw8	restw9	restw10	restw11	restw12			
2	saved0	saved1	saved2	saved3	saved4	saved5	saved6	saved7	restd0	restd1	restd2	restd3	restd4	restd5	restd6	restd7
3	cnvsw	cnvrsw	cnvtsw	cnvws	cnvdw	cnvrdw	cnvtdw	cnvwd	cnvsd	cnvds	negs	negds	scalbs	scalbd	trapfn	loadfs

Table B-4 Priviledged Macro Instruction Code Field (opcode B6)

MSB \ LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	movus	movsu	saveur	restur	reti	wait										

FAIRCHILD
A Schlumberger Company

Advanced Processor
Division

4001 Miranda Ave.
Palo Alto, CA 94304

800 423-5516

In California:
415 858-4249