Instruction Set 9

CHAPTER 9
INSTRUCTION SET

This chapter provides an overview of the instruction set for the 80960SA/SB processor. Included is a discussion of the instruction format, a summary of the instruction groups and the instructions in each group.

This chapter gives detailed descriptions of each of the instructions. The instructions are listed in alphabetical order. Included for each instruction is the assembly-language format, the action taken when the instruction is executed, and examples of how the instruction might be used.

Appendix C provides a detailed description of the factors that affect instruction timing. It also gives the number of clock cycles required for each instruction.

INSTRUCTION FORMATS

Instructions are described in this reference manual in two formats: assembly language and machine level.

Assembly-Language Format

Throughout most of this manual, the instructions are referred to by their assembly-language mnemonics. For example, the add ordinal instruction is referred to as the **addo** instruction.

An assembly-language statement consists of an instruction mnemonic, followed by from 0 to 3 operands, separated by commas. The following example shows the assembly-language statement for the **addo** instruction:

```
addo g5, g9, g7
```

Here, the ordinal operands in global registers g5 and g9 are added together and the result is stored in g7.

A detailed description of the nomenclature used to describe assembly-language instructions is given in this chapter.

Machine Formats

At the machine level of the processor, all instructions are word aligned. Most of the instructions are one word long, although some memory-addressing modes make use of a two-word format.

There are four instruction formats: register (REG), compare and branch (COBR), control (CTRL), and memory (MEM). Each instruction uses one of these formats, which is determined by the opcode field of the instruction.

The machine-level formats for the instructions are described in detail in Appendix B.

INSTRUCTION GROUPS

The 80960SA/SB processor implements all the instructions in the 80960 instruction set, which includes all of the data-movement, arithmetic, logical, and program-control instructions commonly found in computer architectures. The 80960SB processor also includes a set of floating-point instructions and several instructions to handle architectural extensions found in the processor.

The 80960 instruction set is made up of the following groups of instructions:

- Data Movement
- Arithmetic (Ordinal and Integer)
- Logical
- Bit, Bit Field, and Byte
- Comparison
- Branch
- Call/Return
- Fault
- Debug
- Atomic
- Processor Management
- Synchronous Move and Load

The instruction-set extensions found in the 80960SB processor include the following groups of instructions:

- Integer-to-Real Conversion
- Floating Point
- Decimal

Tables 9-1 and 9-2 give a summary of the 80960 instructions and the 80960SA/SB instruction-set extensions, respectively. The actual number of instructions is greater than those shown in this list, because for some operations, several different instructions are provided to handle different operand sizes, data types, or branch conditions.

Table 9-1: Summary of the 80960 Instruction Set

Data Movement	**Arithmetic**	**Logical**	**Bit, Bit Field, and Byte**
Load Store Move Load Address	Add Subtract Multiply Divide Extended Multiply Extended Divide Remainder Modulo Shift Rotate	And Not And And Not Or Exclusive Or Not Or Or Not Nor Exclusive Nor Not Nand	Set Bit Clear Bit Not Bit Check Bit Alter Bit Scan For Bit Scan Over Bit Extract Modify Scan Byte For Equal
Comparison	**Branch**	**Call/Return**	**Fault**
Compare Conditional Compare Compare and Increment Compare and Decrement	Unconditional Branch Conditional Branch Compare and Branch Test Condition Code	Call Call Extended Call System Return Branch and Link	Conditional Fault Synchronize Faults
Debug	**Atomic**	**Processor**	
Modify Trace Controls Mark Force Mark	Atomic Add Atomic Modify	Flush Local Registers Modify Arithmetic Controls Modify Process Controls	

Table 9-2: Summary of the 80960SA/SB Instruction-Set Extensions

Conversion	Floating Point	Synchronous	Decimal
Convert Real to Integer Convert Integer to Real	Move Real Add Subtract Multiply Divide Remainder Scale Round Square Root Sine Cosine Tangent Arctangent Log Log Binary Log Natural Exponent Classify Copy Real Extended Compare	Synchronous Load Synchronous Move	Move Add with Carry Subtract with Carry

The following sections give a brief overview of the instructions in each of these groups. The floating-point instructions are described in Chapter 10.

DATA MOVEMENT

The data movement instructions include those instructions that move data from memory to the global and local registers; that move data from the global and local registers to memory; and that move data among these registers.

Load

The load instructions (listed below) copy bytes or words from memory to a selected register or group of registers:

ld	load
ldob	load byte ordinal
ldos	load short ordinal
ldib	load byte integer
ldis	load short integer
ldl	load long
ldt	load triple
ldq	load quad

For the **ld**, **ldob**, **ldos**, **ldib**, and **ldis** instructions, a memory address and a register are specified in the instruction and the value at the memory address is copied into the register. Zero and sign extending is performed automatically for byte and short (half-word) operands.

The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes from memory into successive registers.

NOTE

When using the load, store, and move instructions that move 8, 12, or 16 bytes at a time, the rules for register alignment must be followed. Refer to the section in Chapter 2 titled "Register Alignment" for a discussion of these rules.

Store

For each load instruction there is a corresponding store instruction (listed below), which copies bytes or words from a selected register or group of registers to memory:

st	store
stob	store byte ordinal
stos	store short ordinal
stib	store byte integer
stis	store short integer
stl	store long
stt	store triple
stq	store quad

For the **st**, **stob**, **stos**, **stib**, and **stis** instructions, a register and memory address are specified in the instruction and the value in the register is copied into memory. For the byte and short instructions, the value in the register is automatically reformatted for the shorter memory location. For the **stib** and **stis** instructions, this reformatting can lead to overflow if the register value is too large to be represented in the shorter memory location.

The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes from successive registers into memory.

Move

The move instructions, listed below, copy data from a register or group of registers to another register or group of registers.

mov	move word
movl	move long word
movt	move triple word
movq	move quad word

These move instructions can only be used to move data among the global and local registers. A set of move-real instructions (**movr**, **movrl**, and **movre**) are provided for moving real number values between the global and local registers and the floating-point registers. The move-real instructions are described in Chapter 10.

Load Address

The **lda** instruction computes an effective address in the address space from an operand presented in one of the addressing modes. A common use of this instruction is to load a constant into a register.

ARITHMETIC

Table 9-3 lists all the arithmetic operations for which the 80960SA/SB processor provides instructions and the data types that the instructions operate on. An "X" in this table indicates that the 80960SA/SB provides an instruction for the specified operation and data type; an "E" indicates that an 80960SA/SB instruction-set extension provides an instruction for the specified operation and data type. An "E*" indicates that the specified operation can be performed on the specified data type using 80960SB extended-instruction-set instructions, but that a unique instruction for this operation is not provided. For example, a specific instruction is not provided to add two extended-real values. However, this operation can be carried out with either the add real (**addr**) or the add long real (**addrl**) instruction.

With two exceptions, all the processor's arithmetic operations are carried out on operands in registers. The processor does not provide instructions that perform arithmetic operations on operands in memory.

The two instructions that are exceptions are the **atadd** (atomic add) and **atmod** (atomic modify) instructions, which are discussed later in this chapter.

A summary of the arithmetic instructions for real (floating-point) data types is provided in Chapter 10. The following sections describe the arithmetic instructions for ordinal and integer data types.

Add, Subtract, Multiply, and Divide

The following instructions perform add, subtract, multiply, or divide operations on integers and ordinals:

addi	add integer
addo	add ordinal
subi	subtract integer
subo	subtract ordinal
muli	multiply integer
mulo	multiply ordinal
divi	divide integer
divo	divide ordinal

These instructions perform operations on one-word operands in registers and store the results in a register.

Table 9-3: Arithmetic Operations

Arithmetic Operations	Integer	Ordinal	Real	Long Real	Extended Real
Add	X	X	E	E	E*
Subtract	X	X	E	E	E*
Multiply	X	X	E	E	E*
Divide	X	X	E	E	E*
Remainder	X	X	E	E	E*
Modulo	X				
Shift Left	X	X			
Shift Right	X	X			
Shift Right Dividing	X				
Scale			E	E	E*
Round			E	E	E*
Square Root			E	E	E*
Sine			E	E	E*
Cosine			E	E	E*
Tangent			E	E	E*
Arctangent			E	E	E*
Exponent			E	E	E*
Log			E	E	E*
Log Binary			E	E	E*
Log Epsilon			E	E	E*
Classify			E	E	E*
Copy Sign					E
Copy Reversed Sign					E

Extended Arithmetic

The following four instructions are provided to support extended arithmetic operations to be performed (i.e., arithmetic operations on operands greater than one word in length):

addc	add ordinal with carry
subc	subtract ordinal with carry
emul	extended multiply
ediv	extended divide

The **addc** and **subc** instructions add or subtract two words (contained in registers) plus a condition code bit (used as a carry bit). If the result has a carry, the carry bit in the condition code is set. Also, a second condition code bit is set if the operation would have resulted in an integer overflow condition. (The three-bit condition code is contained in the arithmetic controls as described in Chapter 2.)

These instructions treat the operands as ordinals; however, the indication of overflow in the condition code facilitates a software implementation of extended-integer arithmetic.

The **emul** instruction multiplies two ordinals (each contained in a register), producing a long ordinal result (stored in two registers). The **ediv** instruction divides a long ordinal by an ordinal, producing an ordinal quotient and an ordinal remainder.

Remainder and Modulo

The following instructions divide one operand by another and retain the remainder of the operation:

remi	remainder integer
remo	remainder ordinal
modi	modulo integer

The difference between the remainder and modulo instructions lies in the sign of the result. For the **remi** and **remo** instructions, the result has the same sign as the dividend; for the **modi** instruction, the result has the same sign as the divisor.

Shift and Rotate

The processor provides the following five shift instructions:

shlo	shift left ordinal
shro	shift right ordinal
shli	shift left integer
shri	shift right integer
shrdi	shift right dividing integer

These instructions shift the operand a specified number of bits to the left or to the right. Bits shifted beyond the register boundary are discarded.

The **shlo** instruction shift zeros in from the least-significant bit, and the **shro** instruction shifts zeros in from the most-significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

The **shli** instruction shifts zeros in from the least-significant bit; if the bits shifted out are not the same as the sign bit, an overflow fault is generated.

The **shri** instruction performs a conventional arithmetic shift-right operation by shifting the sign bit in from the most-significant bit. When this instruction is used to divide a negative integer operand by the power of 2, however, it produces an incorrect quotient. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2.

The **rotate** instruction rotates the bits of the operand to the left (toward higher significance) by a specified number of bits. Bits shifted beyond the left boundary of the register (bit 31) appear at the right boundary (bit 0).

LOGICAL

The following instructions perform bitwise Boolean operations on the specified operands:

and	A and B
notand	(not A) and B
andnot	A and (not B)
xor	not (A = B)
or	A or B
nor	not (A or B)
xnor	A = B
not	not A
notor	(not A) or B
ornot	A or (not B)
nand	not (A and B)

BIT AND BIT FIELD

The bit instructions perform operations on a specific bit in an ordinal operand or on a bit field.

Bit Operations

The following instructions operate on a specified bit:

setbit	set bit
clrbit	clear bit
notbit	not bit
chkbit	check bit
alterbit	alter bit
scanbit	scan for bit
spanbit	span over bit

The **setbit**, **clrbit**, and **notbit** instructions set, clear, or complement (toggle) a specified bit in an ordinal.

The **chkbit** instruction causes the condition-code bits to be set according to the state of a specified bit in a register. The condition code is set to 010_2 if the bit is set and 000_2 otherwise.

The **alterbit** instruction alters the state of a specified bit in an ordinal according to the condition code. If the condition code is 010_2, the bit is set; if the condition code is 000_2, the bit is cleared.

The **scanbit** and **spanbit** instructions find the most significant set bit and clear bit, respectively, in an ordinal.

Bit-Field Operations

There are two bit field instructions **extract** and **modify**. The **extract** instruction converts a specified bit field, taken from an ordinal value, into an ordinal value. In essence, this instruction shifts a bit field in a register to the right and fills in the bits to the left of the bit field with zeros.

The **modify** instruction copies bits from one register, under control of a mask, into another register. Only the unmasked bits in the destination register are modified.

BYTE OPERATIONS

The **scanbyte** instruction performs a byte-by-byte comparison of two ordinals to determine if any two corresponding bytes are equal. The condition code is set according to the results of the comparison.

CONVERSION

Data can be converted from one length to another by means of the load and store instructions. For example, the **ldis** instruction loads a short integer from memory to a register and automatically converts the integer from a half word to a full word.

The 80960SB extended instruction set provides instructions to perform conversions between integer and real data types. These instructions are described in Chapter 10.

COMPARISON

The processor provides several types of instructions that are used to compare two operands. The following sections describe the compare instructions for ordinal and integer data types. The compare instructions for real data types are discussed in Chapter 10.

Compare and Conditional Compare

The compare instructions listed below compare two operands, then set the condition-code bits in the arithmetic controls according to the results.

cmpi	compare integer
cmpo	compare ordinal
concmpi	conditional compare integer
concmpo	conditional compare ordinal

The condition-code bits are set to indicate whether one operand is less than, equal to, or greater than the other operand. (Refer to the section in Chapter 2 titled "Functions of the Arithmetic-Controls Bits" for a discussion of meanings of the condition-code bits for conditional operations.)

The **cmpi** and **cmpo** instructions simply compare the two operands and set the condition-code bits accordingly.

The **concmpi** and **concmpo** instructions first check the status of bit 2 of the condition code. If it is not set, the operands are compared as with the **cmpi** and **cmpo** instructions. If bit 2 is set, no comparison is performed and the condition-code bits are not changed.

The conditional compare instructions are provided specifically to optimize two-sided range comparisons to check if A is between B and C (i.e., $B \leq A \leq C$). Here, a compare instruction (**cmpi** or **cmpo**) is used to check one side of the range (e.g., $A \geq B$) and a conditional compare instruction (**concmpi** or **concmpo**) is used to check the other side (e.g., $A \leq C$) according to the result of the first comparison.

Compare and Increment or Decrement

The following instructions compare two operands, set the condition-code bits according to the results, then increment or decrement one of the operands:

cmpinci	compare and increment integer
cmpinco	compare and increment ordinal
cmpdeci	compare and decrement integer
cmpdeco	compare and decrement ordinal

These instructions are intended for use at the end of iterative loops.

BRANCH

The branch instructions allow the direction of program flow to be changed by explicitly modifying the IP. The processor provides three types of branch instructions:

- unconditional branch
- conditional branch
- compare and branch

The processor also provides a set of instructions for testing the condition code flags of the arithmetic controls. These instructions can be used in conjunction with the compare instructions and the branch instructions as an alternate means of performing conditional branch, and compare and branch operations.

Most of the branch instructions specify the target IP by specifying a signed displacement to be added to the current IP. Other branch instructions specify the memory address of the target IP using one of the processor's addressing modes. This latter group of instructions are called extended-addressing instructions (e.g., branch extended, branch and link extended).

Unconditional Branch

The following four instructions are used for unconditional branching:

b	Branch
bx	Branch Extended
bal	Branch and Link
balx	Branch and Link Extended

The **b** and **bx** instructions cause program execution to jump to the specified target IP. As described later in this chapter, these two instructions perform the same function; however, they use different machine-level instruction formats.

The **bal** and **balx** instructions store the address of the next instruction in a specified register, then jump to the specified target IP. (For the **bal** instruction, the RIP is automatically stored in register g14; for the **balx** instruction the location of the RIP is specified with an instruction operand.) As described in Chapter 4, the branch and link instructions provide a method of performing procedure calls that does not use the processor's call/return mechanism. Here, the saved instruction address is used as a return IP.

The **bx** and **balx** instructions can be made IP-relative by using the IP with displacement addressing mode.

Conditional Branch

With the conditional branch (branch if) instructions, the processor checks the condition-code bits in the arithmetic controls. If these bits match the value specified with the instruction, the processor jumps to the target IP. These instructions use the displacement plus IP method of specifying the target IP:

be	branch if equal
bne	branch if not equal
bl	branch if less
ble	branch if less or equal
bg	branch if greater
bge	branch if greater or equal
bo	branch if ordered
bno	branch if unordered

(Refer to the section in Chapter 2 titled "Functions of the Arithmetic-Controls Bits" for a discussion of meanings of the condition-code bits for conditional operations.)

The **bo** and **bno** instructions refer to comparisons of real numbers. Ordered and unordered real numbers are described in Chapter 10.

Compare and Branch

The compare and branch instructions compare two operands, then branch according to the results. There are three subtypes of instructions in this group: compare integer, compare ordinal, and check bit:

cmpibe	compare integer and branch if equal
cmpibne	compare integer and branch if not equal
cmpibl	compare integer and branch if less
cmpible	compare integer and branch if less or equal
cmpibg	compare integer and branch if greater
cmpibge	compare integer and branch if greater or equal
cmpibo	compare integer and branch if ordered
cmpibno	compare integer and branch if unordered
cmpobe	compare ordinal and branch if equal
cmpobne	compare ordinal and branch if not equal
cmpobl	compare ordinal and branch if less
cmpoble	compare ordinal and branch if less or equal
cmpobg	compare ordinal and branch if greater
cmpobge	compare ordinal and branch if greater or equal
bbs	check bit and branch if set
bbc	check bit and branch if clear

With the compare-ordinal-and-branch and compare-integer-and-branch instructions, two operands are compared and the condition-code bits are set, as with the compare instructions described earlier in this chapter. A conditional branch is then executed as with the conditional branch (branch if) instructions.

With the check-bit-and-branch instructions, one operand specifies a bit to be checked in the other operand. The condition-code bits are set according to the state of the specified bit (i.e., 010_2 if the bit is set and 000_2 if the bit is clear). A conditional branch is then executed according to the setting of the condition-code bits.

Test Condition Codes

The following test instructions allow the state of the condition-code bits to be tested:

teste	test if equal
testne	test if not equal
testl	test if less
testle	test if less or equal
testg	test if greater
testge	test if greater or equal
testo	test if ordered
testno	ttest if unordered

These instructions cause a TRUE (1_2) to be stored in a destination register if the condition code matches the condition specified with the instruction. Otherwise, a FALSE (0_2) is stored in the register.

CALL AND RETURN

The processor offers an on-chip call/return mechanism for making procedure calls to local procedures and kernel procedures. This call/return mechanism is described in detail in Chapter 4. The following four instructions are provided to support this mechanism.

call	call
callx	call extended
calls	call system
ret	return

The **call** and **callx** instructions call local procedures. The **call** instruction specifies the target procedure (the first instruction of the procedure) by adding a signed displacement to the IP. The **callx** instruction uses extended addressing, as described for the **bx** and **balx** instructions, to specify the target procedure. For both of these instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

The **calls** instruction operates similarly to the **call** and **callx** instructions, except that it gets its target procedure address from the system procedure table. An index number included as an operand in the instruction provides an entry point into the procedure table.

Depending on the type of entry being pointed to in the procedure table, the **calls** instruction can cause a supervisor call to be executed. A supervisor call causes the processor to switch to the supervisor stack and to switch to supervisor mode. The supervisor call is described in detail in Chapter 4.

The **ret** instruction performs a return from a called procedure to the calling procedure (the procedure that made the call). This instruction obtains its target IP (return IP) from linkage information that was saved for the calling procedure. The **ret** instruction is used to return from local and supervisor calls and from implicit calls to interrupt and fault handlers.

CONDITIONAL FAULTS

Generally, the processor generates faults automatically as the result of certain operations. Fault handling routines are then invoked to handle the various types of faults without explicit intervention by the currently running program. (Faults are discussed in detail in Chapter 6.)

The following conditional fault instructions permit a fault to be generated explicitly according to the state of the condition-code bits:

faulte	fault if equal
faultne	fault if not equal
faultl	fault if less
faultle	fault if less or equal
faultg	fault if greater
faultge	fault if greater or equal
faulto	fault if ordered
faultno	fault if unordered

The synchronize faults (**syncf**) instruction is provided to force all faults to be precise in situations when the processor is executing two instructions in parallel. The function and use of this instruction is discussed in detail in the section in Chapter 6 titled "Precise and Imprecise Faults."

NOTE

In the 80960SA/SB implementation of the i960 architecture, the **syncf** instruction behaves as a no-op (i.e., it performs no-operation). This is because in this implementation all faults are required to be precise.

DEBUG

The processor supports debugging and monitoring of program activity through the use of trace events. The following instructions support these debugging and monitoring tools:

modtc	modify trace controls
mark	mark
fmark	force mark

The trace functions are controlled through the processor's trace controls bits. Some of these bits allow various types of tracing to be enabled or disabled. Other bits act as flags to indicate when an enabled trace event has been detected. (Trace controls are described in detail in Chapter 7.)

The **modtc** instruction permits the trace controls bits to be modified.

The **mark** instruction causes a breakpoint trace event to be generated if the breakpoint trace mode is enabled. The **fmark** instruction generates a breakpoint trace independent of the state of the breakpoint trace mode flag. The latter two instructions allow a breakpoint to be placed anywhere in a program.

ATOMIC INSTRUCTIONS

The atomic instructions perform read-modify-write operations on operands in memory. They insure that when one atomic operation is performed on a specified block of memory, it will be completed before another atomic operation can be performed on the same block. These instructions are particularly useful in systems where several agents have access to system memory.

There are two atomic instructions: atomic add (**atadd**) and atomic modify (**atmod**). The **atadd** instruction causes an operand to be added to the value in the specified memory location. The **atmod** causes bits in the specified memory location to be modified under control of a mask.

PROCESSOR MANAGEMENT

The processor provides several instructions for use in controlling processor-related functions.

The **modpc** instruction provides a method of reading and modifying the contents of the process controls.

In certain instances, it is necessary to insure that the contents of the local-register save area of the stack frames are the same as the local registers. The flush local registers instruction (**flushreg**) automatically stores the contents of all the local register sets, except the current set, in the register save area of their associated stack frames.

The arithmetic controls cannot be addressed with the load, move, and store instructions or the bit instructions. Instead, special instructions are provided for this purpose.

The modify arithmetic controls instruction (**modac**) permits bits in the arithmetic controls register to be modified under the control of a mask.

80960SA/SB NON-FLOATING-POINT INSTRUCTION-SET EXTENSIONS

The following non-floating-point instructions are extensions to the i960 architecture instruction set. The synchronous load and move instructions are provided in both the 80960SA and 80960SB processors; the decimal instructions are provided only in the 80960SB processor.

Synchronous Load and Move

The processor's store instructions are executed asynchronously with the memory controller. Once the processor sends data out on its bus for storage in main memory, it continues with the next instruction in the instruction stream, assuming that its bus control logic will carry out the operation.

The 80960SA/SB processor provides four special instructions for performing memory operations that perform store and move operations synchronously with memory.

The synchronous load instruction (**synld**) loads a word from memory into a register. When this instruction is performed, the processor waits until a condition code bit is set in the arithmetic controls, indicating that the operation has been completed, before it begins executing the next instruction. The **synld** instruction is used primarily to read the contents of the interrupt-control register, as described in Chapter 5.

The synchronous move instructions (**synmov**, **synmovl**, and **synmovq**) perform synchronous moves of data from one location in memory to another. These instructions are used primarily for sending IAC messages, as described in Chapter 11.

Decimal

The following three instructions are provided for use in decimal-arithmetic algorithms:

dmovt	move and test decimal
daddc	decimal add with carry
dsubc	decimal subtract with carry

These instructions operate on 32-bit decimal operands that contain an 8-bit, ASCII-coded decimal in the least-significant byte of the word (as shown in Figure 8-2).

The **dmovt** instruction moves a decimal operand from one register to another and tests the least significant byte of the operand to determine if it is a decimal digit (0 to 9). It sets the condition code according to the results of the test: 010_2 if the operand contains a decimal digit and 000_2 otherwise.

The **daddc** and **dsubc** instructions operate similarly to the **addc** and **subc** instructions. They add or subtract two decimal digits plus bit 1 of the condition code (used as a carry-in bit). If the operation produces a decimal carry, the condition code is set accordingly. The subtraction operation is carried out in 10's complement arithmetic.

These instructions can be used iteratively to add or subtract decimal values of any length.

With the 80960SB processor, the most efficient method of multiplying or dividing decimal numbers is to convert them into extended-real numbers and use the **mulr** and **divr** instructions. Decimal values of up to 18 decimal digits can be handled with this technique.

INSTRUCTION REFERENCE

The rest of this chapter provides detailed information about each of the instructions for the 80960SA/SB processor. To provide quick access to information on a particular instruction, the instructions are listed alphabetically by assembly-language mnemonic. An explanation of the format and abbreviations used in this chapter is given in the following paragraphs.

The information in the rest of this chapter is oriented toward programmers who are writing assembly-language code for the 80960SA/SB processor. The information provided for each instruction includes the following:

- Alphabetic reference
- Assembly-language mnemonic and name
- Assembly-language format
- Description of the instruction's operation
- Action the instruction carries out when executed (generally presented in the form of an algorithm)
- Faults that can occur during execution
- Assembly-language example
- Opcode and instruction format
- Related instructions

Additional information about the instruction set can be found in the following chapters and appendices in this manual:

- Chapter 9 -- The beginning of this chapter provides a summary of the instruction set by group and description of the assembly-language instruction format
- Appendix A -- Instruction Quick Reference
- Appendix B -- Machine-Level Instruction Formats
- Appendix C -- Instruction Timing

NOTATION

To simplify the presentation of information about the instructions, a simple notation has been adopted in this chapter. The following paragraphs describe this notation.

Alphabetic Reference

The instructions are listed alphabetically by assembly-language mnemonic. If several instructions are related and fall together alphabetically, they are described as a group on a single page.

The reference at the top of each page gives the assembly-language mnemonics for the instructions covered on that page (e.g., **subc**). Occasionally, there are so many instructions covered on the page that it is not practical to give all the mnemonics in the page reference. In these cases, the name of the instruction group is given in capital letters (e.g., BRANCH or FAULT IF).

A box around the alphabetic reference, such as:

addr, addrl

indicates the instruction or group of instructions are extensions to the i960 architecture instruction set. Most boxed extensions apply only to the 80960SB processor; extensions that apply to the 80960SA and 80960SB processors are boxed with a double-line border.

Mnemonic

The Mnemonic section gives the complete mnemonic (in bold-face type) and instruction name for each instruction covered on the page, for example:

subi Subtract Integer

Format

The Format section gives the assembly-language format of the instruction and the type of operands allowed. The format is given in two or three lines. The following is an example of a two line format:

sub*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

The first line gives the assembly-language mnemonic (bold-face type) and the operands (italics). When the format is used for two or more instructions, an abbreviated form of the mnemonic is used. The "*" sign at the end of the mnemonic indicates that the mnemonic has been abbreviated.

The operand names are designed to describe the functions of the operands (e.g., *src*, *len*, *mask*).

The second line of the format shows what is allowed to be entered for each operand. The notation used on this line is as follows:

reg Global (g0 ... g15) or local (r0 ... r15) register

freg Global (g0 ... g15) or local (r0 ... r15) register, or floating-point (fp0 ... fp3) register, where the registers contain floating-point numbers

lit Integer or ordinal literal of the range 0 ... 31

flit Floating-point literal of value 1.0 or 0.0

disp Signed displacement of range -2^{22} ... $(2^{22} - 1)$

mem Address defined with the full range of addressing modes

In some cases, a third line will be added to show specifically what will be in a register or memory location. For example, it may be useful to know that a register is to contain an address. The notation used in this line is as follows:

addr Address

efa Effective address

Description

The Description section describes what the instruction does and the functions of the operands. It also gives programming hints when appropriate.

Action

The Action section gives an algorithm written in a pseudo-code that describes in detail what actions the processor takes when executing the instruction and the precise order of these actions. The main purpose of this section is to show the possible side effects of the instruction. The following is an example of the action algorithm for the **alterbit** instruction:

if (AC.cc **and** 010_2) = 0
  then *dst* ← *src* **and not** (2^(*bitpos* **mod** 32));
  else *dst* ← *src* **or** 2^(*bitpos* **mod** 32);
end if;

In these action statements, the term AC.cc means the condition-code bits in the arithmetic controls.

Faults

The Faults section lists the faults that can be signaled as the result of execution of the instruction. Faults listed with all-capital letters refer to a group of faults; faults listed with initial-capital letters refer to a specific fault.

All instructions can signal a group of general faults which are referred to as STANDARD FAULTS. The standard faults include the trace-instruction and machine-bad-access faults. In addition, for all instructions that have a MEM machine-format (such as, load, store, call extended), the invalid-opcode and operation-unimplemented faults are standard faults.

The following list shows the various fault groups and the individual faults in each group:

TRACE FAULTS
- Instruction Trace
- Branch Trace
- Call Trace
- Return Trace
- Prereturn Trace
- Supervisor Trace
- Breakpoint Trace

OPERATION
- Invalid Opcode
- Unimplemented
- Invalid Operand

ARITHMETIC
- Integer Overflow
- Arithmetic Zero-Divide

FLOATING-POINT
- Floating Overflow
- Floating Underflow
- Floating Invalid-Operation
- Floating Zero-Divide
- Floating Inexact
- Floating Reserved-Encoding

CONSTRAINT
- Constraint Range
- Privileged

PROTECTION
- Length

TYPE
- Type Mismatch

Example

The Example section gives an assembly-language example of an application of the instruction.

Opcode and Instruction Format

The Opcode and Instruction Format section gives the opcode and machine language instruction format for each instruction, for example:

subi 593 REG

The opcode is given in hexadecimal format.

The machine language format is one of four possible formats: REG, COBR, CTRL, and MEM. Refer to Appendix B for more information on the machine-language instruction formats.

See Also

The See Also section gives the mnemonics of related instructions, which can then be looked up alphabetically in this chapter for comparison. For instructions that are grouped on one page (such as **addr** and **addrl**) only the first mnemonic is given.

INSTRUCTIONS

This section contains reference information on the processor's instructions. It is arranged alphabetically by instruction or instruction group.

addc

Mnemonic: **addc** Add Ordinal With Carry

Format:

addc	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Adds the *src2* and *src1* values, and bit 1 of the condition code (used here as a carry in), and stores the result in *dst*. If the ordinal addition results in a carry, bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Regardless of the results of the addition, bits 0 and 1 of the arithmetic controls are always written.

The **addc** instruction can be used for either ordinal or integer arithmetic. The instruction does not distinguish between ordinal and integer source operands. Instead, the processor evaluates the result for both data types and sets bits 0 and 1 of the condition code accordingly.

An integer overflow fault is never signaled with this instruction.

Action:

```
# Let the value of the condition code be xCx.
dst ← src2 + src1 + C;
AC.cc ← 0CV2;
# C is carry from ordinal addition.
# V is 1 if integer addition would have generated an overflow.
```

Faults: STANDARD

Example:

```
# Example of double-precision arithmetic
# Assume 64-bit source operands
# in g0,g1 and g2,g3
cmpo 1, 0          # clears Bit 1 (carry bit) of
                   # the AC.cc
addc g0, g2, g0    # add low-order 32 bits;
                   # g0  ← g2 + g0 + Carry Bit
addc g1, g3, g1    # add high-order 32 bits;
                   # g1  ← g3 + g1 + Carry Bit
# 64-bit result is in g0, g1
```

Opcode: **addc** 5B0 REG

See: **addo, subc**

addi, addo

Mnemonic:

addi	Add Integer
addo	Add Ordinal

Format:

add*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Adds the *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← *src2* + *src1*;

Faults:

STANDARD	Refer to discussion of faults at the beginning of this chapter.
Integer Overflow	Result is too large for destination format. This fault is signaled only when executing the **addi** instruction and if both of the following conditions are met: (1) the integer-overflow mask in the arithmetic-controls registers is clear and (2) the source operands have like signs and the sign of the result operand is different than the signs of the source operands.

Example:

```
addi r4, g5, r9    # r9 ← g5 + r4
```

Opcode:

addi	591	REG
addo	590	REG

See Also: **addc, addr, subi, subo**

addr, addrl

Mnemonics:

addr	Add Real	
addrl	Add Long Real	

Format:

addr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Adds the *src2* and *src1* values and stores the result in *dst*.

For the **addrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
		-∞	**-F**	**-0**	**+0**	**+F**	**+∞**	**NaN**
Src2	**-∞**	-∞	-∞	-∞	-∞	-∞	*	NaN
	-F	-∞	-F	src2	src2	±F or ±0	+∞	NaN
	-0	-∞	src1	-0	±0	src1	+∞	NaN
	+0	-∞	src1	±0	+0	src1	+∞	NaN
	+F	-∞	±F or ±0	src2	src2	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.

When the sum of two operands with opposite signs is zero, the result is +0, except for the round toward -∞ mode, in which case, the result is -0. When zero is added to itself (e.g. *src1* + *src1*, where *src1* is 0), the result retains the sign of the source.

Action: *dst* ← *src2* + *src1*;

addr, addrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Normalized result is too small for destination format.
Floating Invalid Operation	Source operands are infinities of unlike sign.
	One or more operands is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.
	Floating overflow occurred and the overflow exception was masked.

Example:

```
addrl g6, g8, fp3    #fp3 ← g6,g7 + g8,g9
```

Opcode:

addr	78F	REG
addrl	79F	REG

See Also: **addi, subr**

alterbit

Mnemonic: **alterbit** Alter Bit

Format:

alterbit	*bitpos*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Copies the *src* value to *dst* with one bit altered. The *bitpos* operand specifies the bit to be changed; the condition code determines the value the bit is to be changed to. If the condition code is $X1X_2$, the selected bit is set; otherwise, it is cleared.

Action:

if (AC.cc **and** 010_2) = 0
  then *dst* ← *src* **and not** (2^(*bitpos* **mod** 32));
  else *dst* ← *src* **or** 2^(*bitpos* **mod** 32);
end if;

Faults: STANDARD

Example:

```
# assume AC.cc = 010
alterbit 24, g4, g9
# g9 ← g4, with bit 24 set
```

Opcode:

alterbit	58F	REG

See Also: **checkbit, clearbit, notbit, setbit**

and, andnot

Mnemonics:

and	And
andnot	And Not

Format:

and	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg
andnot	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise AND (**and** instruction) or AND NOT (**andnot** instruction) operation on the *src2* and *src1* values and stores the result in *dst*. Note in the action expressions below, the *src2* operand comes first, so that with the **andnot** instruction the expression is evaluated as

{*src2* **andnot** (*src1*)}

rather than

{*src1* **andnot** (*src2*)}.

Action:

and: *dst* ← *src2* **and** *src1*;

andnot: *dst* ← *src2* **and not** (*src1*);

Faults: STANDARD

Example:

```
and 0x17, g8, g2    # g2 ← g8 AND 0x17
andnot r3, r12, r9    # r9 ← r12 AND NOT r3
```

Opcode:

and	581	REG
andnot	582	REG

See Also: **nand, nor, not, notand, notor, or, ornot, xnor, xor**

atadd

Mnemonic: **atadd** Atomic Add

Format:

atadd	*src/dst*,	*src*,	*dst*
	reg	reg/lit	reg
	addr		

Description: Adds the *src* value (full word) to the value in the memory location specified with the *src/dst* operand. The initial value from memory is stored in *dst*.

The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the *src/dst* operand until the operation has been completed).

The memory location in *src/dst* is the address of the first byte (least significant byte) of the word. The address is automatically aligned to a word boundary. (Note that the *src/dst* operand maps to the *src1* operand of the REG machine-code format. Refer to Appendix B for a description of the REG format.)

Action:

```
# force alignment to word boundary
tempa ← src/dst and FFFFFFFC16;
temp ← atomic_read (tempa);
atomic_write (tempa) ← temp + src;
dst ← temp;
```

Faults: STANDARD

Example:

```
atadd r8, r2, r11   # r8 ← r2 + address r8,
                    # where r8 specifies the
                    # address of a word in
                    # memory; r11 ← initial
                    # value stored at address
                    # r8 in memory
```

Opcode: **atadd** 612 REG

See Also: **atmod**

atanr, atanrl

Mnemonics:

atanr	Arctangent Real
atanrl	Arctangent Long Real

Format:

atanr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description:

Calculates the arctangent of the quotient of *src2/src1* and stores the result in *dst*. The result is returned in radians and is in the range of $-\pi$ to $+\pi$, inclusive. The sign of the result is always the sign of *src2*.

For the **atanrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

These instructions are commonly used as part of an algorithm to convert rectangular coordinates to polar coordinates. They can also be used to implement the FORTRAN intrinsic functions ATAN and ATAN2. If *src1* is the floating-point literal value +1.0, then these instructions return a result in the range of $-\pi/2$ to $+\pi/2$.

The following table gives the range of results for various values of *src2* and *src1*, assuming that neither overflow nor underflow occurs.

Src2 \ Src1	-∞	-F	-0	+0	+F	+∞	NaN
-∞	-3π/4	-π/2	-π/2	-π/2	-π/2	*	NaN
-F	-π	-F	-π/2	-π/2	-π/2 to -0	+∞	NaN
-0	-π	-π	-π	-0	-0	+∞	NaN
+0	+π	+π	+π	+0	+0	+∞	NaN
+F	+π	+π to π/2	+π/2	+π/2	+π/2 to +0	+∞	NaN
+∞	+3π/4	+∞	+π/2	+π/2	+π/2	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F Means finite real number.

atanr, atanrl

Action: *dst* ← arctan (*src2/src1*);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
atanrl g8, g10, fp3   # fp3 ←
                      # arctan (g10,g11/g8,g9)
atanrl 1.0, g0, g0    # g0,g1 ← arctan (g0,g1)
```

Opcode:

atanr	680	REG
atanrl	690	REG

See Also: **tanr**

atmod

Mnemonic: **atmod** Atomic Modify

Format:

atmod	*src*,	*mask*,	*src/dst*
	reg	reg/lit	reg
	addr		

Description: Copies the *src/dst* value into the memory location specified in *src*. The bits set in the *mask* operand select the bits to be modified in memory. The initial value from memory is stored in *src/dst*.

The read and write of memory are done atomically (i.e., other processors are prevented from accessing the word of memory specified with the *src/dst* operand until the operation has been completed).

The memory location in *src* is the address of the first byte (least significant byte) of the word to be modified. The address is automatically aligned to a word boundary.

Action:

```
# force alignment to word boundary
tempa ← src and FFFFFFFC16;
temp ← atomic_read (tempa);
atomic_write (tempa) ← (src/dst and mask)
  or (temp and not(mask));
src/dst ← temp;
```

Faults: STANDARD

Example:

```
atmod g5, g7, g10   # g5 ← g5 masked by g7,
                    # where g5 specifies the
                    # address of a word in
                    # memory;
                    # g10 ← initial value
                    # stored at address g5
                    # in memory
```

Opcode:

atmod	610	REG

See Also: **atadd**

b, bx

Mnemonic:

b	Branch
bx	Branch Extended

Format:

b	*targ*
	disp
bx	*targ*
	mem

Description: Branches to the instruction specified with the *targ* operand. The *targ* operand specifies the IP of the target instruction.

With the **b** instruction, the IP specified with the targ operand can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP. When using the Intel 80960SA/SB Assembler, the *targ* operand for the **b** instruction must be a label.

The **bx** instruction performs the same operation as the **b** instruction except that the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP. Here, the the *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 8 for a complete discussion of the addressing modes available with memory-type operands.

NOTE

At the machine level, the **b** instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement for the **b** instruction), which can range from -2^{21} to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language version of the **b** instruction, the Intel 80960SA/SB Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine-instruction format:

displacement = (*targ* - IP)/4

b, bx

For further information about the CTRL instruction format, refer to Appendix B.

Action:

b: IP ← IP + *displacement*; # resume execution at new IP

bx: IP ← *targ*; # resume execution at new IP

Faults: STANDARD

Example:

```
b xyz  # IP ← xyz;

bx 1332 (ip)  # IP ← IP + 1332;
              # this example uses ip-relative
              # addressing.
```

Opcode:

b	08	CTRL
bx	84	MEM

See Also: **bal**, **balx**, BRANCH IF, COMPARE INTEGER AND BRANCH, COMPARE ORDINAL AND BRANCH

bal, balx

Mnemonic:

bal	Branch And Link
balx	Branch And Link Extended

Format:

bal	*targ*	
	disp	
balx	*targ*,	*dst*
	mem	reg

Description: Stores the address of the next instruction (the instruction following the **bal** or **balx** instruction) in a register and branches to the instruction specified with the *targ* operand. The *targ* operand specifies the IP of the target instruction.

The **bal** and **balx** instructions are intended for calling leaf procedures (procedures that do not call other procedures). The IP saved in the register provides a return IP that the leaf procedure can branch to (using a **b** or **bx** instruction) to perform a return from the procedure. Note that these instructions do not use the processor's call-and-return mechanism, so the calling procedure shares its local-register set with the called (leaf) procedure.

With the **bal** instruction, the address of the next instruction is stored in register g14. The *targ* operand value can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP. When using the Intel 80960SA/SB Assembler, the *targ* operand for the **b** instruction must be a label.

The **balx** instruction performs almost the same operation as the **bal** instruction except that the address of the next instruction is stored in *dst* (allowing the return IP to be stored in any available register). With the **balx** instruction, the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP. Here, the *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect branching can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 8 for a complete discussion of the addressing modes available with memory-type operands.

bal, balx

NOTE

At the machine level, the **bal** instruction uses the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement for the **bal** instruction), which can range from -2^{21} to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels or absolute addresses to be used in the assembly-language version of the **bal** instruction, the Intel 80960SA/SB Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

displacement = (*targ* - IP)/4

For further information about the CTRL instruction format, refer to Appendix B.

Action:

bal:	g14 ← IP + 4;	# destination next IP is always g14
	IP ← IP + *targ*;	# resume execution at the new IP
balx:	*dst* ← IP + inst_length;	# instruction length is 4 or 8 bytes
	IP ← *targ*;	# resume execution at the new IP

Faults:

STANDARD

Example:

```
bal xyz  # IP ← xyz;

balx (g2), g4  # IP ← (g2);
               # address of return instruction
               # is stored in g4; example of
               # indirect addressing.
```

Opcode:

bal	0B	CTRL
balx	85	MEM

See Also:

b, bx

bbc, bbs

Mnemonic:

bbc	Check Bit and Branch If Clear
bbs	Check Bit and Branch If Set

Format:

bb*	*bitpos*,	*src*,	*targ*
	reg/lit	reg	disp

Description: Checks the bit in *src* (designated by *bitpos*) and sets the condition code in the arithmetic controls according to the value found. The processor then performs a conditional branch to the instruction specified with the *targ* operand, according to the state of the condition code. When using the Intel 80960SA/SB Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

For the **bbc** instruction, if the selected bit in *src* is clear, the processor sets the condition code to 010_2 and branches to the instruction specified with the *targ* operand; otherwise, it sets the condition code to 000_2 and goes to the next instruction.

For the **bbs** instruction, if the selected bit is set, the processor sets the condition code to 010_2 and branches to *targ*; otherwise, it sets the condition code to 000_2 and goes to the next instruction.

The *targ* operand can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from the current IP.

NOTE

At the machine level, the **bbc** and **bbs** instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from -2^{10} to $(2^{10} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language versions of the **bbc** and **bbs** instructions, the Intel 80960SA/SB Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

displacement = (*targ* - IP)/4

For further information about the COBR instruction format, refer to Appendix B.

bbc, bbs

Action:

bbc:

if (*src* **and** 2^(*bitpos* **mod** 32)) = 0
then AC.cc ← 010_2;
IP ← IP + 4 + (*displacement* * 4);
resume execution at the new IP
else AC.cc ← 000_2;
IP ← IP + 4; # resume execution at the next IP
end if;

bbs:

if (*src* **and** 2^(*bitpos* **mod** 32)) = 1
then AC.cc ← 010_2;
IP ← IP + 4 + (*displacement* * 4);
resume execution at the new IP
else AC.cc ← 000_2;
IP ← IP + 4; # resume execution at the next IP
end if;

Faults: STANDARD

Example:

```
# assume bit 10 of r6 is clear
bbc 10, r6, xyz  # bit 10 of r6 is checked
                 # and found clear;
                 # AC.cc ← 010₂
                 # IP ← xyz;
```

Opcode:

bbc	30	COBR
bbs	37	COBR

See Also: **chkbit**

BRANCH IF

Mnemonics:

be	Branch If Equal
bne	Branch If Not Equal
bl	Branch If Less
ble	Branch If Less Or Equal
bg	Branch If Greater
bge	Branch If Greater Or Equal
bo	Branch If Ordered
bno	Branch If Unordered

Format:

b*	*targ*
	disp

Description: Branches to the instruction specified with the *targ* operand, according to the state of the condition code in the arithmetic controls. When using the Intel 80960SA/SB Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

For all branch-if instructions except the **bno** instruction, the processor branches to the instruction specified with the *targ* operand, if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, it goes to the next instruction.

For the **bno** instruction, the processor branches to the instruction specified with *targ*, if the logical AND of the condition code and the mask-part of the opcode is zero. Otherwise, it goes to the next instruction.

The *targ* operand value can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP.

NOTE

At the machine level, the branch-if instructions use the CTRL instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statements), which can range from -2^{21} to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language version of the branch-if instructions, the Intel Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

displacement = (*targ* - IP)/4

BRANCH IF

For further information about the CTRL instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
bno	000	Unordered
bg	001	Greater
be	010	Equal
bge	011	Greater or equal
bl	100	Less
bne	101	Not equal
ble	110	Less or equal
bo	111	Ordered

For the **bno** instruction (unordered), the branch is taken if the condition code is equal to 000_2.

The mask is in bits 0-2 of the opcode.

Action:

For All Instructions Except **bno**:

if (mask **and** AC.cc) $\neq 000_2$
  then IP ← IP + *displacement*; # resume execution at new IP
end if;

bno:

if AC.cc = 000_2
  then IP ← IP + *displacement*; # resume execution at new IP
end if;

Faults:

STANDARD

BRANCH IF

Example:

```
# assume (AC.cc AND 100₂) ≠ 0
bl xyz  # IP ← xyz;
```

Opcode:

be	12	CTRL
bne	15	CTRL
bl	14	CTRL
ble	16	CTRL
bg	11	CTRL
bge	13	CTRL
bo	17	CTRL
bno	10	CTRL

See Also: **b, bx**

call

Mnemonic: **call** Call

Format: **call** *targ*
disp

Description: Calls a new procedure. The *targ* operand specifies the IP of the first instruction of the called procedure. When using the Intel 80960 Assembler, the *targ* operand must be a label.

In executing this instruction, the processor performs a local call operation as described in Chapter 4 in the section titled "Local Call." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

The *targ* operand can be no farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP.

NOTE

At the machine level, the **call** instruction uses the CTRL instruction format. With this format, the first instruction of the called procedure is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from -2^{21} to $(2^{21} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

To allow labels to be used in the assembly-language version of the **call** instruction, the Intel 80960 Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

displacement = (*targ* - IP)/4

For further information about the CTRL instruction format, refer to Appendix B.

call

Action:

wait for any uncompleted instructions to finish;
temp ← (SP + C) **and not** (C); # round to next boundary
where C is the implementation-defined constant SALIGN * 16 - 1
RIP ← IP;
if register_set_available
  then allocate as new frame;
  else save a register_set in memory at its FP;
    allocate as new frame;
local register references now refer to new frame
IP ← IP + *displacement*;
PFP ← FP;
FP ← temp;
SP ← temp + 64;

Faults: STANDARD

Example:

```
call xyz  # IP ← xyz
```

Opcode:

call	09	CTRL

See Also: **bal, calls, callx**

calls

Mnemonic: **calls** Call System

Format: **calls** *targ*
reg/lit

Description: Calls a system procedure. The *targ* operand gives the number of the procedure being called.

For this instruction, the processor performs the system call operation described in Chapter 4 in the section titled "System Call." The *targ* operand provides an index to an entry in the system procedure table. From this entry, the processor gets the IP of the called procedure.

The procedure called can be either a local procedure or a supervisor procedure, depending on the entry type in the procedure table. If it is a supervisor procedure, the processor also switches to supervisor mode (if it is not already in this mode).

As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. If the processor switches to the supervisor mode, the new stack frame is created on the supervisor stack.

Action:

```
if targ > 259 then raise Protection Length Fault;
wait for any uncompleted instructions to finish;
temp_p_e ← memory (sptbase + 48 + (4 * targ));
# read entry from system-procedure table, where
# sptbase is address of system-procedure table from IMI
RIP ← IP;
IP ← temp_p_e.address;
if (temp_p_e.type = local) or execution_mode = supervisor
 then temp ← (SP + C) and not(C); # round to next boundary
# where C is the implementation-defined constant SALIGN * 16 - 1
   tempRRR ← 000₂;
 else temp ← memory (sptbase + 12); # read supervisor stack pointer
   tempRRR ← 01T₂;  # T is process_controls.trace-enable flag
   execution_mode ← supervisor;
   process_controls.trace-enable flag ← temp.T;
endif;
```

calls

```
if frame_available
   then allocate as new frame;
   else save a frame in memory at its FP;
     allocate as new frame;
# local register references now refer to new frame
endif;
PFP ← FP;
PFP.RRR ← tempRRR;
FP ← temp;
SP ← temp + 64;
```

Faults: STANDARD

Example:

```
calls r12   # IP ← value obtained from
            # procedure table for procedure
            # number given in r12
```

Opcode:

calls	660	REG

See Also: **bal, call, callx**

callx

Mnemonic: **callx** Call Extended

Format: **callx** *targ*
mem

Description: Calls a new procedure. The *targ* operand specifies the IP of the first instruction of the called procedure.

In executing this instruction, the processor performs a local call operation as described in Chapter 4 in the section titled "Local Call." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

This instruction performs the same operation as the **call** instruction except that the target instruction can be farther than -2^{23} to $(2^{23} - 4)$ bytes from the current IP.

The *targ* operand is a memory type, which allows the full range of addressing modes to be used to specify the IP of the target instruction. The "IP + displacement" addressing mode allows the instruction to be IP-relative. Indirect calls can be performed by placing the target address in a register and then using one of the register-indirect addressing modes.

Refer to Chapter 8 for a complete discussion of the addressing modes available with memory-type operands.

Action:

```
wait for any uncompleted instructions to finish;
temp ← (SP + C) and not (C); # round to next boundary
# where C is the implementation-defined constant SALIGN * 16 - 1
RIP ← IP;
if register_set_available
   then allocate as new frame;
   else save a register_set in memory at its FP;
     allocate as new frame;
# local register references now refer to new frame
endif;
IP ← targ;
PFP ← FP;
FP ← temp;
SP ← temp + 64;
```

Faults: STANDARD

callx

Example:

```
callx (g5)   # IP ← (g5), where the address
             # in g5 is the address of the new
             # procedure
```

Opcode:

callx	86	MEM

See Also: **call, calls**

chkbit

Mnemonic: **chkbit** Check Bit

Format:

chkbit	*bitpos*,	*src*
	reg/lit	reg/lit

Description: Checks the bit in *src* designated by *bitpos* and sets the condition code according to the value found. If the bit is set, the condition code is set to 010_2; if the bit is clear, the condition code is set to 000_2.

Action:

if (*src* **and** 2^(*bitpos* **mod** 32)) = 0
  then AC.cc ← 000_2;
  else AC.cc ← 010_2;
end if;

Faults: STANDARD

Example:

```
chkbit 13, g8    # checks bit 13 in g8
```

Opcode:

chkbit	5AE	REG

See Also: **alterbit, clrbit, notbit, setbit**

classr, classrl

Mnemonic:

classr	Classify Real
classrl	Classify Long Real

Format:

classr*	*src*
	freg/flit

Description: Checks the classification of the real number in *src* and stores the class in arithmetic-status bits (3 through 6) of the arithmetic controls.

For the **classrl** instruction, if the *src* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the arithmetic-status bits depending on the classification of the operand.

AStatus	Classification
s000	Zero
s001	Denormalized number
s010	Normal finite number
s011	Infinity
s100	Quiet NaN
s101	Signaling NaN
s110	Reserved operand

The "s" bit is set to the sign of the *src* operand.

Refer to Chapter 10 for a discussion of the different real number classifications.

classr, classrl

Action:

s ← sign_of(*src*)
if *src* = 0
  then arithmetic_status ← $s000_2$;
elseif *src* = denormalized
  then arithmetic_status ← $s001_2$;
elseif *src* = normal finite
  then arithmetic_status ← $s010_2$;
elseif *src* = ∞
  then arithmetic_status ← $s011_2$;
elseif *src* = QNaN
  then arithmetic_status ← $s100_2$;
elseif *src* = SNaN
  then arithmetic_status ← $s101_2$;
elseif *src* = reserved operand
  then arithmetic_status ← $s110_2$;
end if

Faults:

STANDARD — Refer to the discussion of faults at the beginning of this chapter.

None of the floating-point exceptions can be raised.

Example:

```
classrl g12  # classifies long real in g12,g13
```

Opcode:

classr	68F	REG
classrl	69F	REG

clrbit

Mnemonic: **clrbit** Clear Bit

Format:

clrbit	*bitpos*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Copies the *src* value to *dst* with one bit cleared. The *bitpos* operand specifies the bit to be cleared.

Action: *dst* ← *src* **and not**(2^(*bitpos* **mod** 32));

Faults: STANDARD

Example:

```
clrbit 23, g3, g6  # g6 ← g3 with bit 23
                   # cleared
```

Opcode: **clrbit** 58C REG

See Also: **alterbit, chkbit, notbit, setbit**

cmpdeci, cmpdeco

Mnemonics: **cmpdeci** Compare and Decrement Integer
cmpdeco Compare and Decrement Ordinal

Format:

cmpdec*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then decremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	*src1* < *src2*
010	*src1* = *src2*
001	*src1* > *src2*

These instructions are intended for use in ending iterative loops. For the **cmpdeci** instruction, interger overflow is ignored to allow looping down through the minimum integer values. Description)

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
else AC.cc ← 001_2;
end if;
dst ← *src2* - 1; #overflow suppressed for **cmpdeci** instruction

Faults: STANDARD

Example:

```
cmpdeci 12, g7, g1    # g7 and 12 are compared;
                      # g1 ← g7 - 1
```

Opcode:

cmpdeci	5A7	REG
cmpdeco	5A6	REG

See Also: **cmpinco, cmpo**

cmpi, cmpo

Mnemonics:

cmpi	Compare Integer
cmpo	Compare Ordinal

Format:

cmp*	*src1*,	*src2*
	reg/lit	reg/lit)

Description: Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	*src1* < *src2*
010	*src1* = *src2*
001	*src1* > *src2*

The **cmpi** instruction followed by one of the branch-if instructions is equivalent to one of the compare-integer-and-branch instructions. The latter method of comparing and branching produces more compact code; however, the former method can result in faster running code because it takes advantage of the processor's pipelined architecture. The same is true for the **cmpo** instruction and the compare-ordinal-and-branch instructions.

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
else AC.cc ← 001_2;
end if;

Faults: STANDARD

Example:

```
cmpo 0x10, r9   # compare values in r9 and 0x10
                # and set AC.cc
```

Opcode:

cmpi	5A1	REG
cmpo	5A0	REG

See Also: **cmpibe, cmpr, cmpdeci, cmpdeco**

cmpinci, cmpinco

Mnemonics:

cmpinci	Compare and Increment Integer
cmpinco	Compare and Increment Ordinal

Format:

cmpinc*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description:

Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. The *src2* operand is then incremented by one and the result is stored in *dst*.

The following table shows the setting of the condition code for the three possible results of the comparison.

Condition Code	Comparison
100	*src1* < *src2*
010	*src1* = *src2*
001	*src1* > *src2*

These instructions are intended for use in ending iterative loops. For the **cmpinci** instruction, integer overflow is ignored to allow looping up through the maximum integer values.

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
else AC.cc ← 001_2;
end if;
dst ← *src2* + 1; # overflow suppressed for **cmpinci** instruction

Faults:

STANDARD

Example:

```
cmpinco r8, g2, g9    # g2 and r8 are compared;
                      # g9 ← g2 + 1
```

Opcode:

cmpinci	5A5	REG
cmpinco	5A4	REG

See Also:

cmpdeco, cmpo

cmpor, cmporl

Mnemonics:

cmpor	Compare Ordered Real
cmporl	Compare Ordered Long Real

Format:

cmpor*	*src1*,	*src2*
	freg/flit	freg/flit

Description: Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison.

For the **cmporl** instruction, if the *src1* or *src2* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the setting of the condition code for the four possible results of the comparison.

Condition Code	Comparison
100	*src1* < *src2*
010	*src1* = *src2*
001	*src1* > *src2*
000	if either *src1* or *src2* is a NaN

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000_2 and a floating invalid-operation exception is raised. The **cmpor** and **cmporl** instructions operate the same as the **cmpr** and **cmprl** instructions, except that the latter instructions do not signal an exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
elseif *src1* > *src2* **then** AC.cc ← 001_2;
else AC.cc ← 000_2; # indicates one number is a NaN
  raise floating invalid operation fault
end if;

cmpor, cmporl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation	One or more operands are a NaN value.

Example:

```
cmporl g6, g12    # compare value in g12,g13
                  # with value in g6,g7
```

Opcode:

cmpor	684	REG
cmporl	694	REG

See Also: **cmpr, cmpi, BRANCH IF**

cmpr, cmprl

Mnemonics:

cmpr	Compare Real
cmprl	Compare Long Real

Format:

cmpr*	*src1*,	*src2*
	freg/flit	freg/flit

Description: Compares the *src2* and *src1* values and sets the condition code according to the results of the comparison. For the **cmprl** instruction, if the *src1* or *src2* operand references a global or local register, this register is the first (lowest numbered) of two successive registers.

The following table shows the setting of the condition code for the four possible results of the comparison.

Condition Code	Comparison
100	*src1* < *src2*
010	*src1* = *src2*
001	*src1* > *src2*
000	if either *src1* or *src2* is a NaN

The algorithm for these instructions checks the classification of the operands. If either is in the NaN class, the condition code is set to 000_2, but no fault is raised. The **cmpr** and **cmprl** instructions operate the same as the **cmpor** and **cmporl** instructions, except that the latter instructions raise an invalid-operand exception if a NaN value is detected.

If a floating-reserved-encoding fault occurs, the condition code results are undefined.

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
elseif *src1* > *src2* **then** AC.cc ← 001_2;
else AC.cc ← 000_2; # indicates one number is a NaN
end if;

cmpr, cmprl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation	One or more operands are an SNaN value.

Example:

```
cmprl g2, g6  # compare values in g6,g7
              # and g2,g3
```

Opcode:

cmpr	685	REG
cmprl	695	REG

See Also: **cmpor, cmpi, BRANCH IF**

COMPARE AND BRANCH

Mnemonics:

cmpibe	Compare Integer And Branch If Equal
cmpibne	Compare Integer And Branch If Not Equal
cmpibl	Compare Integer And Branch If Less
cmpible	Compare Integer And Branch If Less Or Equal
cmpibg	Compare Integer And Branch If Greater
cmpibge	Compare Integer And Branch If Greater Or Equal
cmpibo	Compare Integer And Branch If Ordered
cmpibno	Compare Integer And Branch If Unordered
cmpobe	Compare Ordinal And Branch If Equal
cmpobne	Compare Ordinal And Branch If Not Equal
cmpobl	Compare Ordinal And Branch If Less
cmpoble	Compare Ordinal And Branch If Less Or Equal
cmpobg	Compare Ordinal And Branch If Greater
cmpobge	Compare Ordinal And Branch If Greater Or Equal

Format:

cmpib*	*src1*,	*src2*,	*targ*
	reg/lit	reg	disp
cmpob*	*src1*,	*src2*,	*targ*
	reg/lit	reg	disp

Description: Compares the *src2* and *src1* values and sets the condition code in the arithmetic controls according to the results of the comparison. If the logical AND of the condition code and the mask-part of the opcode is not zero, the processor branches to the instruction specified with the *targ* operand; otherwise, the processor goes to the next instruction. When using the Intel 80960SA/SB Assembler, the *targ* operand must be a label, which specifies the IP of the target instruction.

The *targ* operand can be no farther than -2^{12} to $(2^{12} - 4)$ bytes from the current IP.

NOTE

At the machine level, the compare-and-branch instructions use the COBR instruction format. With this format, the target instruction for the branch is specified by means of a word-displacement (represented by *displacement* in the following action statement), which can range from -2^{10} to $(2^{10} - 1)$. To determine the IP of the target instruction, the processor converts this *displacement* value to a byte displacement (i.e., multiplies the value by 4). It then adds the resulting byte displacement to the current IP.

COMPARE AND BRANCH

To allow labels to be used in the assembly-language versions of these instructions, the Intel 80960SA/SB Assembler performs the following calculation to convert the *targ* value in an assembly-language instruction to the *displacement* value required by the machine instruction format:

displacement = (*targ* - IP)/4

For further information about the COBR instruction format, refer to Appendix B.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Branch Condition
cmpibno	000	No Condition
cmpibg	001	*src1* > *src2*
cmpibe	010	*src1* = *src2*
cmpibge	011	*src1* ≥ *src2*
cmpibl	100	*src1* < *src2*
cmpibne	101	*src1* ≠ *src2*
cmpible	110	*src1* ≤ *src2*
cmpibo	111	Any Condition
cmpobg	001	*src1* > *src2*
cmpobe	010	*src1* = *src2*
cmpobge	011	*src1* ≥ *src2*
cmpobl	100	*src1* < *src2*
cmpobne	101	*src1* ≠ *src2*
cmpoble	110	*src1* ≤ *src2*

The **cmpibo** instruction always branches; the **cmpibno** instruction never branches.

COMPARE AND BRANCH

The functions that these instructions perform can be duplicated with a **cmpi** instruction followed by a branch-if instruction, as described in the description of the **cmpi** instruction in this chapter.

Action:

if *src1* < *src2* **then** AC.cc ← 100_2;
elseif *src1* = *src2* **then** AC.cc ← 010_2;
else AC.cc ← 001_2;
end if;
if mask **and** AC.cc ≠ 000_2
  then IP ← IP + 4 + (*displacement* * 4);
  # resume execution at the new IP
  else IP ← IP + 4;
# resume execution at the next IP
end if;

Faults: STANDARD

Example:

```
# assume g3 < g9
cmpibl g3, g9, xyz  # g9 is compared with g3;
                    # IP ← xyz.
# assume r7 ≥ 19
cmpobge r7, 19, xyz  # 19 is compared with r7
                     # IP ← xyz.
```

Opcode:

cmpibe	3A	COBR
cmpibne	3D	COBR
cmpibl	3C	COBR
cmpible	3E	COBR
cmpibg	39	COBR
cmpibge	3B	COBR
cmpibo	3F	COBR
cmpibno	38	COBR
cmpobe	32	COBR
cmpobne	35	COBR
cmpobl	34	COBR
cmpoble	36	COBR
cmpobg	31	COBR
cmpobge	33	COBR

See Also: **BRANCH IF**, **cmpi**

concmpi, concmpo

Mnemonics:

concmpi	Conditional Compare Integer
concmpo	Conditional Compare Ordinal

Format:

concmp*	*src1*,	*src2*
	reg/lit	reg/lit

Description: Compares the *src2* and *src1* values if bit 2 of the condition code is not set. If the comparison is performed, the condition code is set according to the results of the comparison.

These instructions are provided to facilitate bounds checking by means of two-sided range comparisons (e.g., is A between B and C?). They are generally used after a compare instruction to test whether a value is inclusively between two other values.

The example below illustrates this application by testing whether the value in g3 is between the values in g5 and g6, where g5 is assumed to be less than g6. First a comparison (**cmpo**) of g3 and g6 is performed. If g3 is less than or equal to g6 (i.e., condition code is either 010_2 or 001_2), a conditional comparison (**concmpo**) of g3 and g5 is then performed. If g3 is greater than or equal to g5 (indicating that g3 is within the bounds of g5 and g6), the condition code is set to 010_2; otherwise, it is set to 001_2.

Action:

if AC.cc $\neq 1XX_2$ **then**
  if *src1* $\leq$ *src2*
    then AC.cc $\leftarrow 010_2$;
    else AC.cc $\leftarrow 001_2$;
  endif;
endif;

Faults: STANDARD

Example:

```
cmpo g6, g3       # compares g6 and g3 and
                  # sets AC.cc
concmpo g5, g3    # if AC.cc is not 1XX,
                  # g5 is compared with g3
```

Opcode:

concmpi	5A3	REG
concmpo	5A2	REG

See Also: **cmpo, cmpi**

cosr, cosrl

Mnemonics:

cosr	Cosine Real
cosrl	Cosine Long Real

Format:

cosr*	*src*,	*dst*
	freg/flit	freg

Description: Calculates the cosine of the value in *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range -1 to +1, inclusive.

For the **cosrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the cosine of various classes of numbers with neither overflow nor underflow.

Src	Dst
-∞	*
-F	-1 to +1
-0	+1
+0	+1
+F	-1 to +1
+∞	*
NaN	NaN

Notes:
F means finite-real number
* indicates floating invalid-operation exception

In the trigonometric instructions, the 80960SB uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 10 titled "Pi" gives this π value, along with some suggestions for representing this value in a program.

Action: *dst* ← **cosine** (*src*);

cosr, cosrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation	The src operand is ∞.
	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
cosrl r8, g2    # cosine of value in r8,r9 is
                # stored in g2,g3
```

Opcode:

cosr	68D	REG
cosrl	69D	REG

See Also: **sinr, sinrl, tanr, tanrl**

cpyrsre, cpysre

Mnemonics:

cpysre	Copy Sign Real Extended
cpyrsre	Copy Reversed Sign Real Extended

Format:

cpy*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Copies the absolute value of *src1* into *dst*. For the **cpysre** instruction, the sign of *src2* is copied to *dst*; for the **cpyrsre** instruction, the opposite of the sign of *src2* is copied to *dst*.

If the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of three successive registers. Also, the number of this register must be a multiple of four (e.g., g0, g4, g8).

These instructions only operate on values in the extended-real format. The same operations can be performed on real- and long-real values using the **setbit** and **clearbit** instructions, or a combination of the **chkbit** and **alterbit** instructions.

Action:

cpysre: **if** *src2* is positive **then** *dst* ← abs (*src1*);
else *dst* ← -abs (*src1*);
endif;

cpyrsre: **if** *src2* is negative **then** *dst* ← abs (*src1*);
else *dst* ← -abs (*src1*);
endif;

Faults: STANDARD — Refer to the discussion of faults at the beginning of this chapter.

Example:

```
cpysre fp0, fp1, fp2
# absolute value from fp0 is copied to
# fp2; sign from fp1 is copied to fp2
```

Opcode:

cpysre	6E2	REG
cpyrsre	6E3	REG

cvtilr, cvtir

Mnemonics:

cvtilr	Convert Long Integer to Real
cvtir	Convert Integer to Real

Format:

cvti*	*src*,	*dst*
	reg/lit	freg

Description: Converts the integer in *src* to a real and stores the result in *dst*. For the **cvtilr** instruction, the *src* operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

Converting an integer to long real format requires two instructions. First, the integer is converted to extended real format by using the **cvtir** or **cvtilr** instruction with a floating-point register as a destination. Then the **movrl** instruction is used to move the value from the floating-point register to two global or local registers, causing an explicit conversion to long real format. (Note that this conversion is always exact.) The example section below illustrates this conversion.

Action: *dst* ← **real** (*src*);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.

The following floating-point exception can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Inexact	Can only be signaled when converting an integer or long integer to real (32-bit) format.

Example:

```
# Conversion of an integer to a long real value
cvtir g6, fp3
movrl fp3, g8  # result stored in g8,g9
```

Opcode:

cvtir	674	REG
cvtilr	675	REG

See Also: **cvtri, movr**

cvtri, cvtril, cvtzri, cvtzril

Mnemonics:

cvtri	Convert Real To Integer
cvtril	Convert Real To Integer Long
cvtzri	Convert Truncated Real To Integer
cvtzril	Convert Truncated Real To Long Integer

Format:

cvtri*	*src*,	*dst*
	freg/flit	reg

Description: Converts the real value in *src* to an integer and stores the result in *dst*.

For the **cvtril** and **cvtzril** instructions, the *dst* operand references the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The nontruncated versions of these instructions round according to the current rounding mode in the Arithmetic Controls register. The truncated versions always round toward zero.

Converting a long real value to an integer requires two instructions. First, the long real value is converted to extended real format by using the **movrl** instruction with a floating-point register as a destination. (Note that this operation is always exact.) Then one of the convert real-to-integer instructions is used to move the value from the floating-point register to one or two global or local registers. The example section below illustrates this conversion.

If the magnitude of the result cannot be represented in the destination, an integer-overflow fault is raised, and the maximum positive or maximum negative value is stored in the destination (depending on whether the real value was positive or negative, respectively).

Action:

dst ← **integer** (*src*);
src is rounded to integer value

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.

The following exceptions can be raised. Whether or not the exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls register.

Integer Overflow	Result is too large for destination format.

cvtri, cvtril, cvtzri, cvtzril

Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.
Floating Invalid Operation	The src operand is ∞.

Example:

```
# Conversion of long real value to an integer
movrl g4, fp2     # long-real source is
                  # converted to extended-real
                  # format and moved to fp2

cvtril fp2, g12   # extended-real value is
                  # converted to long integer
```

Opcode:

cvtri	6C0	REG
cvtril	6C1	REG
cvtzri	6C2	REG
cvtzril	6C3	REG

See Also: **cvtir, movr**

daddc

Mnemonic: **daddc** Decimal Add With Carry

Format:

daddc	*src1*,	*src2*,	*dst*
	reg	reg	reg

Description: Adds bits 0 through 3 of *src2* and *src1* and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of *dst*. If the addition results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of *src2* are copied to *dst* unchanged.

This instruction is intended to be used iteratively to add binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction asssumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in *dst* is unpredictable.

Action:

Let the value of the condition code be xCx.
dst ← *src2* + *src1* + C;
AC.cc ← $0C0_2$;
C is carry from addition of bits 0 through 3 of operands
Bits 4 - 31 of *dst* are same as bits 4 - 31 of *src2*

Faults: STANDARD

Example:

```
daddc g5, g9, g10   # g10 ← g9 + g5 + Carry Bit
                    # where arithmetic is
                    # carried out only on bits 0
                    # through 3 of the operands
```

Opcode: **daddc** 642 REG

See Also: **dsubc, dmovt**

divi, divo

Mnemonic:

divi	Divide Integer
divo	Divide Ordinal

Format:

div*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Divides the *src2* value by the *src1* value and stores the result in *dst*.

For the **divi** instruction, an integer-overflow fault can be signaled.

Action: *dst* ← *src2* / *src1*;

Faults:

STANDARD	Refer to discussion of faults at the beginning of this chapter.
Arithmetic Zero Divide	The *src1* operand is 0.

The following fault condition can be raised with the **divi** instruction. Whether or not a fault is raised depends on the state of its associated mask bit in the arithmetic-controls register.

Integer Overflow	Result is too large for destination format.

Example:

```
divo r3, r8, r13 # r13 ← r8/r3
```

Opcode:

divi	74B	REG
divo	70B	REG

See Also: **ediv, mulo**

divr, divrl

Mnemonic:

divr	Divide Real	
divrl	Divide Long Real	

Format:

divr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Divides the *src2* value by the *src1* value and stores the result in *dst*.

For the **divrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0, ∞, or a NaN.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
		-∞	**-F**	**-0**	**+0**	**+F**	**+∞**	**NaN**
Src2	**-∞**	*	+∞	+∞	-∞	−∞	*	NaN
	-F	+0	+F	**	**	-F	-0	NaN
	-0	+0	+0	*	*	-0	-0	NaN
	+0	-0	-0	*	*	+0	+0	NaN
	+F	-0	-F	**	**	+F	+0	NaN
	+∞	*	-∞	−∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.
**	Indicates floating zero-divide exception.

Action: *dst* ← *src2* / *src1*;

divr, divrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Zero Divide	The *src1* operand is 0 and the *src2* operand is numeric and finite.
Floating Invalid Operation	Both source operands are 0 or both are ∞. One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
divrl g10, g0, fp1 # fp1 ← g0,g1 / g10,g11
```

Opcode:

divr	78B	REG
divrl	79B	REG

See Also: **ediv, mulr, mulrl**

dmovt

Mnemonic: **dmovt** Decimal Move And Test

Format:

dmovt	*src*,	*dst*
	reg	reg

Description: Copies the *src* value into *dst*. The least-significant eight bits of the *src* value are tested to determine whether or not they constitute a valid ASCII decimal (00110000_2 .. 00111001_2), and the condition code is set accordingly. If the value is a valid ASCII decimal, the condition code is set to 000_2; otherwise, it is set to 010_2.

This instruction is intended to be used iteratively to validate decimal strings.

Action:

dst ← *src*;
if *src* = 0011000_2 .. 00111001_2
then AC.cc ← 000_2;
else AC.cc ← 010_2;
end if;

Faults: STANDARD

Example:

```
dmovt g1, g6 # g6 ← g1;
             # g1 tested for decimal value
```

Opcode:

dmovt	644	REG

See Also: **daddc, dsubc**

dsubc

Mnemonic: **dsubc** Decimal Subtract With Carry

Format:

dsubc	*src1*,	*src2*,	*dst*
	reg	reg	reg

Description: Subtracts bits 0 through 3 of *src2* and *src1* and bit 1 of the condition code (used here as a carry bit). The result is stored in bits 0 through 3 of *dst*. If the subtraction results in a carry, bit 1 of the condition code is set. Bits 4 through 31 of *src* are copied to *dst* unchanged.

This instruction is intended to be used iteratively to subtract binary-coded-decimal (BCD) values in which the least-significant four bits of the operands represent the decimal numbers 0 to 9. The instruction assumes that the least significant 4 bits of both operands are valid BCD numbers. If these bits are not valid BCD numbers, the resulting value in *dst* is unpredictable.

Action:
Let the value of the condition code be xCx.
dst ← *src2* - *src1* - 1 + C;
AC.cc ← $0C0_2$;
C is carry from subtraction of bits 0 through 4 of operands
Bits 4 - 31 of *dst* are same as bits 4 - 31 of *src2*

Faults: STANDARD

Example:
```
dsubc r1, r2, r12 # r12 ← r2 - r1 -1 + Carry
                  # Bit, where arithmetic is
                  # carried out only on bits 0
                  # through 3 of the operands
```

Opcode:

dsubc	643	REG

See Also: **daddc, dmovt**

ediv

Mnemonic: **ediv** Extended Divide

Format:

ediv	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Divides *src2* by *src1* and stores the result in *dst*. The *src2* value is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The *src2* operand specifies the lower numbered register, which contains the least significant bits of the operand. The *src2* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...). The *src1* value is a normal ordinal (i.e., 32 bits).

The remainder is stored in the register designated by *dst* and the quotient is stored in the next highest numbered register. The *dst* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...).

This instruction performs ordinal arithmetic.

If this operation overflows (i.e., the quotient or remainder do not fit in 32-bits), no fault is raised and the result is undefined.

Action:

dst ← (*src2* - (*src2* / *src1*) * *src1*); # remainder
dst + 1 ← (*src2* / *src1*); # quotient

Faults: STANDARD, Arithmetic Zero-Divide

Example:

```
ediv g3, g4, g10 # g10 ← remainder of g4,g5/g3
                 # g11 ← quotient of g4,g5/g3
```

Opcode:

ediv	671	REG

See Also: **emul**

emul

Mnemonic: **emul** Extended Multiply

Format:

emul	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Multiplies *src2* by *src1* and stores the result in *dst*. The result is a long ordinal (i.e., 64 bits), which is stored in two adjacent registers. The *dst* operand specifies the lower numbered register, which receives the least significant bits of the result. The *dst* operand must be an even numbered register (i.e., r0, r2, r4, ... or g0, g2, ...).

This instruction performs ordinal arithmetic.

Action:
dst ← (*src1* * *src2*) mod 2^32;
dst + 1 ← (*src* * *src2*)/mod 2^32;

Faults: STANDARD

Example:
```
emul r4, r5, g2  # g2,g3 ← r4 * r5
```

Opcode: **emul** 670 REG

See Also: **ediv**

expr, exprl

Mnemonic:

expr	Exponent Real
exprl	Exponent Long Real

Format:

exp*	*src*,	*dst*
	freg/flit	freg

Description: Calculates an approximation of the exponential value of 2 to the *src* power, minus 1, and stores the result in *dst*. The *src* value must be within the range of -0.5 to +0.5, inclusive. If the *src* value is outside this range, the result is undefined.

For the **exprl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when computing the exponent of various classes of numbers.

Src	Dst
-0.5 to -0	$-(1/\sqrt{2})-1$ to -0
-0	-0
+0	+0
+0 to +0.5	+0 to $\sqrt{2}$ -1

Notes: *** Results are unpredictable.

Action:

dst ← (2^*src*) - 1;

expr, exprl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is set.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
# y = 2^x   (y and x in g0)
# uses identity
#     2^x = 2^(I+f)
#         = 2^I * ((2^f - 1)+1)
# where:  I integer, -0.5 <= f <= +0.5
# assumes round-to-nearest
# does not handle infinities or NaNs
_pow2x:
  roundr  g0,fp0                        # I in fp0
  subr    fp0,g0,g0       # f in g0
  expr    g0,g0
  addr    0f1.0,g0,g0
  cvtri   fp0,g1
  scaler  g1,fp0,g0
```

Opcode:

expr	689	REG
exprl	699	REG

See: **scaler, logr**

extract

Mnemonic: **extract** Extract

Format:

extract	*bitpos*,	*len*,	*src/dst*
	reg/lit	reg/lit	reg

Description: Shifts a specified bit field in *src/dst* right and fills the bits to the left of the shifted bit field with zeros. The *bitpos* value specifies the least significant bit of the bit field to be shifted, and the *len* value specifies the length of the bit field.

Action:

src/dst ← (*src/dst* / 2^(*bitpos* **mod** 32))
and (2^*len* - 1);

Faults: STANDARD

Example:

```
extract 5, 12, g4 # g4 ← g4 with bits 5
                  # through 16 shifted right
```

Opcode:

extract	651	REG

See Also: **modify**

FAULT IF

Mnemonic:

faulte	Fault If Equal
faultne	Fault If Not Equal
faultl	Fault If Less
faultle	Fault If Less Or Equal
faultg	Fault If Greater
faultge	Fault If Greater Or Equal
faulto	Fault If Ordered
faultno	Fault If Unordered

Format:

fault*

Description: Raises a constraint-range fault if the logical AND of the condition code and the mask-part of the opcode is not zero.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
faultno	000	Unordered
faultg	001	Greater
faulte	010	Equal
faultge	011	Greater or equal
faultl	100	Less
faultne	101	Not equal
faultle	110	Less or equal
faulto	111	Ordered

For the **faultno** instruction (unordered), the fault is raised if the condition code is equal to 000_2.

FAULT IF

Action:

For all instructions except **faultno**:

if (mask **and** AC.cc) $\neq 000_2$
then raise constraint-range fault;
end if;

faultno:

if AC.cc = 000_2
then raise constraint-range fault;
end if;

Faults: STANDARD, Constraint Range

Example:

```
# assume AC.cc AND 110₂ ≠ 000₂
faultle
# Constraint Range Fault is generated
```

Opcode:

faulte	1A	CTRL
faultne	1D	CTRL
faultl	1C	CTRL
faultle	1E	CTRL
faultg	19	CTRL
faultge	1B	CTRL
faulto	1F	CTRL
faultno	18	CTRL

See Also: **be, teste**

flushreg

Mnemonic: **flushreg** Flush Local Registers

Format: **flushreg**

Description: Copies the contents of all the cached local-register sets into their associated register-save areas in the procedure stack. The contents of all the local-register sets except for the current set are then marked as invalid. On a return, the local registers for the frame being returned to are then loaded from the stack.

The **flushreg** instruction is provided to allow a compiler or applications program to circumvent the normal call/return mechanism of the processor. For example, a compiler may need to back up several frames in the stack on the next return, rather than using the normal return mechanism that returns one frame at a time. Here, the compiler uses the **flushreg** instruction to update the stack with the current states of the saved register sets. The compiler can then return to any frame in the stack without losing the contents of the saved local-register sets. To return to a frame other than the frame directly below the current frame, the compiler merely modifies the PFP in register r0 of the current frame to point to the frame that it wishes to return to.

Action: Each register set except the current set is flushed to its associated stack frame in memory and marked as purged, meaning that they will be reloaded from memory if and when they become the current local register set.

Faults: STANDARD

Example:
```
flushreg
```

Opcode: **flushreg** 66D REG

fmark

Mnemonic: **fmark** Force Mark

Format: **fmark**

Description: Generates a breakpoint trace-event. This instruction causes a breakpoint trace-event to be generated, regardless of the setting of the breakpoint trace mode flag, providing the trace-enable bit (bit 0) of the process controls is set. (If the trace-enable flag in the process-controls register is clear, the **mark** instruction behaves like a no-op.)

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls word and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

For more information on trace-fault generation, refer to Chapter 6.

Action:

```
if process_controls.trace_enable
  then
    raise breakpoint trace fault
endif
```

Faults: STANDARD, Breakpoint Trace

Example:

```
ld xyz, r4
addi r4, r5, r6
fmark
# Breakpoint trace event is generated at
# this point in the instruction stream.
```

Opcode: **fmark** 66C REG

See Also: **mark**

LOAD

Mnemonic:

ld	Load
ldob	Load Ordinal Byte
ldos	Load Ordinal Short
ldib	Load Integer Byte
ldis	Load Integer Short
ldl	Load Long
ldt	Load Triple
ldq	Load Quad

Format:

ld*	*src*,	*dst*
	mem	reg

Description: Copies a byte or string of bytes from memory into a register or group of successive registers. The *src* operand specifies the address of the first byte to be loaded. The full range of addressing modes may be used in specifying *src*. (Refer to Chapter 8 for a complete discussion of the addressing modes available with memory-type operands.)

The *dst* operand specifies a register or the first (lowest numbered) register of successive registers.

The **ldob** and **ldib**, and **ldos** and **ldis** instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The **ld**, **ldl**, **ldt**, and **ldq** instructions copy 4, 8, 12, and 16 bytes, respectively, from memory into successive registers.

For the **ldl** instruction, *dst* must specify an even numbered register (e.g., g0, g2, ..., g12). For the **ldt** and **ldq** instructions, *dst* must specify a register number that is a multiple of four (e.g., g0, g4, g8). If the data extends beyond register g15 or r15 for the **ldl**, **ldt**, or **ldq** instruction, the results are unpredictable.

Action: *dst* ← memory (*src*);

Faults: STANDARD

Example:

```
ldl 2456 (r3), r10  # r10, r11 ← value of two
                    # words beginning at offset
                    # 2456 plus the address in
                    # r3 in memory
```

LOAD

Opcode:

ld	90	MEM
ldob	80	MEM
ldos	88	MEM
ldib	C0	MEM
ldis	C8	MEM
ldl	98	MEM
ldt	A0	MEM
ldq	B0	MEM

See Also: MOVE, STORE

lda

Mnemonic: **lda** Load Address

Format:

lda	*src*	*dst*
	mem	reg
	efa	

Description: Computes the effective address specified with *src* and stores it in *dst*. The *src* address is not checked for validity.

An important application of this instruction is to load a constant longer than 5 bits into a register. (To load a register with a constant of 5 bits or less, the move instruction (**mov**) can be used with a literal as the *src* operand.)

Action: *dst* ← efa (*src*);

Faults: STANDARD

Example:

```
lda 58(g9), g1    # Computes the effective
                  # address specified with
                  # 58(g9) and stores it in g1

lda 0x749, r8     # loads the constant 0x749
                  # in r8
```

Opcode:

lda	8C	MEM

logbnr, logbnrl

Mnemonic:

logbnr	Log Binary Real
logbnrl	Log Binary Long Real

Format:

logbnr*	*src*,	*dst*
	freg/flit	freg

Description: Calculates the $\log_2$ (*src*) and stores the integral part of this value (i.e., the part to the left of the binary point) as a real number in *dst*. The result of this operation is an unbiased exponent. When *src* is a denormalized number, *dst* is the unbiased exponent that *src* would have if the format had unlimited exponent range.

(The fractional part of $\log_2$ (*src*) is ignored. If the fractional part is needed, use the **logr** or **logrl** instruction.)

This instruction implements the IEEE recommended function *logb*. It is useful for calculating the order of magnitude of a number.

For the **logbnrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log binary of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
-∞	+∞
-F	±F
-0	**
+0	**
+F	±F
+∞	+∞
NaN	NaN

Notes:

F	means finite real number
**	indicates floating zero-divide exception

logbnr, logbnrl

Note that the significand of the *src* operand can be extracted by using the **scaler** or **scalerl** instruction.

Action:

dst ← (**log$_2$** (unbiased exponent (*src*)) - fraction);
the integral part of the unbiased exponent of *src*
is stored in *dst* as a biased real

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.
Floating Zero Divide	The *src* operand is 0.

Example:

```
logbnrl g12, fp3    # fp3 ← integral part
                    # of log2 (g12,g13)
```

Opcode:

logbnr	68A	REG
logbnrl	69A	REG

See Also: **logr, scaler**

logepr, logeprl

Mnemonic:

logepr	Log Epsilon Real	
logeprl	Log Epsilon Long Real	

Format:

logepr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Calculates (*src2* * $\log_2$ (*src1* + 1)), and stores the result in *dst*.

For the **logeprl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1				
		(1/√2)-1 to -0	**-0**	**+0**	**+0 to √2 - 1**	**NaN**
Src2	**-∞**	-∞	*	*	-∞	NaN
	-F	+F	+0	-0	-F	NaN
	-0	+0	+0	-0	-0	NaN
	+0	-0	-0	+0	+0	NaN
	+F	-F	-0	+0	+F	NaN
	+∞	+∞	*	*	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.

This instruction offers optimal accuracy for values of *src1* + 1 close to 1 (i.e., for values of *src1* close to 0). This expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in *src2*.

logepr, logeprl

The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result stored in *dst*:

$$\text{scale factor} = \log_n 2$$

The range of *src1* is restricted to the following:

$$1/\text{sqrt}(2) \leq src1 + 1 \leq \text{sqrt}(2)$$

When the *src1* operand is outside this range, the **logr** or **logrl** instruction can be used with very insignificant loss of accuracy by adding 1.0 to *src1*.

Action:

dst ← *src2* * $\mathbf{log_2}$ (*src1* + 1);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src1* operand is 0 and the *src2* operand is ∞. The *src1* operand does not fall within the range defined in the above description section. One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

logepr, logeprl

Example:

```
logepr g8, g4, fp2
# fp2 ← g4,g5 * log2 (g8,g9 + 1)
```

Opcode:

logepr	681	REG
logeprl	691	REG

See Also: **logr**

logr, logrl

Mnemonic:

logr	Log Real
logrl	Log Long Real

Format:

logr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Calculates (*src2* * $\log_2$ (*src1*)), and stores the result in *dst*. (The **logbnr** and **logbnrl** instructions perform this function more efficiently, if only an estimate is needed.)

For the **logrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src2 \ Src1	-∞	-F	-0	+0	+F	+∞	NaN
-∞	*	*	**	**	±∞	-∞	NaN
-F	*	*	**	**	±F	-∞	NaN
-0	*	*	*	*	±0	*	NaN
+0	*	*	*	*	±0	*	NaN
+F	*	*	**	**	±F	+∞	NaN
+∞	*	*	**	**	±∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.
**	Indicates floating zero-divide exception.

The **logr** instruction combined with the **expr** instruction forms the basis for the power function x^y.

logr, logrl

Adding 1.0 to a number to be used as the *src1* operand will cause information to be lost. To perform this function, use the **logepr** or **logeprl** instruction.

These instructions provide a simple method of converting the result of the $\log_2$ arithmetic into a value in another logarithm base by including a scale factor in *src2*. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result stored in *dst*;

$$\text{scale factor} = \log_n 2$$

Action:

dst ← *src2* * $\mathbf{log_2}$ (*src1*);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Zero Divide	The *src1* operand is 0 and *src2* is non-zero.
Floating Invalid Operation	The *src1* and *src2* operands are both 0.
	The *src1* operand is ∞ and the *src2* operand is 0.
	The *src1* operand is 1 and the *src2* operand is ∞.
	The *src1* operand is negative and nonzero.
	One or more operands are an SNaN value.

logr, logrl

	Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
logrl r2, g8, g2
# g2,g3 ← g8,g9 * log2(r2,r3)
```

Opcode:

logr	682	REG
logrl	692	REG

See Also: **expr, logepr**

mark

Mnemonic: **mark** Mark

Format: **mark**

Description: Generates a breakpoint trace event if the breakpoint trace mode has been enabled. The breakpoint trace mode is enabled if the trace-enable bit (bit 0) of the process controls and the breakpoint-trace mode bit (bit 7) of the trace controls have been set.

When a breakpoint trace event is detected, the trace-fault-pending flag (bit 10) of the process controls and the breakpoint-trace-event flag (bit 23) of the trace controls are set. Before the next instruction is executed, a trace fault is generated.

If the breakpoint-trace mode has not been enabled, the **mark** instruction behaves like a no-op.

For more information on trace-fault generation, refer to Chapter 7.

Action:

if process_controls.trace_enable **and** breakpoint_trace_flag
  then
    raise trace breakpoint fault **endif**

Faults: STANDARD, Breakpoint Trace

Example:

```
# Assume that the breakpoint trace mode is
# enabled.
ld xyz, r4
addi r4, r5, r6
mark
# Breakpoint trace event is generated at
# this point in the instruction stream.
```

Opcode: **mark** 66B REG

See Also: **fmark**, **modpc**, **modtc**

modac

Mnemonic: **modac** Modify AC

Format:

modac	*mask*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Reads and modifies the arithmetic controls. The *src* operand contains the value to be placed in the arithmetic controls and the *mask* operand specifies the bits that may be changed. Only the bits set in *mask* are modified in the arithmetic controls. Once the arithmetic controls have been changed, their initial state is copied into *dst*.

Action:

temp ← AC
AC ← (*src* **and** *mask*) **or**
(AC **and not** (*mask*));
dst ← temp;

Faults: STANDARD

Example:

```
modac g1, g9, g12  # AC ← g9, masked by g1
                   # g12 ← initial value of AC
```

Opcode:

modac	645	REG

See Also: **modpc, modtc**

modi

Mnemonic: **modi** Modulo Integer

Format:

modi	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Divides *src2* by *src1*, where both are integers, and stores the modulo remainder of the result in *dst*. If the result is nonzero, *dst* has the same sign as *src1*.

Action:
if (*src1* = 0) Arithmetic Zero Divide fault;
dst ← *src2* - ((*src2*/*src1*) * *src1*);
if ((*src2* * *src1* < 0) **and** (*dst* ≠ 0)) *dst* ← *dst* + *src1*;
src1, *src2* and *dst* are 32 bits

Faults: STANDARD, Arithmetic Zero Divide

Example:
```
modi r9, r2, r5    # r5 ← modulo (r2/r9)
```

Opcode:

modi	749	REG

See Also: **div, remi**

modify

Mnemonic: **modify** Modify

Format:

modify	*mask*,	*src*,	*src/dst*
	reg/lit	reg/lit	reg

Description: Modifies selected bits in *src/dst* with bits from *src*. The *mask* operand selects the bits to be modified: only the bits set in the *mask* operand are modified in *src/dst*.

Action: *src/dst* ← (*src* **and** *mask*) **or** (*src/dst* **and not** (*mask*));

Faults: STANDARD

Example:
```
modify g8, g10, r4    # r4 ← g10 masked by g8
```

Opcode: **modify** 650 REG

See Also: **alterbit, extract**

modpc

Mnemonic: **modpc** Modify Process Controls

Format:

modpc	*src*,	*mask*,	*src/dst*
	reg/lit	reg/lit	reg

Description: Reads and modifies the process controls as specified with *mask* and *src/dst*. The *src/dst* operand contains the value to be placed in the process controls and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified in the process controls. Once the process controls have been changed, their initial value is copied into *src/dst*. The *src* operand is a dummy operand that should be set equal to the *mask* operand.

The processor must be in the supervisor mode to modify the process controls using this instruction. If the *mask* operand is set to 0, this instruction can be used to read the process controls, without the processor being in the supervisor mode.

If the action of this instruction results in the priority of the processor being lowered, the interrupt table is checked for pending interrupts.

Changing the state, resume, internal state, and trace enable fields of the process controls can lead to unpredictable behavior, as described in Chapter 3 in the section titled "Changing the Process-Controls."

If **modpc** is used to change the trace-enable bit, the processor may not recognize the change before the next four instructions have been executed.

Action:

```
if ((mask ≠ 0)
        {
        if (PC.em ≠ supervisor)) Type-mismatch fault;
        temp ← PC;
        PC ← (mask and src/dst) or (PC and not (mask));
        src/dst ← temp;
        if (temp.p > PC.p) check_pending_interrupts;
        }
else src/dst ← PC;
endif;
```

modpc

Faults: STANDARD, Type Mismatch

Example:
```
modpc g9, g9, g8    # process controls ← g8
                    # masked by g9
```

Opcode: **modpc** 655 REG

See Also: **modac, modtc**

modtc

Mnemonic: **modtc** Modify Trace Controls

Format:

modtc	*mask*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Reads and modifies the trace controls as specified with *mask* and *src*. The *src* operand contains the value to be placed in the trace controls and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified in the trace controls. Once the trace controls have been changed, their initial state is copied into *dst*.

Since bits 8 through 15 and 24 through 31 of the trace-controls word are reserved, the *mask* operand is ANDed with $00FF00FF_{16}$ to insure that these bits are not set in the mask.

The changed trace controls take effect on the first non-branching instruction fetched from memory. Since instructions are prefetched four at a time, the trace controls may not take effect for up to the next four instructions executed.

For more information on the trace controls, refer to Chapter 7.

Action:

```
temp ← trace_controls;
temp1 ← 00FF00FF16 and mask;
trace_controls ←
   (temp1 and src) or
   (trace_controls and not(temp1));
dst ← temp;
```

Faults: STANDARD

Example:

```
modtc g12, g10, g2
# trace controls ← g10 masked by g12;
# previous trace controls stored in g2
```

Opcode: **modtc** 654 REG

See Also: **modac**, **modpc**

MOVE

Mnemonic:

mov	Move
movl	Move Long
movt	Move Triple
movq	Move Quad

Format:

mov*	*src*,	*dst*
	reg/lit	reg

Description: Copies the content of one or more source registers (specified with the *src* operand) to one or more destination registers (specified with the *dst* operand).

For the **movl**, **movt**, and **movq** instructions, the *src* and *dst* operands specify the first (lowest numbered) register of several successive registers. The *src* and *dst* registers must be even numbered (e.g., g0, g2) for the **movl** instruction and an integral multiple of four (e.g., g0, g4) for the **movt** and **movq** instructions.

When the *src* and *dst* operands overlap, the value moved is unpredictable.

Action: *dst* ← *src*;

Faults: STANDARD

Example:

```
movt g8, r4  # r4, r5, r6 ← g8, g9, g10
```

Opcode:

mov	5CC	REG
movl	5DC	REG
movt	5EC	REG
movq	5FC	REG

See Also: **ld, movr, st**

movr, movre, movrl

Mnemonic:

movr	Move Real
movrl	Move Long Real
movre	Move Extended Real

Format:

movr*	*src*,	*dst*
	freg/flit	freg

Description: Copies a real value from one or more source registers (specified with the *src* operand) to one or more destination registers (specified with the *dst* operand).

For the **movrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. For the **movre** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of three successive registers.

When copying real numbers between global or local registers and floating-point registers, conversion between real or long-real format to extended-real format is performed implicitly. Conversion between real and long-real formats must be done through floating-point registers and requires two instructions, as illustrated in the example below.

When the **movre** instruction moves an operand from global or local registers to a floating-point register, it automatically truncates the most-significant 16 bits of the word in the third register (refer to Figure 10-5). Likewise, when this instruction is used to move an operand from a floating-point register to global or local registers, it adds 16 zeros to the third word. The **movre** instruction is not a numeric instruction; it merely manipulates bits.

The **movr** and **movrl** instructions can cause a floating-point exception to be raised, which might result in a fault being raised, as is explained in the section below on faults. The **movre** instruction can never raise an exception and thus never faults.

Action: *dst* ← *src*;

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The source operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

movr, movre, movrl

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	Source operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
# Conversion of real value in g3
# to a long real value, which is
# stored in g4,g5
movr g3, fp2
movrl fp2, g4
```

Opcode:

movr	6C9	REG
movrl	6D9	REG
movre	6E1	REG

See Also: **mov**

muli, mulo

Mnemonic:

muli	Multiply Integer
mulo	Multiply Ordinal

Format:

mul*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst*.

Action: *dst* ← *src2* * *src1*;

Faults: STANDARD, Integer Overflow

Example:

```
muli r3, r4, r9    # r9 ← r4 * r3
```

Opcode:

muli	741	REG
mulo	701	REG

See Also: **emul, mulr**

mulr, mulrl

Mnemonic:

mulr	Multiply Real
mulrl	Multiply Long Real

Format:

mulr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Multiplies the *src2* value by the *src1* value and stores the result in *dst*.

For the **mulrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0, ∞, or a NaN.

The following table shows the results obtained when multiplying various classes of numbers together, assuming that neither overflow nor underflow occurs.

Src2 \ Src1	-∞	-F	-0	+0	+F	+∞	NaN
-∞	+∞	+∞	*	*	-∞	-∞	NaN
-F	+∞	+F	+0	-0	-F	-∞	NaN
-0	*	+0	+0	-0	-0	*	NaN
+0	*	-0	-0	+0	+0	*	NaN
+F	-∞	-F	-0	+0	+F	+∞	NaN
+∞	-∞	-∞	*	*	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.

When you need to multiply by the power of 2, the **scaler** and **scalerl** instructions can also be used.

mulr, mulrl

Action: *dst* ← *src2* * *src1*;

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	One source operand is 0 and the other is ∞.
	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
mulrl g12, g4, fp2    # fp2 ← g4,g5 * g12,g13
```

Opcode:

mulr	78C	REG
mulrl	79C	REG

See Also: **emul, muli, scaler**

nand

Mnemonic: **nand** Nand

Format:

nand	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise NAND operation on the *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← (**not** (*src2*)) **or** (**not** (*src1*));

Faults: STANDARD

Example:
```
nand g5, r3, r7    # r7 ← r3 NAND g5
```

Opcode:

nand	58E	REG

See Also: **and, andnot, nor, not, notand, notor, or, ornot, xnor, xor**

nor

Mnemonic: **nor** Nor

Format:

nor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise NOR operation on the *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← (**not** (*src2*)) **and** (**not** (*src1*));

Faults: STANDARD

Example: nor g8, 28, r5 # r5 ← 28 NOR g8

Opcode: **nor** 588 REG

See Also: **and, andnot, nand, not, notand, notor, or, ornot, xnor, xor**

not, notand

Mnemonic:

not	Not
notand	Not And

Format:

not	*src*,	*dst*	
	reg/lit	reg	
notand	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise NOT (**not** instruction) or NOT AND (**notand** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

not: *dst* ← **not** (*src*);

notand: *dst* ← (**not** (*src2*)) **and** *src1*;

Faults: STANDARD

Example:

```
not g2, g4            # g4 ← NOT g2
notand r5, r6, r7     # r7 ← NOT r6 AND r5
```

Opcode:

not	58A	REG
notand	584	REG

See Also: **and, andnot, nand, nor, notor, or, ornot, xnor, xor**

notbit

Mnemonic: **notbit** Not Bit

Format:

notbit	*bitpos*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Copies the *src* value to *dst* with one bit toggled. The *bitpos* operand specifies the bit to be toggled.

Action: *dst* ← *src* **xor** 2^(*bitpos* **mod** 32);

Faults: STANDARD

Example:

```
notbit r3, r12, r7    # r7 ← r12 with the bit
                      # specified in r3 toggled
```

Opcode: **notbit** 580 REG

See Also: **alterbit, chkbit, clrbit, setbit**

notor

Mnemonic: **notor** Not Or

Format:

notor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise NOT OR operation on the *src2* and *src1* values and stores the result in *dst*.

Action: *dst* ← (**not** (*src2*)) **or** *src1*;

Faults: STANDARD

Example:
```
notor g12, g3, g6  # g6 ← NOT g3 OR g12
```

Opcode: **notor** 58D REG

See Also: **and, andnot, nand, nor, not, notand, or, ornot, xnor, xor**

or, ornot

Mnemonic:

or	Or
ornot	Or Not

Format:

or	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg
ornot	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise OR (**or** instruction) or ORNOT (**ornot** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

or: *dst* ← *src2* **or** *src1*;

ornot: *dst* ← *src2* **or** (**not** (*src1*));

Faults: STANDARD

Example:

```
or 14, g9, g3          # g3 ← g9 OR 14
ornot r3, r8, r11      # r11 ← r8 OR NOT r3
```

Opcode:

or	587	REG
ornot	58B	REG

See Also: **and, andnot, nand, nor, not, notand, notor, xnor, xor**

remi, remo

Mnemonic:

remi	Remainder Integer
remo	Remainder Ordinal

Format:

rem*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Divides *src2* by *src1* and stores the remainder in *dst*. The sign of the result (if nonzero) is the same as the sign of *src2*.

Action: *dst* ← *src2* - ((*src2* / *src1*) * *src1*);

Faults:

STANDARD	Refer to discussion of faults at the beginning of this chapter.
Integer Overflow	Result is too large for destination format. This fault is signaled only when executing the **remi** instruction and if both of the following conditions are met: (1) the integer-overflow mask in the arithmetic-controls registers is clear and (2) the source operands have like signs and the sign of the result operand is different than the signs of the source operands.

Example:

```
remo r4, r5, r6    # r6 ← r5 rem r4
```

Opcode:

remi	748	REG
remo	708	REG

See Also: **remr**, **modi**

remr, remrl

Mnemonic:

remr	Remainder Real	
remrl	Remainder Long Real	

Format:

remr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Divides *src2* by *src1* and stores the remainder in *dst*. The sign of the result (if nonzero) is the same as the sign of *src2*.

For the **remrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
		-∞	**-F**	**-0**	**+0**	**+F**	**+∞**	**NaN**
Src2	**-∞**	*	*	*	*	*	*	NaN
	-F	src2	-F or -0	**	**	-F or -0	src2	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	src2	+F or +0	**	**	+F or +0	src2	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.
**	Indicates floating zero-divide exception.

When the result is 0, its sign is the same as that of *src2*. When the *src1* is ∞, the result is equal to the *src2*.

remr, remrl

The result of this operation is always exact if the destination format is at least as wide as the *src2* and *src1*.

The remainder provided with the **remr** and **remrl** instructions is different from the remainder described in the IEEE floating-point standard. The difference is related to how the quotient (N) of the expression (*src2/src1*) is determined.

As shown below in the action statement, N for the **remr** and **remrl** instructions is the nearest integer value obtained when the exact result (E) of the expression (*src2/src1*) is truncated toward zero. N will always be less than or equal to the absolute value of E.

For the IEEE standard, N is simply the nearest integer value to E. Here, N may be less than, equal to, or greater than the absolute value of E.

To help determine the IEEE remainder from the result given by the **remr** and **remrl** instructions, the following information about the quotient is given in the arithmetic-status field in the arithmetic controls:

Arithmetic Status Bit	Meaning
6	Q1, the next-to-last quotient bit
5	Q0, the last quotient bit
4	QR, the value the next quotient bit would have if one more reduction were performed (the "round" bit of the quotient)
3	QS, set if the remainder after the QR reduction would be nonzero (the "sticky" bit of the quotient)

The information can then be used to determine the IEEE standard remainder, as shown in the example on the next page.

Action:

dst ← *src2* - (N * *src1*);
where N = truncate (*src2/src1*).
Here, (*src2/src1*) is truncated
toward zero to the nearest integer.

remr, remrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Zero Divide	The *src1* operand is 0.
Floating Invalid Operation	The *src2* operand is ∞.
	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
# z = ieee_rem(x, y)
# z is in g0,g1; x is in g0,g1; y is in g2,g3
_ieee_rem:
    remrl g2, g0, g0
    modac 0, 0, g4
    bbc 4, g4, 2f
    # QR=0, implies g0 < y/2 and z=g0
    bbs 3, g4, 1f
    # QR=1,QS=1, implies g0 > y/2 and z=g0-y
    bbc 5, g4, 2f
    # QR=1,QS=0,Q0=0, implies g0=y/2 and z=g0
1: clrbit 31, g3, g2     # |y|
   subrl g2, g0, g0
2: ret
```

Opcode:

remr	683	REG
remrl	693	REG

See Also: **remi, modi**

ret

Mnemonic: **ret** Return

Format: **ret**

Description: Returns process control to the calling procedure. The current stack frame (i.e., that of the called procedure) is deallocated and the FP is changed to point to the stack frame of the calling procedure. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the return status field and prereturn trace flag determine the action that the processor takes on the return. These fields are contained in bits 0 through 3 of register r0 of the calling procedure's local registers.

Refer to Chapter 4 for further discussion of the **ret** instruction.

Action:

```
wait for any uncompleted instructions to finish;
case return_status is

000₂: FP ← PFP;
      free current register_set;
      if register_set (FP) not allocated
        then retrieve from memory(FP);
      end if;
      IP ← RIP;

001₂: x ← memory(FP-16);
      y ← memory(FP-12);
      go to case 000₂ action;
      arithmetic_controls ← y;
      if execution_mode = supervisor
        then process_controls ← x;
      end if;

010₂: if execution_mode ≠ supervisor
        then go to case 000₂ action;
        else process_controls.T ← 0;
          execution_mode ← user;
          go to case 000₂ action;
        end if;
```

ret

011_2: **if** execution_mode ≠ supervisor
then go to case 000_2 action;
else process_controls.T ← 1;
execution_mode ← user;
go to case 000_2 action;
end if;

100_2: undefined

101_2: undefined

110_2: undefined

111_2: x ← memory(FP-16);
y ← memory(FP-12);
go to case 000_2 action;
arithmetic_controls ← y;
if execution_mode = supervisor
then process_controls ← x;
check_pending_interrupts;
end if;

Faults: STANDARD

Example:
```
ret     # process control returns to
        # calling procedure
        # environment
```

Opcode:

ret	0A	CTRL

See Also: **call, calls, callx**

rotate

Mnemonic: **rotate** Rotate

Format:

rotate	*len*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Copies *src* to *dst* and rotates the bits in the resulting *dst* operand to the left (toward higher significance). (The bits shifted off the left end of the word are inserted at the right end of the word.) The *len* operand specifies the number of bits that the *dst* operand is rotated. The *len* operand can range from 0 to 31.

This instruction can also be used to rotate bits to the right. Here, the number of bits the word is to be rotated right is subtracted from 32 to get the *len* operand.

Action: *dst* ← **rotate** (*len* **mod** 32 (*src*))

Faults: STANDARD

Example:

```
rotate r4, r8, r12    # r12 ← r8
                      # with bits rotated
                      # r4 bits to left
```

Opcode:

rotate	59D	REG

See Also: **SHIFT**

roundr, roundrl

Mnemonic:

roundr	Round Real
roundrl	Round Long Real

Format:

roundr*	*src*,	*dst*
	freg/flit	freg

Description: Rounds *src* to the nearest integral value, depending on the rounding mode, and stores the result in *dst*.

For the **roundrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

If the *src* operand is ∞ the result is *src*. If the *src* operand is not an integral value, a floating-inexact exception is raised.

Action:

dst ← round_to_integral_value (*src*);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
roundrl r4, r10
# r10,r11  ← r4,r5 rounded
```

Opcode:

roundr	68B	REG
roundrl	69B	REG

scaler, scalerl

Mnemonic:

scaler	Scale Real
scalerl	Scale Long Real

Format:

scaler*	*src1*,	*src2*,	*dst*
	reg/lit	freg/flit	freg

Description: Multiplies *src2* by 2 to the power of *src1* and stores the result in *dst*. The *src1* operand is an integer; whereas, *src2* and *dst* are reals.

For the **scalerl** instruction, if the *src2* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1		
		-N	**0**	**+N**
Src2	**-∞**	-∞	-∞	-∞
	-F	-F	-F	-F
	-0	-0	-0	-0
	+0	+0	+0	+0
	+F	+F	+F	+F
	+∞	+∞	+∞	+∞
	NaN	NaN	NaN	NaN

Notes:
F means finite-real number.
N means integer.

In most cases, only the exponent is changed and the mantissa (fraction) remains unchanged. However, when the *src2* operand is a denormalized value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

scaler, scalerl

Refer to the sections titled "Floating Overflow Exception" and "Floating Underflow Exception" in Chapter 10 for further discussion of how overflow and underflow are handled.

Action: *dst* ← *src2* * (2^*src1*)

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src2* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src2* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
scalerl g6, g2, fp0
# fp0 ← g2,g3 * 2^g6
```

Opcode:

scaler	677	REG
scalerl	676	REG

See Also: **mulr**

scanbit

Mnemonic: **scanbit** Scan For Bit

Format:

scanbit	*src*,	*dst*
	reg/lit	reg

Description: Searches the *src* value for the most-significant set bit (1 bit). If a most-significant 1 bit is found, its bit number is stored in *dst* and the condition code is set to 010_2. If the *src* value is zero, all 1's are stored in *dst* and the condition code is set to 000_2.

Action:

dst ← $FFFFFFFF_{16}$;
AC.cc ← 000_2;
for i in 31..0 **reverse loop**
    if (*src* **and** 2^i) ≠ 0
    then
        dst ← i;
        AC.cc ← 010_2;
        exit;
    end if;
end loop;

Faults: STANDARD

Example:

```
# assume g8 is nonzero
scanbit g8, g10
# g10 ← bit number of
# most-significant set bit
# in g8; AC.cc ← 010₂
```

Opcode:

scanbit	641	REG

See Also: **spanbit**

scanbyte

Mnemonic: **scanbyte** Scan Byte Equal

Format:

scanbyte	*src1*,	*src2*
	reg/lit	reg/lit

Description: Performs a byte-by-byte comparison of *src1* and *src2* and sets the condition code to 010_2 if any two corresponding bytes are equal. If no corresponding bytes are equal, the condition code is set to 000_2.

Action:

if (*src1* **and** $000000FF_{16}$) = (*src2* **and** $000000FF_{16}$) **or**
(*src1* **and** $0000FF00_{16}$) = (*src2* **and** $0000FF00_{16}$) **or**
(*src1* **and** $00FF0000_{16}$) = (*src2* **and** $00FF0000_{16}$) **or**
(*src1* **and** $FF000000_{16}$) = (*src2* **and** $FF000000_{16}$)
  then AC.cc ← 010_2;
  else AC.cc ← 000_2;
endif;

Faults: STANDARD

Example:

```
# assume r9 = 11AB1100₁₆
scanbyte 0x00AB0011, r9
# AC.cc ← 010₂
```

Opcode:

scanbyte	5AC	REG

setbit

Mnemonic: **setbit** Set Bit

Format:

setbit	*bitpos*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Copies the *src* value to *dst* with one bit set. The *bitpos* operand specifies the bit to be set.

Action: *dst* ← *src* **or** 2^(*bitpos* **mod** 32);

Faults: STANDARD

Example:

```
setbit 15, r9, r1
# r1 ← r9 with bit 15 set
```

Opcode: **setbit** 583 REG

See Also: **alterbit, chkbit, clrbit, notbit**

SHIFT

Mnemonic:

shlo	Shift Left Ordinal
shro	Shift Right Ordinal
shli	Shift Left Integer
shri	Shift Right Integer
shrdi	Shift Right Dividing Integer

Format:

sh*	*len*,	*src*,	*dst*
	reg/lit	reg/lit	reg

Description: Shifts *src* left or right by the number of bits indicated with the *len* operand and stores the result in *dst*. Bits shifted beyond the register boundary are discarded. For values of *len* greater than 32, the processor interprets the value as 32.

The **shlo** instruction shift zeros in from the least-significant bit, and the shro instruction shifts zeros in from the most-significant bit. These instructions are equivalent to **mulo** and **divo** by the power of 2, respectively.

The **shli** instruction shifts zeros in from the least-significant bit; if the bits shifted out are not the same as the sign bit, an overflow fault is generated. If overflow occurs, the sign of the result is the same as the sign of the *src* operand.

The **shri** instruction performs a conventional arithmetic shift-right operation by shifting the sign bit in from the most-significant bit. When this instruction is used to divide a negative integer operand by the power of 2, it produces an incorrect quotient. (The discarding of the bits shifted out has the effect of rounding the result toward negative.)

The **shrdi** instruction is provided for dividing integers by the power of 2. With this instruction, 1 is added to the result if the bits shifted out are non-zero and the operand is negative, which produces the correct result for negative operands.

The **shli** and **shrdi** instructions are equivalent to **muli** and **divi** by the power of 2.

SHIFT

Action:

shlo:
if *len* < 32
then *dst* ← *src** 2^*len*;
else *dst* ← 0;
end if;

shro:
if *len* < 32
then *dst* ← *src*/2^*len*;
else *dst* ← 0;
end if;

shli:
dst ← *src** 2^*len*;

shri:
if *src* ≥ 0
then if *len* < 32
  then *dst* ← *src*/2^*len*;
  else *dst* ← 0;
else if *len* < 32
  then *dst* ← (*src* - 2^*len* + 1)/2^*len*;
  else *dst* ← -1;
  end if;
end if;

shrdi:
dst ← *src*/2^*len*;

Faults: STANDARD, Integer Overflow

Example:

```
shli 13, g4, r6
# g6 ← g4 shifted left 13 bits
```

Opcode:

shlo	59C	REG
shro	598	REG
shli	59E	REG
shri	59B	REG
shrdi	59A	REG

See Also: **divi, muli, rotate**

sinr, sinrl

Mnemonics:

sinr	Sine Real
sinrl	Sine Long Real

Format:

sinr*	*src*,	*dst*
	freg/flit	freg

Description: Calculates the sine of *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range -1 to +1, inclusive.

For the **sinrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the sine of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
-∞	*
-F	-1 to +1
-0	-0
+0	+0
+F	-1 to +1
+∞	*
NaN	NaN

Notes:
F means finite-real number
* Indicates floating invalid-operation exception

In the trigonometric instructions, the 80960SA/SB uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 10 titled "Pi" gives this π value, along with some suggestions for representing this value in a program.

Action:

dst ← **sine** (*src*);

sinr, sinrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Invalid Operation	The *src* operand is ∞.
	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
sinrl g6, g0
# sine of value in g6,g7
# is stored in g0,g1
```

Opcode:

sinr	68C	REG
sinrl	69C	REG

See Also: **cosr, tanr**

spanbit

Mnemonic: **spanbit** Span Over Bit

Format:

spanbit	*src*,	*dst*
	reg/lit	reg

Description: Searches the *src* value for the most-significant clear bit (0 bit). If a most-significant 0 bit is found, its bit number is stored in *dst* and the condition code is set to 010_2. If the *src* value is all 1's, all 1's are stored in *dst* and the condition code is set to 000_2.

Action:

dst ← $FFFFFFFF_{16}$;
AC.cc ← 000_2;
for i in 31..0 **reverse loop**
  if (*src* and 2^i) = 0
  then
    dst ← i;
    AC.cc ← 010_2;
    exit;
  end if;
end loop;

Faults: STANDARD

Example:

```
# assume r2 is not ffffffff16
spanbit r2 r9
# r9 ← bit number of
# most-significant clear bit
# in r2; AC.cc ← 0102
```

Opcode:

spanbit	640	REG

See Also: **scanbit**

sqrtr, sqrtrl

Mnemonic:

sqrtr	Square Root Real
sqrtrl	Square Root Long Real

Format:

sqrtr*	*src*,	*dst*
	freg/flit	freg

Description: Calculates the square root of *src* and stores it in *dst*.

For the **sqrtrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
-∞	*
-F	*
-0	-0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

Notes:
F Means finite-real number
* Indicates floating invalid-operation exception

With these instructions, it is not possible to raise a floating overflow or floating underflow fault unless the *src* operand is in a floating-point register and the *dst* operand is not.

Action: *dst* ← sqrt (*src*);

sqrtr, sqrtrl

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src* operand is less than -0.
	The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
sqrtrl g6, fp0
# fp0 ← sqrt of g6,g7
```

Opcode:

sqrtr	688	REG
sqrtrl	698	REG

STORE

Mnemonic:

st	Store
stob	Store Ordinal Byte
stos	Store Ordinal Short
stib	Store Integer Byte
stis	Store Integer Short
stl	Store Long
stt	Store Triple
stq	Store Quad

Format:

st*	*src*,	*dst*
	reg	mem

Description: Copies a byte or string of bytes from a register or group of registers to memory. The *src* operand specifies a register or the first (lowest numbered) register of successive registers.

The *dst* operand specifies the address of the memory location where the byte or the first byte of a string of bytes is to be stored. The full range of addressing modes may be used in specifying *dst*. (Refer to Chapter 8 for a complete discussion of the addressing modes available with memory-type operands.)

The **stob** and **stib**, and **stos** and **stis** instructions store a byte and half word, respectively, from the low order bytes of the *src* register. The **st**, **stl**, **stt**, and **stq** instructions copy 4, 8, 12, and 16 bytes, respectively, from successive registers to memory.

For the **stl** instruction, *src* must specify an even numbered register (e.g., g0, g2, ..., g12). For the **stt** and **stq** instructions, *src* must specify a register number that is a multiple of four (e.g., g0, g4, g8).

Action: memory (*dst*) ← *src*;

Faults: STANDARD, Integer Overflow Fault (**stib** and **stis** instructions only)

Example:

```
st g2, 1256 (g6)
# word beginning at offset
# 1256 + (g6) ← g2
```

STORE

Opcode:

st	92	MEM
stob	82	MEM
stos	8A	MEM
stib	C2	MEM
stis	CA	MEM
stl	9A	MEM
stt	A2	MEM
stq	B2	MEM

See Also: LOAD, MOVE

subc

Mnemonic: **subc** Subtract Ordinal With Carry

Format:

subc	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2*, adds bit 1 of the condition code (used here as a carry bit), and stores the result in *dst*. If the ordinal subtraction results in a carry, bit 1 of the condition code is set.

This instruction can also be used for integer subtraction. Here, if integer subtraction results in an overflow, bit 0 of the condition code is set.

The **subc** instruction does not distinguish between ordinals and integers: it sets bits 0 and 1 of the condition code regardless of the data type.

Action:

```
# Let the value of the condition code by xCx.
dst ← src2 - src1 + C;
AC.cc ← 0CV2;
# C is carry from ordinal subtraction.
# V is 1 if integer subtraction would have generated
# an overflow.
```

Faults: STANDARD

Example:

```
subc g5, g6, g7
# g7 ← g6 - g5 + Carry Bit
```

Opcode:

subc	5B2	REG

See Also: **addc**

subi, subo

Mnemonic:

subi	Subtract Integer
subo	Subtract Ordinal

Format:

sub*	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Subtracts *src1* from *src2* and stores the result in *dst*. The binary results from these two instructions are identical. The only difference is that **subi** can signal an integer overflow.

Action: *dst* ← *src2* - *src1*;

Faults: STANDARD, Integer Overflow (**subi** instruction only)

Example:

```
subi g6, g9, g12   # g12 ← g9 - g6
```

Opcode:

subi	593	REG
subo	592	REG

See Also: **addi, addr, subc, subr**

subr, subrl

Mnemonic:

subr	Subtract Real
subrl	Subtract Long Real

Format:

subr*	*src1*,	*src2*,	*dst*
	freg/flit	freg/flit	freg

Description: Subtracts *src1* from *src2* and stores the result in *dst*.

For the **subrl** instruction, if the *src1*, *src2*, or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when subtracting various classes of numbers, assuming that neither overflow nor underflow occurs.

		Src1						
		-∞	**-F**	**-0**	**+0**	**+F**	**+∞**	**NaN**
Src2	**-∞**	*	-∞	−∞	−∞	−∞	−∞	NaN
	-F	+∞	±F or ±0	src2	src2	-F	−∞	NaN
	-0	+∞	src1	±0	-0	src1	−∞	NaN
	+0	+∞	src1	+0	±0	src1	−∞	NaN
	+F	+∞	+F	src2	src2	±F or ±0	−∞	NaN
	+∞	+∞	+∞	+∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

F	Means finite real number.
*	Indicates floating invalid-operation exception.

When the difference between two operands of like sign is zero, the result is +0, except for the round toward -∞ mode, in which case the result is -0. This instruction also guarantees that +0 - (-0) = +0, and that -0 - (+0) = -0.

When one source operand is ∞, the result is ∞ of the expected sign. If both source operands are ∞ of the same sign, an invalid-operation exception is raised.

subr, subrl

Action: *dst* ← *src2* - *src1*;

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	One or more operands is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	Source operands are infinities of like sign.
	One or more operands are an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
subrl g6, fp0, fp1
# fp1 ← fp0 - g6,g7
```

Opcode:

subr	78D	REG
subrl	79D	REG

See Also: **subi, subc, addr**

syncf

Mnemonic: **syncf** Synchronize Faults

Format: **syncf**

Description: Waits for any faults to be generated associated with any prior uncompleted instructions.

NOTE

In the 80960SA/SB implementation of the i960 architecture, this instruction acts as a no-op.

Action:

```
if arithmetic_controls.nif
        then;
        else wait until no imprecise faults can occur
                        associated with any uncompleted instructions;
end if;
```

Faults: STANDARD

Example:

```
ld xyz, g6
addi r6, r8, r8
syncf
and g6, 0x11, g8
# the syncf instruction insures that any faults
# that may occur during the execution of the
# ld and addi instructions occur before the
# and instruction is executed
```

Opcode: **syncf** 66F REG

See Also: **mark, fmark**

synld

Mnemonic: **synld** Synchronous Load

Format:

synld	*src*,	*dst*
	reg	reg
	addr	

Description: Copies a word from the memory location specified with *src* into *dst* and waits for the completion of all memory operations, including those initiated prior to the **synld** instruction. When the load has been successfully completed, the condition code is set to 010_2.

The primary function of this instruction is for reading the Interrupt Control Register. It may be used when it is important to guarantee the completion of the load operation before proceeding.

The setting of the condition code indicates whether or not the load was completed successfully. If the load operation results in a bad access condition, the condition code is set to 000_2.

Action:

```
tempa ← src and FFFFFFFC16; # force alignment
if tempa = FF00000416
  then dst ← interrupt_control_reg;
    AC.cc ← 0102;
  else dst ← memory (tempa);
    AC.cc ← 0102;
end if;
```

Faults: STANDARD

Example:

```
lda 0xff000004, g8
# g8 ← address of interrupt-control register
synld g8, g9
# g9 ← contents of interrupt-control register
# AC.cc = 0102
```

Opcode:

synld	615	REG

See Also: **synmov**

synmov, synmovl, synmovq

Mnemonic:

synmov	Synchronous Move
synmovl	Synchronous Move Long
synmovq	Synchronous Move Quad

Format:

synmov*	*dst*,	*src*
	reg	reg
	addr	addr

Description: Copies 1 (**synmov**), 2 (**synmovl**), or 4 (**synmovq**) words from the memory location specified with *src* to the memory location specified with *dst* and waits for the completion of all memory operations, including those initiated prior to this instruction. When the move has been successfully completed, the condition code is set to 010_2.

The *src* and *dst* operands specify the address of the first (lowest address) word. These addresses should be for word boundaries (**synmov**), double-word boundaries (**synmovl**), or quad-word boundaries (**synmovq**). If not, the processor forces alignment to these boundaries.

The primary function of these instructions is for sending IAC messages. However, this instruction is not restricted to IAC applications. It may be used when it is important to guarantee the completion of the move operation before proceeding.

The setting of the condition code indicates whether or not the move was completed successfully. If the move operation results in a bad access condition (e.g., sending an IAC message to a wrong address, the condition code is set to 000_2.

Address $FF000010_{16}$ is used to send an IAC message. Refer to Chapter 11 for further information about sending IAC messages.

synmov, synmovl, synmovq

Action:

synmov:

```
tempa ← dst and FFFFFFFC16; # force alignment
if tempa = FF00000416
  then interrupt_control_reg ← memory (src)
    AC.cc ← 0102;
  else temp ← memory (src);
    memory (tempa) ← temp;
    wait for completion;
      AC.cc ← 0102;
end if;
```

synmovl:

```
tempa ← dst and FFFFFFF816; # force alignment
temp ← memory (src);
memory (tempa) ← temp;
wait for completion;
  AC.cc ← 0102;
```

synmovq:

```
tempa ← dst and FFFFFFF016; # force alignment
temp ← memory (src);
if tempa = FF00001016
  then AC.cc ← 0102;
    use temp as a received iac message;
  else memory (tempa) ← temp;
    wait for completion;
      AC.cc ← 0102;
end if;
```

Faults: STANDARD

synmov, synmovl, synmovq

Example:

```
lda ff000010_16, g7
# g7 ← ff000010_16
synmovq g7, g8
# g8 ← IAC message from address ff000010_16
# AC.cc = 010_2
```

Opcode:

synmov	600	REG
synmovl	601	REG
synmovq	602	REG

See Also: **synld**

tanr, tanrl

Mnemonics:

tanr	Tangent Real
tanrl	Tangent Long Real

Format:

tanr*	*src*,	*dst*
	freg/flit	freg

Description: Calculates the tangent of *src* and stores the result in *dst*. The *src* value is an angle given in radians. The resulting *dst* value is in the range of -∞ to +∞, exclusive; a result of -∞ or +∞ will result in a floating invalid-operation exception being signaled.

For the **tanrl** instruction, if the *src* or *dst* operand references a global or local register, this register is the first (lowest numbered) of two successive registers. Also, this register must be even numbered (e.g., g0, g2, g4).

The following table shows the results obtained when taking the tangent of various classes of numbers, assuming that neither overflow nor underflow occurs.

Src	Dst
-∞	*
-F	-F to +F
-0	-0
+0	+0
+F	-F to +F
+∞	*
NaN	NaN

Notes:

F	Means finite real number
*	Indicates floating invalid-operation exception

If the source operand is a finite value, the result will be finite, unless the *src* operand is in a floating-point register and the *dst* operand is not.

NOTE

A value of -π/2 is generated as the result of executing *atanr 0, g0, fp0*, where the value of g0 is - 1. A resulting value of infinity is expected, when *tan fp0, fp1* is subsequently executed. The resulting value instead is a large positive value because of the initial rounding error incurred when calculating the value of -π/2.

tanr, tanrl

In the trigonometric instructions, the 80960SA/SB uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. The section in Chapter 10 titled "Pi" gives this π value, along with some suggestions for representing this value in a program.

Action:

dst ← tangent (*src*);

Faults:

STANDARD	Refer to the discussion of faults at the beginning of this chapter.
Floating Reserved Encoding	The *src* operand is an unnormalized (including denormalized) value and the normalizing-mode bit in the arithmetic controls is clear.

The following floating-point exceptions can be raised. Whether or not an exception results in a fault being raised depends on the state of its associated mask bit in the arithmetic controls.

Floating Overflow	Result is too large for destination format.
Floating Underflow	Result is too small for destination format.
Floating Invalid Operation	The *src* operand is ∞. The *src* operand is an SNaN value.
Floating Inexact	Result cannot be represented exactly in destination format.

Example:

```
tanrl g4, fp0    # tangent of value in g4,g5 is
                 # stored in fp0
```

Opcode:

tanr	68E	REG
tanrl	69E	REG

See Also: **cosr, sinr**

TEST

Mnemonic:

teste	Test For Equal
testne	Test For Not Equal
testl	Test For Less
testle	Test For Less or Equal
testg	Test For Greater
testge	Test For Greater or Equal
testo	Test For Ordered
testno	Test For Unordered

Format:

test*	*dst*
	reg

Description: Stores a true (1) in *dst* if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, the instruction stores a false (0) in *dst*.

The following table shows the condition-code mask for each instruction:

Instruction	Mask	Condition
testno	000	Unordered
testg	001	Greater
teste	010	Equal
testge	011	Greater or equal
testl	100	Less
testne	101	Not equal
testle	110	Less or equal
testo	111	Ordered

For the **testno** instruction (Unordered), a true is stored if the condition code is 000_2; otherwise a false is stored.

TEST

Action: For All Instructions Except **testno**:

if (mask **and** AC.cc) $\neq 000_2$
then *dst* ← 1; # *dst* set for true
else *dst* ← 0; # *dst* set for false
end if;

testno:

if AC.cc = 000_2
then *dst* ← 1; # *dst* set for true
else *dst* ← 0; # *dst* set for false
end if;

Faults: STANDARD

Example:
```
# assume AC.cc = 100₂
testl g9  # g9 ← 00000001₁₆
```

Opcode:

teste	22	COBR
testne	25	COBR
testl	24	COBR
testle	26	COBR
testg	21	COBR
testge	23	COBR
testo	27	COBR
testno	20	COBR

See Also: **cmpi, cmpdeci, cmpinci**

xnor, xor

Mnemonic:

xnor	Exclusive Nor
xor	Exclusive Or

Format:

xnor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg
xor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description: Performs a bitwise XNOR (**xnor** instruction) or XOR (**xor** instruction) operation on the *src2* and *src1* values and stores the result in *dst*.

Action:

xnor: *dst* ← **not** (*src2* **or** *src1*) **or** (*src2* **and** *src1*);

xor: *dst* ← (*src2* **or** *src1*) **and** **not** (*src2* **and** *src1*);

Faults: STANDARD

Example:

```
xnor r3, r9, r12    # r12 ← r9 XNOR r3
xor g1, g7, g4      # g4 ← g7 XOR g1)
```

Opcode:

xnor	589	REG
xor	586	REG

See Also: **and, andnot, nand, nor, not, notand, notor, or, ornot**