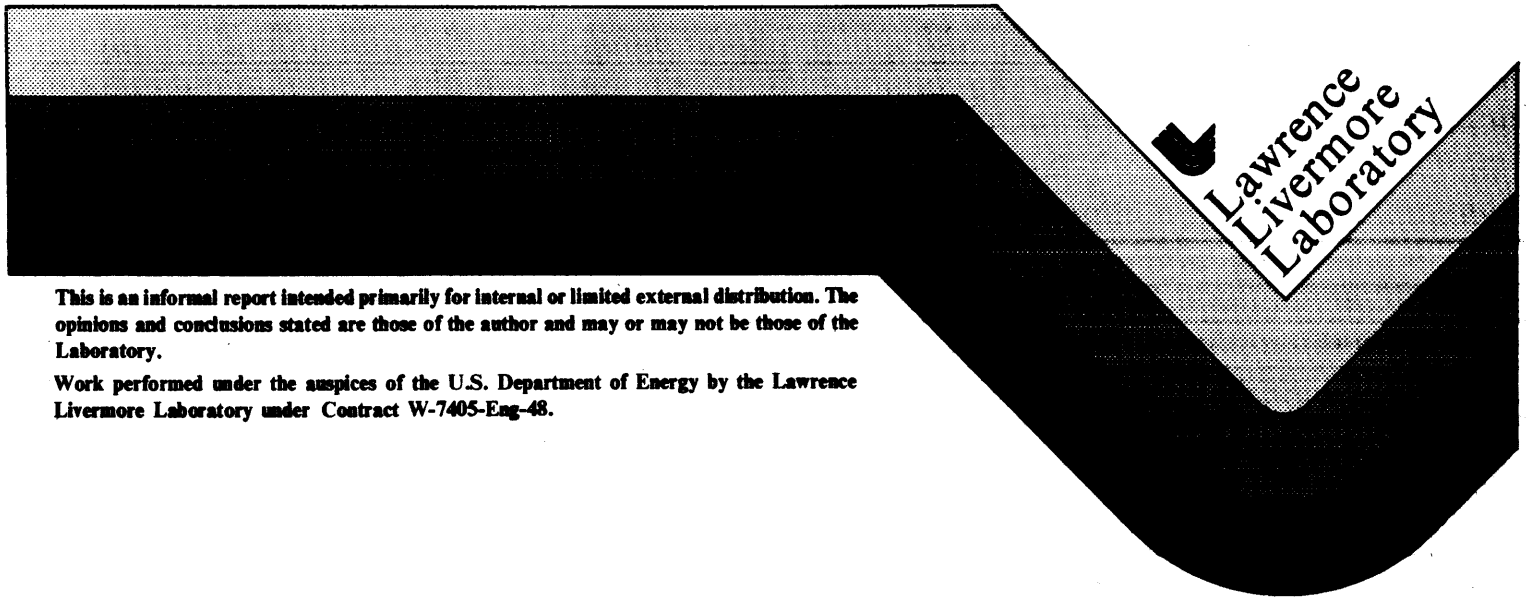
UCID- 19782

S-1 Uniprocessor Architecture

April 21, 1983



1 Introduction

The S-1 Mark IIA uniprocessor is the second generation of a pipelined vector and scalar processing computer with a virtual address space of 2^{29} thirty-six bit words, addressable in quarterwords, and a physical address space of 2^{32} singlewords. This manual describes its native mode instruction set and an assembler for that instruction set.

While a Mark IIA uniprocessor can operate alone or as part of a multiple-instruction-stream multiple-data-stream (MIMD) multiprocessor, this manual deals only with single processor operation. It also avoids implementation-dependent details like instruction timing and numerical values corresponding to opcode mnemonics.

Section 1 presents an overview of the architecture. Section 2, which assumes knowledge of the material in Section 1, divides the native mode instructions into groups, preceding each group with architectural details pertaining to that group. Section 3 describes the FASM assembler, but one can understand the assembly language examples in the previous sections without having read this description.

1.1 Notation

The remainder of the manual uses the following conventions for the sake of conciseness (the reader may want to skim these now and read them carefully only after encountering them in the text):

Radices Throughout the text, numbers appear in radix 10 unless otherwise noted. In the assembly language examples, numbers appear in radix 8 unless they include decimal points, which indicate they are in radix 10.

a . . b stands for the integers or elements from a through b inclusive.

{a,b,c,d} represents some one of a, b, c, or d.

M[x] represents the contents of memory at quarterword address x. Context should make clear whether this is a quarterword, halfword, singleword, or doubleword.

R[x] represents the contents of the registers at location x. Again, context should make clear whether this is a quarterword, halfword, singleword, or doubleword.

R0 . . R31 refer to the 32 singlewords in the register space (see Section 1.2.3).

X.Y denotes a field (that is, a series of consecutive bits) named "Y" within a memory location or register named "X".

X<n:m> denotes a field within X beginning at bit *n* and ending at bit *m*. **X<n>** represents the *n*th bit of X. We number the most significant ("leftmost") bit of a singleword "0" and the least significant bit "35". Sometimes, when we talk about an individual field within a word, we will number the bits starting at the leftmost bit within the field itself.

OP1, OP2, S1, S2, DEST
represent the *result* of evaluating the operand field of an instruction--that is, the register, memory location, or constant specified by the operand field rather than the operand field itself. Thus, for example, OD2 refers to the second operand field within an XOP instruction while OP2 refers to the register, memory location, or constant specified via that field.

SIGNED(X) means that X is a two's complement integer.

UNSIGNED(X) means that X is an unsigned integer, where all bits (including the most significant) contribute to the magnitude.

ZERO_EXTEND(X)
says to extend the precision of X by attaching zeroes to the left of it.

SIGN_EXTEND(X)
says to extend the precision of X by attaching copies of the sign bit of X to the high-order end of X.

LOW_ORDER(X), HIGH_ORDER(X)
designate the least-significant and most-significant portion of X, respectively. When context does not make clear how much of X to include, we will state the precision explicitly.

Address The term "address" means a virtual address unless stated otherwise.

↑ vs ** In the pseudo-pascal code which illustrates some of our instructions, we use "**" to indicate exponentiation, since the symbol "↑" is means something else in the FASM assembler.

This is an example of an example.

In addition, the assembly language examples use two constructs which may not immediately be clear.

First, we use "<>" instead of "()" brackets to parenthesize expressions, indicating the precedence of operators.

Second, when the operand of an instruction consists of one or more values separated by "?" marks and enclosed in square brackets, the assembler places those values in consecutive singlewords in memory and uses as the instruction operand the address of the first of those singlewords. Thus, the following examples have essentially the same effect:

```
DSPACE
F:      128
        256
        512
        1028
ISPACE
        PUSHADR SP,F
```

and:

```
ISPACE
        PUSHADR SP,[128 ? 256 ? 512 ? 1028]
```

These data literals are discussed in section 3.3.4. Notice that square brackets are also used in the syntax for indexing in the architectural addressing modes explained in section 1.6.5:

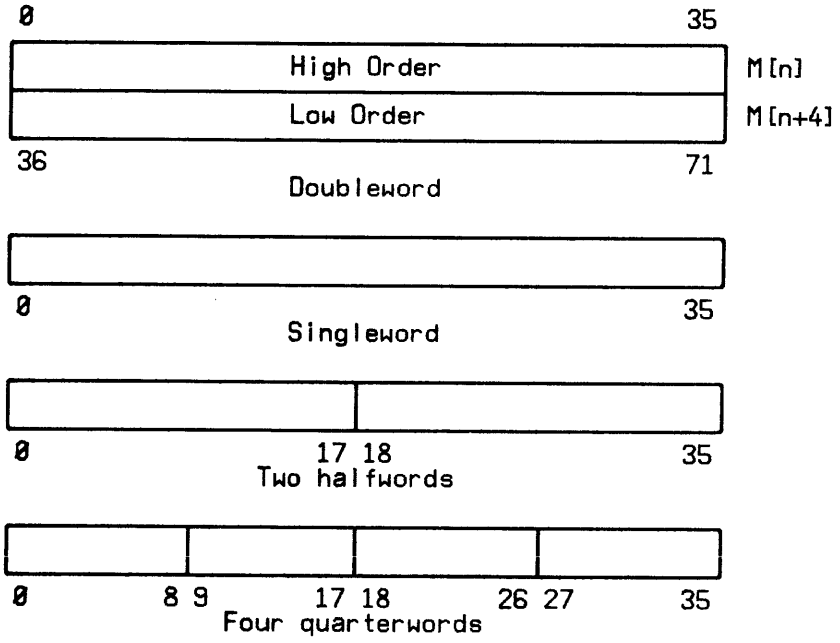
```
MOV.S.S RTA,(R7)FBRGD[RTB]
```

1.2 Words, Memory, and Registers

1.2.1 Words

The fundamental "word" in the S-1 native mode architecture is called a *singleword*, and is 36 bits long. Bits within a singleword are numbered from 0 upward, beginning at the most significant bit.

Many instructions access data in any of four different precisions--quarterword (QW), halfword (HW), singleword (SW), or doubleword (DW)--with equal ease.



Which precision a particular instruction deals with is either implicit in the instruction--the DJMPZ instruction, for example, always compares singlewords--or indicated by tacking a *modifier* onto the instruction name. For example, the notation "ADD.{Q,H,S,D}" means that

```
ADD.Q
```

adds quarterwords while

```
ADD.D
```

adds doublewords.

Unless otherwise specified, instructions address memory in terms of quarterwords regardless of the precision they deal with. For example, the first singleword in memory lies at address 0, the second

lies at address 4, the third lies at address 8, and so on. Quarterwords within a halfword, singleword, or doubleword have increasing addresses from left to right. Thus if a quarterword and a singleword have the same address, then the quarterword is the high order (most significant, or leftmost) quarterword of the singleword. Similarly, the more significant singleword in a doubleword has the lower address.

Halfwords and singlewords must be *aligned*: the address of a halfword must be a multiple of 2 or an ALIGNMENT_ERROR hard trap will occur. Similarly, the address of a singleword must always be a multiple of 4.

Any two consecutive singlewords can constitute a doubleword (though some implementations of the architecture may access a doubleword more efficiently if it is aligned on true doubleword boundaries, so that its address is a multiple of 8).

From now on, we use the term "word" interchangeably with "singleword" and refer to "anyword" when any of the four precisions is acceptable.

1.2.2 Memory

The processor has a physical address space of 2^{32} singlewords (quarterword addressable). At any time there are four (possibly) different virtual address spaces, one for each level of protection, called *rings*.

We use the term ADDRESS(X) to mean the virtual address of X and PHYSICAL_ADDRESS(X) to mean its physical address.

More precisely, ADDRESS(X) is a singleword in the form of a *pointer*, as described in Section 1.8.1: a five-bit *tag* field, one of whose purposes is to specify a ring, followed by a 31-bit address field which can address any quarterword in an entire 2^{31} quarterword space. Thus, ADDRESS(X) specifies both a tag and a quarterword address.

The architecture permits one to regard a virtual address space as a set of segments instead of a single vector of quarterwords, and thus an address may specify three coordinates: a ring, a segment and a quarterword address within that segment. The 31-bit address field specifies both the segment and the address within the segment.

The rings are numbered 0 . . 3, with ring 0 the topmost in the hierarchy. A ring can be protected against improper access on the part of a ring which lies below it in the hierarchy. In addition, the processor establishes a level dividing the rings. Those above the level are *privileged* while those below the level are not. Another term for unprivileged execution is *user mode*. Certain instructions are called "privileged" because attempting to execute them in user mode causes a

PRIVILEGE_VIOLATION hard trap (Section 1.9.6).

1.2.3 General Purpose Registers

An unprivileged process can access a single *register file*, a set of general purpose registers equivalent to 32 singlewords of memory. As with memory, instructions can access quarterword, halfword, singleword and doubleword entities within the registers, and they always address the registers in terms of quarterwords. The alignment rules that apply to memory also apply to the registers.

The architecture actually provides sixteen different register files numbered 0 through 15. When in privileged mode, the processor can access various register files and can choose which file is to be used by a particular unprivileged process.

Placing a "%" in front of an address tells the assembler to access the register space instead of memory. For example, an instruction which refers to "%4" will access the fifth quarterword in the register space (if it is dealing in quarterwords) or the third halfword (if it is dealing in halfwords), and so on. The registers act as a circular list, so %0 follows %127. Thus, for example, the eight quarterwords from %124 through %3 can constitute one doubleword.

Because one most often manipulates the registers as singlewords, the remainder of this manual will use the notation "R0" to represent the singleword at register address %0, "R1" to represent the singleword at register address %4, and so on up to "R31". Within the assembler, one can easily define the symbols "R0" through "R31" to have this meaning.

Certain register addresses have advantages over the rest while others have restrictions.

Indexing: Registers R0, R1, and R2 cannot be used as base registers for the "pseudoregister" addressing mode, which is explained further in Section 1.6.3.

Program counter: Register R3 has a dual identity. When an instruction uses R3 as the base for an address calculation (see Section 1.6.3), it accesses the program counter instead of R3 itself. When an instruction uses R3 in any other way, it accesses the true R3. There is no connection between the value in R3 and the value of the program counter; one particular usage of R3 within the addressing modes is simply defined to give the program counter instead.

Vectors: Certain vector instructions use R0, R1, and R2 to point to operands. When we use these registers in this fashion, we call them SR0, SR1, and SR2 respectively. Register R3 is also used to specify the lengths of vectors, and is then called SIZEREG.

RTA and RTB: Registers R4 and R6 are in a sense "easier" to access than the rest, and are named RTA and RTB respectively. For example, a three-operand instruction cannot in general access three different registers--but it can do so if the destination register is either RTA or RTB (Section 1.5.2).

When an instruction accesses RTA as a doubleword, it obtains both R4 and R5; we often refer to R5 as "RTA1". Similarly, we often refer to R7 as "RTB1".

Stack frame and closure pointers: One of the subroutine calling mechanisms provided by the architecture maintains stack frames by using register R29 as a closure pointer (called CP) and R30 as a frame pointer (called FP)(Section 2.11).

Stack pointer: Traps, interrupts, and subroutine calling instructions all use an upward-growing stack in memory to store return addresses and other context information. ("Upward-growing" means that pushing an item increases the address of the top of the stack.) R31 serves as the stack pointer for this particular stack, and is called SP. SP points to the first free location on the stack. (The instruction set makes it easy to use other registers or even memory locations as stack pointers to implement additional stacks for other purposes, as described in Section 2.10. But when we talk about "the stack" rather than "a stack", we mean the stack whose pointer is register SP.)

The table below summarizes the uses of the registers.

Register	Special characteristics
R0 . . R2	Cannot be base for pseudoregister mode; used as SR0, SR1, SR2 by vector instructions
R3	When used as base gives program counter instead; also used to specify vector length
R4, R5	RTA area
R6, R7	RTB area
R8 . . R27	None
R29, R30	Closure and frame pointers, CP and FP
R31	Subroutine stack pointer, SP

1.3 Program Counter

The program counter (PC) is an internal processor register (not part of any general purpose register file) containing a pointer to the instruction in memory that is currently being executed. Because instructions consist of singlewords aligned on singleword boundaries, the contents of the PC must always be a multiple of four. When an instruction contains multiple words, the PC continues to point to the first of them throughout the execution of that instruction.

Some operations refer to PC_NEXT_INSTR, which is the value the program counter will have for the following instruction in memory. A subroutine call, for example, places PC_NEXT_INSTR on the stack as its return address.

One can consider the PC to have a tag specifying the ring number used to fetch instructions. This ring is called the *ring of execution.* Any attempt to alter the contents of PC--a jump, call, or return instruction, for example--is subject to the validation checking described in Section 1.8.2.

1.4 Processor and User Status Registers

PROCESSOR_STATUS, the processor status, is an internal register (not part of any general purpose register file) which contains a number of fields affecting the behavior of the processor as a whole. Instructions which access this register are privileged. The following table and paragraphs describe briefly the purpose of each field; details generally appear elsewhere in this document.

Bits	Purpose
0 . . 1	EMULATION
2	VMM
3 . . 4	PRIVILEGED
5 . . 6	RING_ALARM
7 . . 10	REGISTER_FILE
11 . . 15	PRIORITY
16	TRACE_ENB
17	TRACE_PEND
18	CALL_TRACE_ENB
19	CALL_TRACE_PEND
20	UNMAPPED_MODE
21 . . 22	FLOW_TABLE
23 . . 31	Reserved
32 . . 35	FLAGS

EMULATION Determines which instruction set the processor currently executes. EMULATION=0 gives the native mode described in this document.

VMM Enables virtual machine mode, in which attempting to execute any privileged instruction and certain user mode instructions causes a trap.

PRIVILEGED Any ring whose number is less than or equal to PRIVILEGED is privileged.

RING_ALARM When the processor fetches an instruction, if the PC specifies a ring whose number is greater than RING_ALARM, the RING_ALARM_TRAP hard trap occurs. This permits deferral of an event until a critical inner ring operation completes.

REGISTER_FILE
Determines which of the sixteen register files is currently available to unprivileged processes. See Section 2.14.

PRIORITY Determines what priority an interrupt must have in order to interrupt the processor. See Section 1.10.

TRACE_ENB If this bit is on at the beginning of an instruction, TRACE_PEND is set at the end of the instruction--in other words, setting this bit enables trace traps for subsequent instructions, and the trap effectively occurs after each of those

instructions. Clearing this bit permits one final trap after the instruction which does the clearing. See Section 1.11.

TRACE_PEND If this bit is on at the beginning of an instruction, the processor traps before executing the instruction. Ordinarily, instead of manipulating TRACE_PEND directly, one manipulates TRACE_ENB and allows it to manage TRACE_PEND.

CALL_TRACE_ENB

Analogous to TRACE_ENB, this bit enables a separate trap for tracing instructions which call subroutines and return from them. Section 1.11 details the behavior of the trap and Section 2.11 enumerates the instructions to which it applies.

CALL_TRACE_PEND

Analogous to TRACE_PEND, this bit applies only to instructions that call a subroutine or return from one.

UNMAPPED_MODE

Causes the processor to bypass the usual virtual-to-physical mapping scheme and instead to use 31-bit addresses to access the first 2^{31} quarterwords of physical memory. The processor ignores tags and does not check segment bounds. This mode is useful for starting up a system or for simple diagnostics which run without a general purpose operating system.

FLOW_TABLE Enables one of four flow tables, each of which can contain 256 entries. While enabled, a flow table acts as a FIFO list of the PC values for the 256 most recently fetched instructions.

Reserved The effect of attempting to set these bits is undefined.

FLAGS This field is available for use by software.

USER_STATUS, the user status, is an internal register (not part of any general purpose register file) containing fields which affect the processor's behavior for a particular user or process. Instructions which access this register can execute in user mode.

The following table shows the position of the fields within register USER_STATUS.

Bits	Purpose
0	CARRY
1 . . 2	FLT_OVFL_MODE
3 . . 4	FLT_UNFL_MODE
5 . . 6	FLT_NAN_MODE
7	INT_OVFL_MODE
8	INT_Z_DIV_MODE
9 . . 13	FLT_RND_MODE
14	FLT_OVFL
15	FLT_UNFL
16	FLT_NAN
17	INT_OVFL
18	INT_Z_DIV
19 . . 23	INT_RND_MODE
24	UINT_OVFL_MODE
25	UINT_OVFL
26 . . 31	Reserved
32 . . 35	FLAGS

The fields which deal with integer arithmetic (CARRY, INT_OVFL, UINT_OVFL, INT_Z_DIV, INT_OVFL_MODE, UINT_OVFL_MODE, INT_Z_DIV_MODE, and INT_RND_MODE are described in Section 2.1 and the fields which deal with floating point arithmetic (FLT_OVFL, FLT_UNFL, FLT_NAN, FLT_OVFL_MODE, FLT_UNFL_MODE, FLT_NAN_MODE, and FLT_RND_MODE) are described in Section 2.2.

The effect of attempting to set the reserved bits is undefined.

The FLAGS field provides software-definable bits whose purpose is not specified by the architecture.

1.5 Instruction Formats

The heart of every instruction is a singleword which specifies one opcode and up to three operands.

Opcode: An opcode tells the processor what operation to perform--an ADD, an AND, a MOV, or whatever. In addition, the architecture uses the 12-bit opcode field of an instruction word to encode *modifiers* which are represented by a dot followed by one of several possible choices. For example, the ADD instruction comes in four different flavors: ADD.Q deals with quarterwords, ADD.H with halfwords, ADD.S with singlewords, and ADD.D with doublewords. In this manual, "ADD.{Q,H,S,D}" denotes a choice of these four flavors. Similarly, the SHFA instruction actually uses two different opcodes to incorporate its modifier: SHFA.LF for a left shift and SHFA.RT for a right shift.

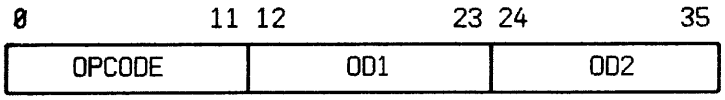
If an instruction takes more than one modifier, the order of the modifiers is significant. If one modifier refers to the first operand and the other to the second, the modifier for the first operand comes first. For example, MOV.S.Q converts a quarterword to a singleword whereas MOV.Q.S converts a singleword to a quarterword.

The mapping of the "virtual" opcodes shown in this manual onto actual, numerical opcode values is implementation dependent. In particular, if two virtual opcodes have the same effect--or can be made to have the same effect by swapping the order of their operands--an implementation may choose to map them to a single actual opcode.

Operands: Most instructions specify operands by means of an *operand descriptor* (OD), a 12-bit field that can indicate a constant, a register, a memory location anywhere within the 2^{31} quarterword address space, or indexed addressing using some combination of constants, registers, and memory.

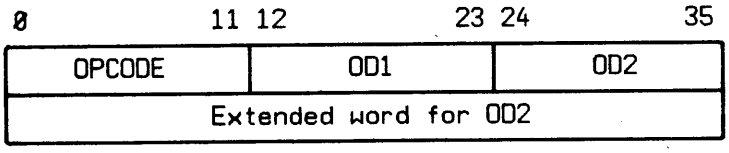
Sometimes the OD itself suffices to encode the operand--a small constant or a register, for example. Such an operand is called a *short operand* or SO. Obviously, more elaborate operands require more than twelve bits, so frequently an operand descriptor will tell the processor to use a word following the instruction as an *extended word* (EW). Such an operand is called a *long operand* or LO. Note that "long" and "short" refer to the length of the addressing mode, not to the length--quarterword, halfword, and so on--of the operand itself.

Thus, a two-operand instruction with operand descriptors OD1 and OD2 could require a singleword in memory if each descriptor specifies a short operand (that is, the 12-bit field can completely describe the operand):



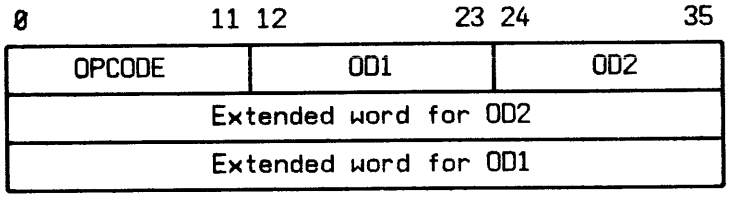
Both operands fit inside ODs

or would require two consecutive singlewords in memory if, for example, the second of the operands is an LO and thus calls for extended addressing:



OD2 calls for extended word

or would require three consecutive singlewords in memory if both operands called for extended addressing:



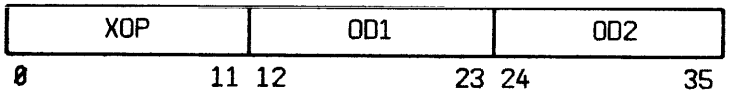
Both operands call for extended words

Note that when both extended words are present, the one used with OD2 occurs first.

The processor logically evaluates all operands, including extended addressing if necessary, before executing the instruction and before updating the program counter. The order of operand evaluation is undefined.

The preceding examples all showed the most common format for the initial singleword of an instruction: an opcode and two operand descriptors. In all, however, there are five different formats, called XOP, TOP, HOP, SOP, and JOP. We will first explain the formats and then explain how an operand descriptor and extended word combine to encode an operand.

1.5.1 Two-address Format (XOP)



XOP Format

Typically a two-address instruction evaluates operand descriptors OD1 and OD2 to obtain operands OP1 and OP2 respectively, then reads from OP2, performs the specified operation, and writes into OP1.

Unless otherwise noted, if an XOP instruction uses only one operand then it uses OD1 and requires that the field used to encode OD2 be zero, or an OPERAND_NOT_REQUIRED hard trap will occur. If an XOP instruction uses no operands, the fields for both OD1 and OD2 must be zero, or that trap will occur. The FASM assembler automatically handles these cases. If an instruction uses neither operand, FASM sets both fields to zero. If you write only one operand and the instruction needs only one, FASM sets the unused OD field to zero. If the instruction needs two, FASM uses the same operand twice.

For example, FASM emits the same code for the following two instructions because the INC instruction requires two operands:

```
INC COUNT,COUNT      ; COUNT := COUNT + 1
INC COUNT            ; COUNT := COUNT + 1
```

The following example uses INC more flexibly:

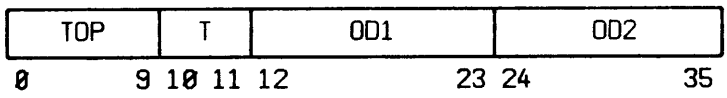
```
INC COSTPLUS1,COST   ; COSTPLUS1 := COST + 1
```

The RUS instruction requires only one operand, so providing two would be an error:

```
RUS RTA              ; RTA := USER_STATUS
```

When an XOP instruction stores results in both operands, it stores OP2 first (see the example under the EXCH instruction in Section 2.6).

1.5.2 Three-address (TOP) Format



TOP Format

A typical three-address instruction operates on data from two operands and deposits the result in the third.

Because not enough bits are available to provide three operand descriptor fields, a TOP contains only two, OD1 and OD2. A two-bit field called "T" describes how the instruction uses those two operands and what it uses for the third.

If we use "TOP" to represent the operation performed by any particular TOP instruction, then we can use the following equation to represent the effect of the instruction:

```
DEST := S1 TOP S2
```

The "T" field determines which operands to use for DEST, S1, and S2 according to the following table:

T	DEST	S1	S2
0	OP1	OP1	OP2
1	OP1	RTA	OP2
2	RTA	OP1	OP2
3	RTB	OP1	OP2

FASM automatically sets "T". The following are all legal combinations:

```
ADD X,X,Y          ; X := X + Y (T field = 0)
ADD X,RTA,Y        ; X := RTA + Y (T field = 1)
ADD RTA,X,Y        ; RTA := X + Y (T field = 2)
ADD RTB,X,Y        ; RTB := X + Y (T field = 3)
```

If X, Y, Z, and RTA are all distinct, the following examples are illegal and FASM will give error messages:

```
ADD X,Y,Z          ; Illegal
ADD X,Y,Y          ; Illegal
ADD X,Y,X          ; Illegal
```

This special ability to specify RTA and RTB via the T field does not preclude specifying RTA or RTB as ordinary operands inside the descriptors OD1 and OD2, however. The following examples are therefore perfectly correct:

```
ADD RTB,X,RTA          ; RTB := X + RTA (T field = 3
                       ;     and OP2 = RTA)
ADD X,RTA,RTB          ; X := RTA + RTB (T field = 1
                       ;     and OP2 = RTB)
```

Reverse form: The T field of a TOP instruction provides asymmetric features: it can specify that the first operand (S1) is either RTA or identical with the destination (DEST), but it cannot do the same for the second operand (S2). The asymmetry would handicap non-commutative instructions like those for subtraction and division, so such instructions generally have *reverse forms* that swap S1 and S2. The name of a reverse form instruction is that of the normal form with a "V" appended.

If we use "TOP" to represent the operation performed by any particular reverse form, then we can use the following equation to represent the effect of the instruction:

```
DEST := S2 TOP S1
```

The instruction SUBV, for example, is the reverse form of the TOP instruction SUB:

```
SUB X,RTA,Y      ; X := RTA - Y
SUBV X,RTA,Y     ; X := Y - RTA
SUB X,Y          ; X := X - Y
SUBV X,Y         ; X := Y - X
```

Without SUBV, subtracting RTA from Y and storing the result in X would be impossible in a single instruction:

```
SUB  X,Y,RTA           ; Illegal
```

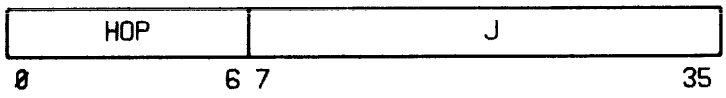
A reverse form swaps the precisions of the operands as well as their order in the expression that describes the instruction. If, for example, the normal form of an instruction expects S1 to have twice the precision of S2, then the reverse form expects S2 to have twice the precision of S1. If the normal form uses a single operand from S2 and a pair from S1, the reverse form uses S1 and a pair from S2.

Short form: If only two operands appear, FASM will use the first one as both S1 and DEST. Thus the following pairs of instructions are equivalent:

```
ADD X,X,Y          ; X := X + Y
ADD X,Y            ; X := X + Y

SUBV X,X,Y         ; X := Y - X
SUBV X,Y           ; X := Y - X
```

When an ordinary TOP instruction stores more than two results, it stores S2 before S1 and S1 before DEST. When a reverse form TOP instruction stores more than two results, it stores S1 before S2 and S2 before DEST. Any unused OD field must be set to zero; the assembler does this automatically.

1.5.3 HOP Format



HOP Format

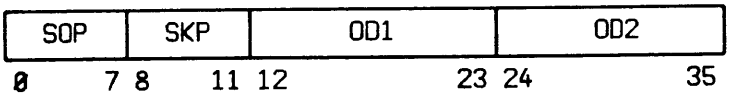
A single instruction, SJMP, uses this format to jump to a location relative to the current program counter. The processor uses the "J" field as an unsigned displacement, expressed in singlewords. The address calculation "wraps around" if it exceeds the maximum address:

```
GOTO (PC+4*SIGNED(J)) MOD (2**31)
```

Thus the instruction can actually jump to any singleword in a virtual address space. To jump backward, the instruction merely uses a J field large enough to cause the address calculation to wrap around.

In practice, the assembly language programmer simply provides a label for the branch destination and lets the assembler calculate the J field.

1.5.4 Skip (SOP) Format



SOP Format

Generally a SOP instruction compares two operands and, depending on the result, branches relative to the current program counter. The term "skip" has a broader meaning here than in many architectures; the destination of the branch can be any location within -8 . . 7 singlewords of the program counter (which is, as defined in Section 1.3, considered to point to the first word of the skip instruction itself).

The SOP field tells the processor what condition to test for, the SKP field tells it where to branch, and operand descriptors OD1 and OD2 can specify two operands to be compared. The following statement describes a typical SOP instruction:

```
IF OP1 SOP OP2 THEN GOTO PC+4*SIGNED(SKP)
```

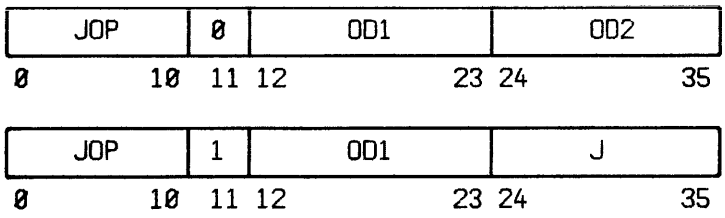
To use a SOP instruction in FASM, simply provide a label for the skip destination. The assembler will automatically subtract the current location to compute the offset.

```
        ; If X is greater than Y, swap them
        SKP.LEQ X,Y,NOSWAP
        EXCH X,Y
NOSWAP: ...
```

Omitting the label is the same as skipping the next instruction. Thus, the following example has the same effect as the previous one:

```
        ; If X is greater than Y, swap them
        SKP.LEQ X,Y
        EXCH X,Y
NOSWAP: ...
```

1.5.5 Jump (JOP) Format



JOP Format

Jump instructions generally perform an operation on a piece of data and then branch. The JOP field is the opcode and OD1 is an operand descriptor that specifies the operand OP1.

When bit 11 (called the "PR" bit) is 1, the processor performs a relative jump. The "J" field is a signed offset that permits branching to any singleword location within -2048 . . 2047 singlewords of the current location. (By definition, the program counter points to the JOP instruction itself while the processor interprets the instruction.) The processor adds "J" to the PC to obtain a jump destination, or *JUMPDEST*.

When bit 11 is 0, the processor performs an absolute jump. It evaluates operand descriptor OD2 and, if necessary, an extended word to obtain the JUMPDEST, allowing direct, indirect, or indexed addressing--but sometimes costing an extra word of memory to do so. If OD2 specifies a register or constant, an ILLEGAL_REGISTER_OPERAND or ILLEGAL_CONSTANT_OPERAND hard trap occurs.

The FASM assembler decides automatically whether to use an absolute or relative JOP; simply provide it with a branch destination label:

```
JMPZ.GTR.S X,AWAY          ; IF X .GT. 0 THEN GOTO AWAY
```

Specifying a more complicated operand for the JUMPDEST--the contents of a register, for example--forces FASM to emit an absolute jump:

```
JMPZ.GTR.S X,(R16)0     ; IF X .GT. 0 THEN GOTO (the
                        ;  address found in R16)
```

Omitting the jump destination label in FASM has the same effect as jumping past the following instruction. Thus the next two examples are equivalent:

```
        JMPZ.EQL.S A,F
        EXCH.S A,B
F:      ...

        JMPZ.EQL.S A
        EXCH.S A,B
F:      ...
```

1.5.6 Vector Instructions

Vector instructions generally use the same format as XOP instructions. OD1 and OD2 are operand descriptors which may specify either scalars or vectors, depending on the particular instruction. Negative or zero vector lengths cause a vector instruction to process no data. No attempt to calculate the operand addresses is made. Therefore, no spurious page faults, access violations, or segment bounds violations will occur.

A vector is simply a series of consecutive scalars which must lie in memory, not in the registers. When a series of consecutive scalars may lie in either memory or the registers, we call it a *block*. Unless noted otherwise, vector instructions obtain from register R3--also called SIZEREG--the length of the vectors they operate on. SIZEREG expresses lengths in terms of elements, not quarterwords. Thus, for example, SIZEREG=100 indicates the vectors are 200 quarterwords long if the current instruction operates on halfwords or 800 quarterwords if the current instruction operates on doublewords.

When an instruction uses OD1 to specify a vector, it evaluates OD1 to obtain OP1, regards OP1 as the first element of the vector (*not* a pointer to the vector) and assumes the remaining elements follow OP1 in memory. The same is true of OD2. Thus, when we refer to "the vector x" we mean the vector whose first element is x.

When a vector instruction needs more than two operands, it uses registers R0, R1, and R2--also called SR0, SR1, and SR2 respectively--as *pointers* to the additional vectors in memory.

Unless otherwise noted, the result of a vector operation is undefined if a source operand and a destination operand overlap (unless they coincide).

Many vector instructions permit the user to choose by means of a {SR,OP1} modifier whether to put the result back into OP1 or into an arbitrary vector pointed to by the appropriate SR register.

At the beginning of the description of each vector instruction, to the right of the name of the instruction, a symbolic equation describes its operands. For example, the following means that a vector operand and a scalar operand produce a vector result:

V:=VS

while the following means that two vector operands produce two scalar results:

SS:=VV

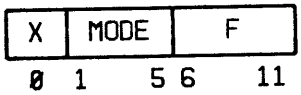
1.6 Operand Descriptors

This section explains the capabilities of the operand descriptors referred to in the preceding instruction formats. Note that some operands are specified through operand descriptors and others are not. For example, the relative jump version of the JOP format uses an operand descriptor called OD1 to specify operand OP1 while it uses a field called J--which does *not* obey the rules for an operand descriptor--to specify the jump destination. The fields which are not operand descriptors have already been described under each of the instruction formats.

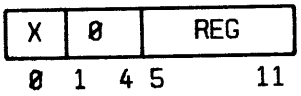
1.6.1 Subfields of an Operand Descriptor

As mentioned earlier, operands which are specified by operand descriptors belong to two classes. If an operand fits inside an OD, we call it a *short operand* (SO); if it requires an extended word (EW), we call it a *long operand* (LO). Note that "long" and "short" refer to the complexity of the addressing mode, *not* to the precision of the operand: a short operand may, for example, be a quarterword, halfword, singleword, or doubleword.

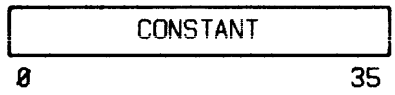
A 12-bit operand descriptor field is generally partitioned into three subfields called OD.X, OD.MODE, and OD.F:



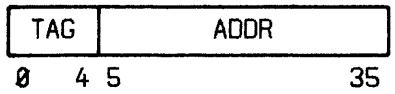
The sole exception occurs when the four high-order bits of OD.MODE are all zeros, in which case the low-order bit of OD.MODE joins the OD.F field to form a field called OD.REG:



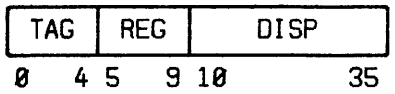
When X=1 the OD requires an EW, and that EW can be partitioned in three ways, depending on the value encoded in the OD:



Constant EW



Simple-base EW



Complex-base EW

1.6.2 Constant Operands

Any operand descriptor can specify a constant, though particular instructions may prohibit them. For example, operand descriptor OP1 of a MOV.S.S instruction can encode a constant, but the instruction will encounter an ILLEGAL_CONSTANT_OPERAND hard trap because storing into a constant is illegal. Similarly, it is illegal for an instruction to attempt to obtain ADDRESS(x) if x is a constant.

The assembler interprets an expression preceded by "#" as a constant. The assembler will encode the constant as compactly as possible. Constants in the range -32 . . 31 will fit in SO format while the LO format accommodates up to 36-bit signed constants:

```
ADD.S A,#-5              ; -5 would become an SO constant
ADD.S A,#TABLESIZE       ; Illustrates the use of expressions
ADD.S A,#<TABLESIZE-1>   ;   as constants
```

Bracketing the number or expression with "[]" symbols forces FASM to use the LO format even if the constant is small enough to fit in the SO format. This makes it possible to use a symbolic debugger to patch the constant to a larger value later on, and guarantees that the size of the code emitted will not vary with the size of the constant:

```
ADD.S A,#[-5]
ADD.S A,#[TABLESIZE-1]
ADD.S A,#[106125103113]
```

(Note that because a "#" precedes them, the square brackets here do *not* denote assembly literals.)

The precision of an instruction is inherent in the opcode, not the operands, so a constant in either SO or LO format is ordinarily converted from a 36-bit entity to the desired precision at execution time, either reducing precision by discarding high-order bits or increasing precision by extending the sign bit.

If an instruction calls for doubleword precision, however, it is possible to specify different conversions. Putting "!0 ?" in front of the constant but within the brackets sets the high-order half of the doubleword to zero and the low-order half to the constant. Putting "? !0" after the constant but within the brackets sets the high-order half of the doubleword to the constant and the low-order half to zero:

```
MOV.D.D A,#[-1]          ; A := 777777777777777777777 octal
MOV.D.D A,#[!0 ? -1]     ; A := 000000000000777777777777
MOV.D.D A,#[-1 ? !0]     ; A := 777777777777000000000000
```

Note that these conversions are not available unless the instruction calls for doubleword precision. For any other precision, it is possible to encode these conversions in the OD format, but the processor will convert the constant in the ordinary manner--by discarding high-order bits or extending the sign bit.

Indexed constants: This operand format specifies a 36-bit signed constant and a singleword-aligned register. It adds the value in the register to the constant, converts the sum to the precision of the instruction by either discarding bits or extending the sign, and uses the result as a constant operand. Note that the addition ignores integer overflow, and that specifying R3 accesses register R3 rather than the program counter:

```
; one instruction...
ADD.S RTA,RTA,#[4](RTB) ; RTA := RTA + RTB + 4
; versus two...
ADD.S RTA,RTA,RTB       ; RTA := RTA + RTB
ADD.S RTA,#4            ; RTA := RTA + 4
;(x+1)*(x-1) or x^2-1:
MULT.S RTA,#[1](RTA),#[-1](RTA) ; RTA := (RTA + 1) * (RTA - 1)
                                ; or RTA := RTA^2 - 1
```

NOTATION

Symbol	Meaning	
sc	-32 . . 31	short constant
lc	-2^{35} . . $2^{35}-1$	long constant
ar	0 step 4 until 124	aligned register

SHORT OPERAND CONSTANTS

(If the constant is too big, the assembler automatically uses the LO form)

FASM notation	Evaluation	OD Format
		0 1 5 6 11
#sc	sc	0 | 2 | sc

LONG OPERAND CONSTANTS

FASM notation	Evaluation	OD Format	EW Format
		0 1 5 6 11	0 35
#[lc]	SIGNED(lc)	1 | 2 | 1	lc
#[!0 ? lc]	ZERO_EXTEND(lc)	1 | 2 | 2	lc
#[lc ? !0]	lc*2**36	1 | 2 | 3	lc
#[lc](%ar)	SignExtend(lc) +R[ar]	1 | 2 | 32+ar/4	lc

Figure 1-1
Constant Operand Formats

1.6.3 Short Operand Variables

The SO format can denote two kinds of variable: a register or a location in memory accessed as a *pseudoregister.*

Registers: The SO format can access any quarterword address within the register space, subject to the usual rules for alignment of entities larger than a quarterword. Specifying register R3 accesses register R3, not the program counter.

```
ADD.S RTA,%8.              ; Add contents of singleword at reg %8
                           ;  (third singleword in registers) to RTA
ADD.Q RTA,%11.             ; Add contents of quarterword at register
                           ;  %11 to RTA (due to misalignment, ADD.H,
                           ;  ADD.S, or ADD.D would be illegal)
ADD.H RTA,%<COUNTER+2>     ; Illustrates the use of expressions
```

Pseudoregisters: In itself, pseudoregister addressing provides a compact means of specifying a memory location. The name *pseudoregister* arises because the more elaborate addressing modes described in Section 1.6.5 incorporate this pseudoregister mode to give a memory location the same capabilities as a register.

Pseudoregister addressing uses a singleword-aligned register to point to an address in memory and provides a quarterword offset to select an anyword in the vicinity of the address pointed to. The offset must lie in the range -128 . . 124 and be divisible by 4.

The register serves as a *base* pointer--an important concept throughout all the memory addressing modes. Its upper 5 bits serve as the tag which, among other things, specifies the desired ring. Its lower 31 bits contain an address. The concept of a base pointer is additionally important because it determines the meaning of register R3. When one uses R3 as a base pointer, one obtains the program counter instead of R3 itself. And last of all, the base pointer determines the segment in which an operand lies (Section 1.7.2). The first term of every memory addressing calculation is considered a base pointer, and a singleword fetched from memory to serve as an indirect address is considered a base pointer also.

As for pseudoregister addressing in particular, note that while the register containing the base pointer must be singleword-aligned, the alignment of the entity it points to is governed only by the precision of the instruction. Thus, for a halfword instruction, the register must point to an aligned halfword. Similarly, the actual operand obtained by adding the offset to the pointer must be aligned properly for the precision of the instruction.

As an example of pseudoregister addressing, let VSP be a register used to point to an upward-growing stack of parameters and variables in memory. Pseudoregister mode makes it easy to access variables relative to the top of the stack:

```
ADD.S (VSP)-4,#7           ; Add 7 to top singleword on stack
                           ; (for upward-growing stack, pointer
                           ; indicates next free location)
EXCH.S (VSP)-8.,(VSP)-4    ; Swap top two singlewords of stack
SKP.EQL.S (VSP)-20.,(VSP)-4 ; Compare top singleword with fifth
                           ; singleword
```

As another example, suppose that register R7 contains a tagged pointer to a Pascal record structure. Pseudoregister addressing can access components of that record:

```
MOV.S.S RTA,(R7)4          ; move second word of record to RTA
MULT.S RTB,(R7),(R7)8      ; RTB gets product of first and
                           ; third words
```

As Section 1.6.4 explains, one of the LO addressing modes has the same syntax as the pseudoregister mode, and permits a larger offset. The assembler automatically uses the LO format if the desired offset is too large.

NOTATION

Symbol	Meaning	
r	0 . . 127	register
pr	12 step 4 until 124	pseudoregister base
sao	-128 step 4 until 124	short aligned offset
R[x]	Contents of register location x	
M[x]	Contents of memory location x	
B[x]	Evaluate x as a base pointer;	

(don't use x as an address from which to fetch something);
if x=R3 use PC instead

SHORT OPERAND VARIABLES

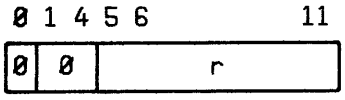
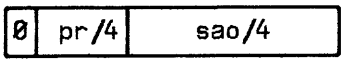
FASM notation	Evaluation	OD Format
%r	R[r]	
(%pr)sao	M[B[R[pr]]+sao]	

Figure 1-2
Short Operand Formats

1.6.4 Long Operand Variables

Long operand variable formats use the extended word alone to encode various memory address computations.

Fixed-base: This mode uses a 31-bit field to specify a base address in memory. (The tag is implicitly that of the ring in which the instruction is executing; no field is provided to encode a tag explicitly.)

One may either use the entity at that address as the operand, or treat it as a new base pointer for indirect addressing:

```
MOV.S.S RTA,AVAR        ; Copy the singleword at
                        ; memory location AVAR to RTA
MOVP.P.A APTR,AVAR      ; Make APTR point to AVAR
MOV.S.S RTA,APTR@       ; Address AVAR indirectly through APTR
```

Variable-base: This mode uses a singleword-aligned register as a base pointer (that is, it has a tag in its upper 5 bits and an address in its lower 31 bits.) The computation adds a 26-bit signed offset to the address field of the pointer. One may use the resulting address either to fetch the operand or to fetch a new base pointer which in turn specifies the operand:

```
MOV.H.H RTA,(R7)1000.   ; Copy to RTA the halfword
                        ; which is 1000 quarterwords above the
                        ; quarterword pointed to by R7
MOV.Q.Q RTA,(R7)1       ; The assembler automatically uses the
                        ; LO format here because the SO
                        ; pseudoregister format requires the
                        ; offset to be a multiple of 4
MOVP.P.A (R7)1000.,AVAR ; Make (R7)1000. point to AVAR
MOV.S.S RTA,(R7)1000.@  ; Address AVAR indirectly through
                        ; the pointer at (R7)1000.
```

NOTATION

Symbol	Meaning	
ar	0 step 4 until 124	aligned register
la	$0 . . 2^{31}-1$	long address
sd	$-2^{25} . . 2^{25}-1$	short displacement
M[x]	Contents of memory location x	
R[x]	Contents of register location x	
B[x]	Evaluate x as a base pointer;	

(don't use x as an address from which to fetch something);
if x=R3 use PC instead

LONG OPERAND VARIABLES

FASM notation	Evaluation	OD Format	EW Format
		0 1 5 6 11	0 4 5 9 10 35
la	M[B[la]]	1 | 2 | 0	4..7 | la
la@	M[B[M[B[la]]]]	1 | 2 | 0	2,3,8..11 | la
(%ar)sd	M[B[R[ar]]+sd]	1 | 2 | 0	20..23 | ar/4 | sd
(%ar)sd@	M[B[M[B[R[ar]]+sd]]]	1 | 2 | 0	24..27, 30..31 | ar/4 | sd

Figure 1-3
Long Operand Variable Formats

1.6.5 Combined Long and Short Operand Variables

These addressing modes use both the short operand and the extended word to encode memory address calculations. In each case, one may choose to use a pseudoregister in place of one of the registers involved in the address calculation, thus nesting one calculation inside another.

In their most general form, these calculations sum three terms: a *base pointer*, an *offset*, and an *index* (though not every term need always appear) after shifting the index:

(BASE POINTER)OFFSET[INDEX]↑SHIFT

Unless otherwise mentioned, the base pointer is a singleword pointer (that is, it has a tag in the upper five bits and an address in the lower 31 bits.) The offset and index values are added to the 31-bit address using modulo 2^{31} arithmetic. This means that the sum cannot overflow into the tag field, and that when the offset is 31 bits long, one may regard it either as a signed value or as an unsigned value that "wraps around" the virtual address space.

The shift moves the index 0, 1, 2, or 3 bits leftward (multiplying it by 1, 2, 4, or 8) so that the index can effectively represent a number of quarterwords, halfwords, singlewords, or doublewords. (For example, because the architecture always addresses memory in terms of quarterwords, singlewords are 4 addresses apart rather than 1 address apart. To step through a table of singlewords, one must either increment the index by 4 each time--which is usually inconvenient--or use the built-in shift feature to multiply by 4.) If omitted, the shift defaults to 0.

The modes which provide indexing permit indirect addressing either before the indexing operation:

(BASE POINTER)OFFSET@[INDEX]↑SHIFT

or afterward:

(BASE POINTER)OFFSET[INDEX]↑SHIFT@

In the first case, the calculation adds the offset to the base pointer, obtains a new base pointer from the resulting address, and adds the index to the new base pointer to find the operand. In the second, the calculation adds both the offset and the index to the base pointer, obtains a new base pointer from the resulting address, and uses that base pointer to find the operand. When indirection follows the indexing operation, the shift must be either 0 or 2.

Based: This mode uses a base pointer (which can be either a singleword-aligned register or a singleword memory location specified by means of pseudoregister addressing) and a 31-bit offset.

```
MOVP.P.A (R7)-4,F          ; Make the singleword at (R7)-4
                                 ; point to F
MOV.S.S RTA,((R7)-4)100.   ; Move to RTA the singleword
                           ; which lies 100 quarterwords above
                           ; F
MOVP.P.A ((R7)-4)100.,AVAR ; Make F+100 point to AVAR
MOV.S.S RTA,((R7)-4)100.@  ; Use that pointer to address AVAR
                           ; indirectly
```

Based-indexed: This mode uses a base pointer (which can be either a singleword-aligned register or a singleword memory location specified by means of pseudoregister addressing), a 26-bit signed offset, and a singleword-aligned register for indexing. Indirect addressing is possible either before or after the indexing operation:

```
MOV.Q.Q RTA,(R7)100.[RTB]        ; Move to RTA the quarterword
                                 ; obtained by using R7 as a base
                                 ; pointer to memory, adding a
                                 ; 100-quarterword offset to the
                                 ; pointer, and offsetting further
                                 ; by the value found in RTB
MOV.Q.Q RTA,((R7)-4)100.[RTB]    ; Similar to the previous example,
                                 ; but use as the base pointer the
                                 ; singleword specified by
                                 ; pseudoregister (R7)-4
MOV.H.H RTA,((R7)-4)100.[RTB]↑1  ; Similar to the previous example,
                                 ; but multiply the index register by
                                 ; 2 since we are addressing halfwords
MOV.Q.Q RTA,(R7)100.@[RTB]       ; In any of the previous three
                                 ; examples, one may use the offset to
                                 ; find a new base pointer, indirect
                                 ; address through it, and then use
                                 ; the index register as a further
                                 ; offset
MOV.H.H RTA,(R7)100.[RTB]↑2@     ; Alternatively, one may choose to
                                 ; use the singleword obtained by the
                                 ; indexing operation as an indirect
                                 ; addressing pointer.
```

Fixed-based-indexed: This mode provides a 31-bit base address and an index (which can be either a singleword-aligned register or a singleword in memory specified by a pseudoregister). Because the 31-bit base address provides no means of encoding a tag, the tag is implicitly that of the ring in which the instruction is executing. One may choose indirection either immediately before or immediately after the indexing operation.

```
MOV.Q.Q RTA,BPTR[RTB]            ; Move to RTA the quarterword found
                                 ; by using BPTR to point to memory and the
                                 ; value stored in RTB as an offset from
                                 ; that location
MOV.D.D RTA,BPTR[RTB]↑3          ; Like the previous example, but multiply
                                 ; the index by 8 since we are dealing with
                                 ; doublewords
MOV.Q.Q RTA,BPTR[(R7)-4]         ; Shows the use of pseudoregister
                                 ; (R7)-4 as the index
MOV.Q.Q RTA,BPTR@[RTB]           ; Use the singleword at BPTR as an indirect
                                 ; address pointer and index from the location
                                 ; to which it points
MOV.Q.Q RTA,BPTR[RTB]@           ; Similar to the first example, but use the
                                 ; singleword located by the indexing oper-
                                 ; ation as an indirect address pointer
```

Register-based-indexed: This mode provides a singleword-aligned register as the base pointer, a 26-bit signed offset, and an index (which may be either a singleword-aligned register or a singleword in memory specified by a pseudoregister). One may choose indirection either preceding or following the indexing operation.

```
MOV.Q.Q RTA,(R7)100.[(R8)-4]     ; Move to RTA the quarterword found
                                 ; by using R7 to point to memory, adding
                                 ; an offset of 100. to the address given
                                 ; in R7, and then adding as an additional
                                 ; offset the value stored in the singleword
                                 ; specified by pseudoregister (R8)-4
MOV.S.S RTA,(R7)100.[(R8)-4]↑2   ; Like the initial example, but multiply
                                 ; the index by 4 because we are
                                 ; dealing with singlewords
MOV.Q.Q RTA,(R7)100.@[(R8)-4]    ; Indirection preceding indexing
MOV.Q.Q RTA,(R7)100.[(R8)-4]↑2@  ; Indirection following indexing
```

To illustrate the usefulness of a combined short and long operand variable addressing mode, consider the following fragment of a Pascal procedure:

```
VAR
   I: INTEGER; TABLE: ARRAY [5..9] OF INTEGER;
BEGIN
FOR I := 5 TO 9 DO
   TABLE[I] := I;
```

To construct the operand for TABLE[I], assume first that FP is a register pointing to the beginning of the stack frame for the procedure, and that the TABL'th byte in the stack frame points to the memory location which would be the 0th element of TABLE if TABLE had a 0th element. The following operand would specify that pointer:

(FP)TABL

and the following operand would specify that fictional 0th element:

(FP)TABL@

If VI is the byte offset from the beginning of the stack frame to variable I, then the following indexes to find the Ith element of TABLE. Note the use of a shift to access singlewords properly:

(FP)TABL@[(FP)VI]↑2

The entire loop might look like this:

```
      MOV.S.S  (FP)VI,#5
LOOP: MOV.S.S  (FP)TABL@[(FP)VI]↑2,(FP)VI
      ISKP.LEQ (FP)VI,#9.,LOOP
```

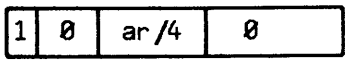
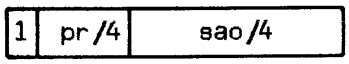
(We assume VI and TABL are not too large to fit within this operand format, and that the value of I is not used again following the loop.)

NOTATION

Symbol	Meaning	Symbol	Meaning
ar	0 step 4 until 124	M[x]	Contents of memory location x
pr	12 step 4 until 124	R[x]	Contents of register location x
sao	-128 step 4 until 124	B[x]	Evaluate x as a base pointer
la	$0 . . 2^{31}-1$		(if x=R3 use PC instead)
ld	$-2^{30} . . 2^{30}-1$	sh	0 . . 3
sd	$-2^{25} . . 2^{25}-1$	ssh	0 or 2

COMBINED LONG AND SHORT OPERAND VARIABLES

Substitute either of these short operands . . .

FASM Notation	Evaluation	OD Format
%ar	R[ar]	
(%pr)sao	M[B[R[pr]]+sao]	

. . . for "SO" in the following:

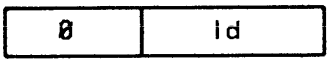
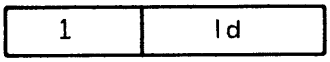
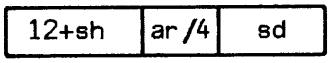
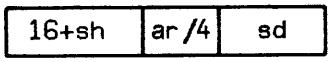
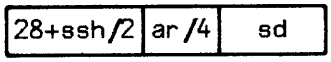
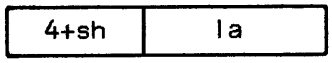
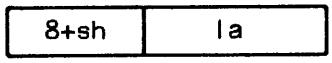
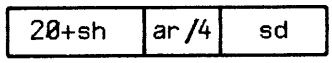
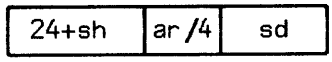
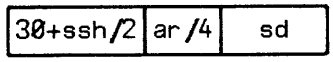
FASM notation	Evaluation	EW Format
(SO)ld	M[B[SO]+ld]	
(SO)lde	M[B[M[B[SO]+ld]]]	
(SO)sd[%ar]↑sh	M[B[SO]+sd+R[ar]*2**sh]	
(SO)sde[%ar]↑sh	M[B[M[B[SO]+sd]]+R[ar]*2**sh]	
(SO)sd[%ar]↑sshe	M[B[M[B[SO]+sd+R[ar]*2**ssh]]]	
la[SO]↑sshe	M[B[M[B[la]+SO*2**ssh]]]	
la[SO]↑sh	M[B[la]+SO*2**sh]	
lae[SO]↑sh	M[B[M[B[la]]]+SO*2**sh]	
(%ar)sd[SO]↑sh	M[B[R[ar]]+sd+SO*2**sh]	
(%ar)sde[SO]↑sh	M[B[M[B[R[ar]]+sd]]+SO*2**sh]	
(%ar)sd[SO]↑sshe	M[B[M[B[R[ar]]+sd+SO*2**ssh]]]	

Figure 1-4
Combined Long and Short Operand Variable Formats

1.6.6 NEXT Versus FIRST/SECOND

Certain instructions are defined to deal not just with an operand but also with elements that follow that operand in memory.

Vector instructions are an important example. If the first element of a vector is *x*, we use the terminology "NEXT(*x*)" to describe the element which follows *x* in memory and has the same precision as *x*. Thus, if the first element of a vector is F, then the second element is NEXT(F), the third element is NEXT(NEXT(F)), and so on. The elements are handled independently, so no special alignment rules govern them.

Certain other instructions deal with pairs of elements: the operand and the single element following that operand. For example, the CMAG instruction computes the magnitude of a complex number, where OP2 is the real part of the complex number and the entity following OP2 is the imaginary part. In these cases, we use the terminology "FIRST(*x*)" and "SECOND(*x*)" to describe the operand *x* and its successor. If the precision of the instruction is quarterword or halfword, then the two elements must align together to form a single entity of twice that precision.

Operands described in terms of NEXT also differ from those described in terms of FIRST/SECOND with respect to constants.

When an operand described in terms of NEXT is a constant, the instruction replicates the constant to provide the required number of elements, each having the precision specified by the instruction. The VTRANS instruction, for example, copies one vector to another, so the following sets each element of vector A to 7:

```
VTRANS.S.S ARRAY,#7
```

When an operand *x* described in terms of FIRST/SECOND is a constant and the precision of the instruction is quarterword, halfword, or singleword, the instruction expands the constant to twice that precision, uses the high order half as FIRST(*x*), and uses the low order half as SECOND((*x*). (When expanding a singleword constant to a doubleword, it observes the special constant addressing modes for doing so.) For instance, the BNDSF.RTA.B instruction is an XOP which sets RTA true or false depending on whether OP2 lies within the bounds specified by FIRST(OP1) and SECOND(OP1), so the following example:

```
BNDSF.RTA.B.S #[!0 ? 7],A
```

will test to see whether A lies within the range 0 . . 7 and set RTA accordingly.

When an operand *x* described in terms of FIRST/SECOND is a constant and the precision of the

instruction is doubleword, the instruction replicates the constant to provide FIRST(x) and SECOND(x). Thus, for example,

```
BNDSF.RTA.B.D #[!0 ? 7],A
```

will test to see whether A lies between 7 and 7.

1.6.7 Forbidden Operand Formats

Certain combinations of bits in the OD and EW formats do not constitute legal addressing modes. The processor interprets these as invalid long operands, causing a RESERVED_ADDRESS_MODE hard trap:

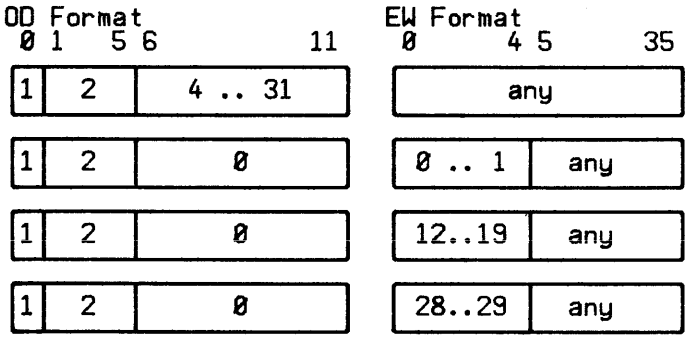


Figure 1-5
Forbidden Operand Formats

1.7 Virtual to Physical Address Translation

The address translation mechanism maps 31-bit virtual addresses onto 34-bit physical addresses, providing both segmentation and paging. It provides four different virtual address spaces, one per ring, which may overlap.

1.7.1 Paging

The paging mechanism permits a virtual address space to be mapped onto widely scattered pieces of physical memory, eliminating problems of memory fragmentation in a multiprogramming system. It facilitates demand paging by recording whether a page has been accessed or altered, and by trapping on any attempt to access a page that is absent from memory. And it permits one to restrict the right to read, write, or execute each individual page.

A page is 4096 quarterwords long. We thus say that the constant LOGPAGESIZE is 12. Because a single virtual address space may contain as many as 2^{19} pages, it is evident that the page mapping tables may themselves need to be paged.

In fact, the address translation mechanism has four different steps. Instead of a giant page table 2^{19} entries long, it uses many little page tables each 16 entries long, so not every page table need be in memory at once. Taken together, the 16 pages pointed to by one page table make up a *segmentito.*

A giant table called a *Descriptor Segment* contains a pointer to each of the (at most) 2^{15} page tables for each of the 4 virtual address spaces--or 2^{17} page tables in all. If the Descriptor Segment were wired permanently into memory, an address reference would require two translations: one to find the proper page table and another to find the proper page. But the Descriptor Segment itself is composed of pages grouped into segmentitos, so an address reference first requires two translations to find the appropriate point in the Descriptor Segment, and then two more translations to find the target address.

Figure 1-6 traces the entire process. A register called the *Descriptor Segment Pointer* (DSEGP) holds the 34-bit physical address of the first word of the Descriptor Segmentito Table. Because the Descriptor Segment points to (at most) four sets of 2^{15} segmentitos and each pointer requires 8 quarterwords, the Descriptor Segment never exceeds 2^{20} quarterwords. That translates into a maximum of 16 segmentitos, which means at most 16 entries (called *Segmentito Table Entries*, or STEs) in the Descriptor Segmentito Table. The 2-bit number of the ring being accessed together with the first 2 bits of the virtual address select one entry from the 16 in the Descriptor Segmentito Table. In turn, that entry points to the physical address of the first word of a Descriptor Page Table, which has an entry (called a *Page Table Entry*, or PTE) for each of the 16 pages comprising that segmentito. Bits 2 . . 5 of the virtual address select one entry from the 16 in that particular Descriptor Page Table, which points to one page of the Descriptor Segment itself.

The Descriptor Segment, of course, contains nothing but pointers to segmentitos that make up the 4 virtual address spaces. In fact, this page of pointers is identical in form to the Descriptor Segmentito Table, except that it has more entries and the entries point to pages inside one of the virtual address spaces instead of inside the Descriptor Segment. Thus, we have labeled it a "Target Segmentito Table." (Note, however, that the page shown is probably only one of many pages of segmentito pointers required to describe the entire ring, and that the Descriptor Segment is a continuous list of such pointers, not a separate table for each ring.) Bits 6 . . 14 select one STE from this table, which points to the physical address of the first word of a Target Page Table, which has an entry for each of the 16 pages comprising that segmentito.

Bits 15 . . 19 of the virtual address select one PTE from that page table, which points to the physical address of the first word of a page. Lastly, bits 19 . . 30 of the virtual address select a quarterword from that page.

Using less than the full mapping: One need not use the entire mapping structure provided. Any segmentito or page table entry may be null, either because the corresponding segmentito or page is absent from memory or because the virtual address space in question is smaller than the maximum allowable size.

Overlapping virtual address spaces: It is possible to make part or all of different virtual address spaces overlap, simply by making some of their STE or PTE entries point to the same physical memory. Some operating systems have customarily placed user and executive together in one address space, providing protection by restricting access to particular pages. To achieve such operation with this architecture, one may simply arrange the entries in the Descriptor Segmentito Table to point to the same set of Descriptor Page Tables for each ring, thus mapping all four rings onto the same physical memory and reducing the size of the mapping tables by roughly a factor of four.

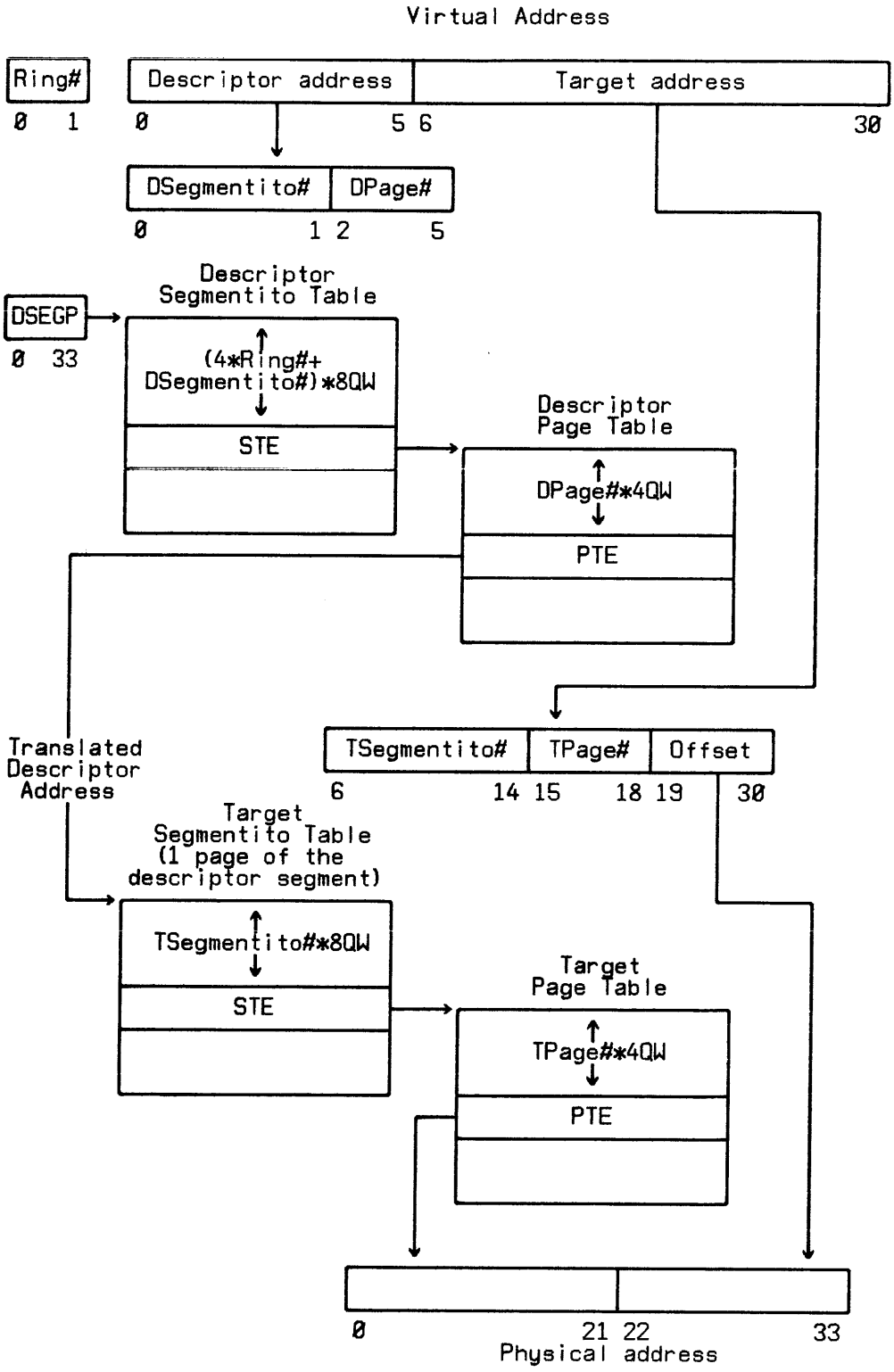


Figure 1-6
Virtual-to-physical address translation

1.7.2 Segmentation

One can view a virtual address space as a set of segments, so that the address for any particular entity consists of a pair of coordinates: the segment number and the offset from the beginning of the segment. If an index or offset causes an address calculation to exceed lower or upper segment bounds, an OUT_OF_BOUNDS hard trap occurs.

Segments can vary in size, consisting of one or more segmentitos, but a segment must obey three rules: the number of segmentitos in the segment must be a power of two, the segmentitos must be consecutive within the virtual address space (which means simply that the pointers to them must be consecutive in the descriptor segment) and the virtual address of the beginning of the segment must be an integer multiple of the size of the segment.

Those three rules make it easy to check segment bounds. Given any virtual address known to be within a segment, plus the size of the segment, the processor can determine whether a second, "suspect" address lies within the same segment merely by comparing the upper 19-x bits of the 31-bit addresses (where x is the base 2 logarithm of the number of pages in the segment, a number which we refer to as LOGSEGSIZE).

As a result, the processor need not maintain an explicit table of segment boundaries. Instead, the pointer to each segmentito merely contains a field giving the size of the segment containing that segmentito.

As an example, assume we know some address x lies within a particular segment, and we know the segment contains 8 (2^3) segmentitos. To see whether an address y lies in the same segment, first discard the 12 low order bits of x and y, which merely represent varying addresses within a page; because a segment must start and end on segmentito boundaries and thus page boundaries, we need merely check that the suspect address lies on a permissable page, without worrying about where within the page it lies. But then we can discard an additional 4 low order bits from each of x and y because they merely represent varying addresses within a segmentito; given that a segment must start and end on segmentito boundaries, we need merely check that the suspect address lies on a permissable segmentito, without worrying about where within that segmentito it lies. Finally, we can discard an additional 3 bits just because the size of the segment is 2^3 segmentitos. Those 3 bits must be zero for the first of the 8 segmentitos in order for the segment to start on an integer multiple of its size, and as a result they must equal 7 for the last of the 8 segmentitos. Since the 3 bits can have any value from 0 to 7 and still lie within the segment, we need not worry about them, either. The remaining bits should be identical for every legal address within the segment, so we compare the remaining bits of x and y. Only if they match did the two original addresses lie within the same segment.

Segment bounds checking: Every memory address calculation begins with a base pointer, establishing which segment is being addressed. The rule for bounds checking is simply that a memory access must lie within the same segment as the previous base pointer. Thus, the base pointer plays the role of address x in the previous example, and the actual operand being accessed serves as y.

When an address calculation involves indirection, the indirect pointer must lie within the same segment established by the base. But the pointer then establishes a new base, possibly in a different segment, and subsequent memory accesses must lie within the same segment as the new base.

Bounds checking occurs only on actual memory accesses, so it is permissable for an offset to reach outside the segment bounds provided a subsequent indexing operation brings the calculation back within bounds before the access occurs.

1.7.3 Segmentito and Page Table Entries

Segmentito table entries: Each STE is a doubleword (shown in Figure 1-7) with the following fields:

VALID If this bit is set, the page table for this segmentito is in memory and the processor uses the remainder of the doubleword as described. Otherwise, the segmentito is absent, the processor ignores the rest of the doubleword and software may use it as desired. Attempting to access an absent segmentito causes a SEGMENTITO_FAULT hard trap (or, if the segmentito is part of the descriptor segment, a DSEG_SEGMENTITO_FAULT hard trap).

PTA Singleword physical address of the corresponding page table.

WB Write bracket. Attempting to write into this segmentito from a ring (or, more formally, with a validation level) greater than WB causes an READ_WRITE_BRACKET_FAULT hard trap.

EB Execute bracket. Attempting to execute this segmentito from a ring (or, more formally, with a validation level) greater than EB causes an EXECUTE_BRACKET_FAULT hard trap. Note that a cross-ring call via the instruction CALLX and the gate mechanism (Section 1.9.5) is not considered an attempt to execute the called segmentito, and is thus exempt from EB restrictions.

RB Read bracket. Attempting to read this segmentito from a ring (or, more formally, with a validation level) greater than RB causes an READ_WRITE_BRACKET_FAULT hard trap.

ACCESS Specifies access modes as defined later in this section for all pages in this segmentito.

SIZE Specifies the size of the segment that contains this segmentito, expressed as a base 2 logarithm of the number of pages in the segment (for example, SIZE=8 indicates the segment contains 2^8 pages, which is 2^4 segmentitos). SIZE must not be less than 4 (2^4 pages, or 1 segmentito) or greater than 19 (2^{19} pages, or 2^{15} segmentitos, or an entire 2^{31} quarterword address space.)

FLAGS Reserved for use by software.

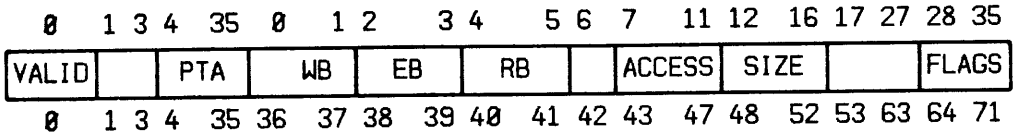


Figure 1-7
Segmentito table entry format

Page table entries: Each PTE is a singleword (shown in Figure 1-8) with the following format:

VALID If this bit is set, implying that this page is in memory, the processor uses the remainder of the singleword as described here. Otherwise, the page is absent and the software may use the remainder of the singleword as desired. Attempting to access an absent page causes a PAGE_FAULT hard trap.

USED If VALID=1, this bit indicates the page has been accessed. (More precisely, the processor sets this bit when it brings into the map cache (Section 2.13) the mapping information for this page.)

MODIFIED If VALID=1, this bit indicates the page has been modified. (More precisely, the processor sets this bit when it marks the corresponding map cache entry to show that the page has been written into.)

FLAGS Reserved for use by software.

ACCESS Specifies access modes for this page as defined later in this section.

PAGENO The 22 high order bits of the physical address of this page.

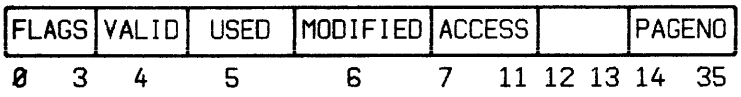


Figure 1-8
Page table entry format

Access modes: The access permitted for a particular page is the logical AND of the ACCESS fields in the STE and the PTE for that page. They permit an operating system to mark a page for read-only access, write-only access, execute-only access, or any combination of reading, writing, and execution. An instruction which is fetched from a memory location which does not have execute access will cause an INSTRUCTION_ACCESS_VIOLATION. An instruction which attempts to access a memory location in violation of read and write access bits will cause an DATA_ACCESS_VIOLATION hard trap. (Of course, the attempted access must pass the checking defined by the RB, EB, and WB fields in the STE, too.) Within each ACCESS field, the bits have the following meanings:

WRITE_PERMIT
Instructions may alter this segmentito/page.

EXECUTE_PERMIT
A process may execute instructions fetched from this segmentito/page

READ_PERMIT Instructions may read from this segmentito/page.

IO_PAGE I/O instructions may address this page, but ordinary instructions may not. Note that the WRITE_PERMIT and READ_PERMIT bits determine whether the I/O instructions can write or read this page.

SHARED This bit provides advice to the hardware which can improve multiprocessor performance, but which is not necessary for correctness. It has no effect unless the page is writable. For a writable page, reading a location from memory ordinarily causes the hardware to demand from the crossbar read/write access to that location. If the SHARED bit is 1, however, reading a location causes the hardware to demand only read access, though a subsequent attempt to write that location will then demand read/write access.

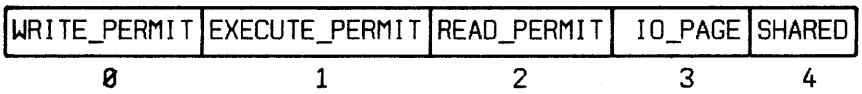


Figure 1-9
Bits in ACCESS field

1.8 Rings and Protection

The uniprocessor architecture provides three principal kinds of protection.

The first, specified in the PRIVILEGED field of the PROCESSOR_STATUS register as mentioned earlier, determines the rings from which privileged instructions may be fetched for execution.

The second, discussed in the preceding sections, applies to privileged and non-privileged instructions alike, and to all four rings: unless otherwise noted, the architecture provides segment bounds checking (which prevents a memory address calculation from erroneously exceeding the boundaries of a segment) and access mode checking (which controls the ability of any instruction to read, write, or execute a particular page).

A third kind of protection allows "downward" accesses (in which an instruction executing in a given ring reaches into a less protected ring to access an operand) but forbids "upward" accesses (in which an instruction reaches into a more protected ring). This involves a process called *validation*, which checks the TAG field of a pointer and alters it or, if necessary, invokes a BAD_ADDRESS_TAG or BAD_ADDRESS_TAG hard trap to protect more protected (lower-numbered) rings against forbidden accesses from less protected (higher-numbered) rings. There are two kinds of validation: *address validation* occurs when a pointer is used in addressing an operand or specifying a jump destination; and *pointer validation* occurs when a pointer is itself an operand (usually when the pointer is being moved from one place to another). The following sections discuss the pointer format, address validation, and pointer validation.

1.8.1 Pointer Format

As mentioned earlier, the pointers that serve as the base for most memory address calculations and all indirect references incorporate both a TAG field and an ADDRESS field (Figure 1-10). Pointer tags play an important role in dynamic linking, in memory accesses from one ring to another, and in calls from one ring to another.

Though the architecture also features self-relative pointers and byte pointers, the word "pointer" by itself in this manual will always mean a tagged pointer with the format shown in Figure 1-10.

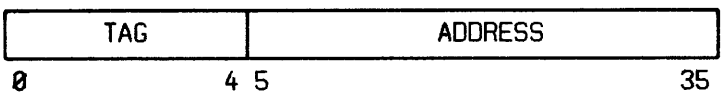


Figure 1-10
Pointer Format

Various values of the TAG field have the following meanings:

Tag	Meaning
0	Fault. Any attempt to use a pointer with a fault tag will cause either a BAD_POINTER_TAG or a BAD_ADDRESS_TAG depending on the mode of usage.
1	Gate. As explained in Section 1.9.5, the CALLX instruction can use a pointer with a gate tag to implement a procedure call from one ring to another. Any attempt to reference memory through a gate pointer will cause a BAD_ADDRESS_TAG hard trap. If a BASEPTR or VALIDP instruction attempts to operate on a gate pointer it will fail with a BAD_POINTER_TAG hard trap. The MOVP instruction may, however, move such a pointer.
2	NIL. If an instruction attempts to use this pointer to reference memory a BAD_ADDRESS_TAG hard trap will result. If the BASEPTR or VALIDP instruction attempts to operate on this pointer, a BAD_ADDRESS_TAG hard trap occurs. The MOVP instruction may, however, move this pointer. A language translator such as LISP, Pascal, or PL/I may use this pointer to implement the NIL or NULL construct.
3	Self-relative. The address field of the pointer is a signed, quarterword displacement giving the offset of the referenced location relative to the location containing the pointer. When such a pointer is used, it is treated as if it where a pointer to the addressed location. When stored in a non-self- relative pointer, the result will have a ring-tag for the validation level of the location holding the self-relative pointer. A self-relative pointer and the location that it addresses must be in the same segment, otherwise, an OUT_OF_BOUNDS hard trap occurs. Since it must be in memory, if a pointer with a self-relative tag is found in a register, an ILLEGAL_RELATIVE_POINTER trap occurs. Also, if a self-relative pointer is used as the base pointer in an addressing mode, a BAD_ADDRESS_TAG trap occurs.
4	Ring 0 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 0 address space. If such a reference (for any ring tagged pointer) would cause a validation error a BAD_ADDRESS_TAG hard trap will occur. If, during pointer validation, a MOVP instruction tries to alter any ring tag a BAD_POINTER_TAG hard trap will result. Both forms of ring validation are explained in section 1.8.2
5	Ring 1 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 1 address space.

6	Ring 2 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 2 address space.
7	Ring 3 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 3 address space.
8 . . 30	User tag. An instruction which references memory though this pointer will attempt to access the specified ADDRESS within the same ring from which it obtained the pointer (more precisely, it will access memory using as the initial validation level the validation level derived in fetching the pointer; see Section 1.8.2.) Because these 23 tags are equivalent architecturally, software may use them for its own purposes, such as encoding the data type of the entity being addressed.
31	Fault. This behaves exactly like a tag of zero. Because all but the largest-magnitude positive and negative integers will have either 0 or 31 in the tag field, assigning special meanings to tags of 0 and 31 increases the likelihood that the erroneous use of a random singleword as a pointer will be detected as an error.

1.8.2 Address Validation

The address validation that occurs during operand or jump destination evaluation applies to two classes of pointers: those with TAG values in the range 4 . . 7, which are called *ring pointers*; and those with TAG values in the range 8 . . 30, which are called *user pointers*. (One frequently refers to *ring tags* and *user tags* in a similar fashion.).

An instruction or pointer is "trusted" by the ring from which it is fetched, and by higher-numbered rings. Address validation enforces two rules. First, an instruction cannot access a ring unless the instruction and each pointer used in computing the address are trusted by that ring. Second, an instruction cannot access a location unless the instruction and each pointer used in computing the address of that location are trusted by the ring specified by the EB, WB, or RB field--whichever is appropriate--of the STE (Section 1.7.3) for the segmentito containing that location.

Because the architecture allows virtual address spaces to overlap, it is imprecise to say that an instruction, pointer, or operand "lies within a ring". The page containing the instruction, pointer, or operand may lie within multiple rings. For an instruction, we refer instead to the "ring of execution", meaning the ring specified by the PC in fetching the current instruction. For a pointer or operand, we refer to the *validation level*, an internal value derived by the addressing mechanism which specifies which ring number to use in accessing the desired entity.

Using those terms, here is the algorithm for address validation:

1. For each operand, the address calculation mechanism initializes the validation level to the number of the ring of execution.

2. Each time the calculation handles a pointer, it uses the validation level and the tag to derive a new validation level:

a. If the tag is a ring tag and the ring number is less than the validation level, a BAD_ADDRESS_TAG hard trap occurs.

b. If the tag is a ring tag and the ring number is greater than or equal to the validation level, the new validation level is the ring number.

c. If the tag is a user tag, the validation level is unchanged.

Note that the validation level can never decrease, because that would allow access to a more protected ring.

Of course, an attempt to access memory is also subject to checking specified by the ACCESS fields in the STE and PTE entries, and to that specified by the WB, EB, and RB fields in the STE entry: the validation level derived in computing the address must be less than or equal to that specified by the WB, EB, or RB field--whichever is appropriate.

To illustrate the rule that an instruction cannot use a pointer to access a ring which is more protected than the ring of execution, suppose the following instruction executes in ring 1:

MOV RTA,(R7)100.@

The initial validation level is therefore 1. The address calculation first uses R7 as its base pointer. If R7 contains a pointer with a ring 2 tag and an address F, then the calculation proceeds legally because 2>1, and the validation level increases to 2. Next the calculation fetches an indirect pointer from address F+100 within the virtual address space of ring 2. Suppose that pointer has a tag of 1 and an address of B. Because 1 is less than the current validation level, a hard trap occurs--even though the instruction itself is executing in ring 1 and could have accessed location B in ring 1 directly. In this fashion, the cross-ring access mechanism prevents a pointer which is only trusted to the level of ring 2 from exploiting the capabilities of a more trusted instruction executing in ring 1.

To illustrate the additional checking provided by the EB, WB, and RB fields in the STE entry, suppose that ring 1 and ring 2 are mapped to the same physical memory. If address F lies in a segmentito for which the WB field in the segmentito is 1 and the RB field is 2, then either of the following instructions can execute in ring 1:

MOV.S.S RTA,F
MOV.S.S F,RTA

(Recall from Section 1.6.4 that the tag for the operand "F" is implicitly that of the ring in which the instruction executes.) The first instruction can execute in ring 2 as well, because RB=2. But the second instruction will trap if it executes in ring 2, because WB=1. In this manner, one can give the executive read/write access to a segmentito while limiting the user to read-only access.

1.8.3 Pointer Validation

By itself, the address validation mechanism discussed in the previous section is not sufficient to protect lower-numbered rings against mischief from higher-numbered rings. The ring number used to fetch a pointer helps determine its validation level, so simply moving the pointer from a higher-numbered ring to a lower-numbered one could give it additional capabilities.

For example, a user executing in ring 3 might construct a pointer to data in ring 0 and then pass the pointer as the address of a parameter to an operating system routine executing in ring 0, thereby deceiving the operating system into accessing, on behalf of the user, data which is forbidden to the user.

Therefore, whenever one moves a ring pointer or user pointer, it undergoes a second kind of validation, called *pointer validation*, which alters its tag or, if necessary, traps to avoid giving the pointer additional privileges. This validation is built into an instruction called MOVP, which should be used in place of MOV whenever one moves a pointer. If a pointer is moved implicitly--if it is passed from one ring to another via a register, for example--the recipient must deliberately validate it using the VALIDP instruction.

Pointer validation involves two steps:

1. If the pointer is in a register, the initial validation level is the number of the ring of execution. If the pointer is in memory, set the initial validation level to equal the address validation level derived in fetching it from memory.

2. Use that validation level to derive a new tag:

    a. If the tag is a ring tag and the validation level is greater than the number of the ring specified by the tag, invoke the BAD_POINTER_TAG hard trap (because this pointer wants to access a more protected ring than the one from which it was obtained).

    b. If the tag is a ring tag and the validation level is less than or equal to the number of the ring specified by the tag, preserve the tag (because this pointer wants to access a less protected ring than the one from which it was obtained).

    c. If the tag is a user tag and the validation level equals the number of the ring

of execution, preserve the tag. (Because the pointer was obtained from the ring of execution, it cannot possibly be moving to a more protected ring. Moving it to a less protected ring is harmless; at worst, if the pointer is fetched from that ring and used for indirection, it will appear to point to a less protected entity than it did before.)

d. If the tag is a user tag and the validation level is greater than the number of the ring of execution, replace the tag with the ring tag corresponding to the validation level (the pointer may be moving to a more protected ring than the one from which it was obtained, so make the latter explicit).

To illustrate these rules, suppose a user routine called USER, executing in ring 3, has called an operating system routine called EXEC, executing in ring 0. USER has constructed a ring pointer called BAD, located in ring 3 but pointing to ring 0, and has passed in register R0 a pointer to BAD. (For the moment, we will assume the pointer in R0 is correct and trustworthy.) EXEC executes the following instruction to move BAD into a location called TRUSTED within ring 0:

```
MOVP.P.P TRUSTED, (R0)
```

The processor first calculates the address of BAD, using the address validation algorithm. The address validation level starts at 0, the ring of execution, and becomes 3, the ring number specified by the pointer in R0.

Once the instruction has addressed BAD, the pointer validation algorithm starts with 3, the validation level derived during the address calculation, and examines the tag field of BAD itself, which is a ring tag for ring 0. Because 0 is less than 3, the MOVP instruction traps.

Suppose instead that BAD is a user pointer. This time, when EXEC attempts to move it to TRUSTED, the processor first calculates the validation level as 3, and then moves BAD to TRUSTED. Because the validation level is greater than the ring of execution, the processor replaces the user tag with the ring tag for ring 3. No error (and thus no trap) occurs.

But suppose instead that the pointer passed in register R0 is itself bad--that is, USER has constructed it to point to data in ring 0. The validation level of a pointer located in register 0 and pointing to ring 0 is in fact 0, so no trap will occur when EXEC addresses memory through R0. Even if EXEC is suspicious and attempts to move the pointer from R0 to TRUSTED before using it, the validation level still matches the ring tag, so no trap occurs:

```
MOVP.P.P TRUSTED,R0
MOVP.P.P TRUSTED,TRUSTED@
```

That illustrates the importance of using the VALIDP instruction to validate a pointer generated by an untrustworthy process and passed to a trustworthy routine through a register. Provided a called routine applies VALIDP properly to every pointer passed in a register, it is protected completely because the validation mechanisms will prevent violations by any other pointers inside structures

passed to it.

1.8.4 Validation on JOP instructions

The validation mechanisms discussed in the preceding sections apply in a special fashion to absolute-addressing JOP instructions (that is, jump, call, and return instructions in which the PR bit is not 1). The PC is itself a pointer whose ring tag defines the current ring of execution. So such instructions effectively perform a MOVP to store the new value into it. In general, if such an instruction attempts to move a NIL or gate tag pointer to the PC, a BAD_ADDRESS_TAG trap occurs; when such an instruction moves a pointer which has a user tag, it converts the tag to that of the current ring of execution.

These pointer validation mechanisms prevent an instruction executing in a higher-numbered ring from calling a routine located in a lower-numbered ring. Because such calls are needed to permit user code to obtain operating system services, the architecture provides two mechanisms that circumvent the validation scheme in a controlled fashion: the TRPEXE instruction, discussed in section 1.9.4, and the CALLX instruction with gates, discussed in section 1.9.5.

1.9 Traps, Interrupts, and Gates

Traps and interrupts signal the processor to change its context temporarily and deal with an exceptional situation. Traps usually result from errors, while interrupts are usually invoked by external devices in need of I/O service. The mechanism described here is also used by the TRPEXE and TRPSLF instructions and gate calls to change to a more privileged context such as the operating system kernel. For ease of exposition the word trap will often be used in a generic sense to refer to all conditions which are handled with this mechanism. When used in a specific context it will be used in conjunction with a modifier such as *hard trap* or *TRPEXE trap*.

For each type of trap which may occur, a series of singlewords in memory called a *trap vector* provides information on handling the trap. The processor obtains new state information from the vector, pushes its previous state onto a stack, and branches to a trap handler address specified by the vector.

(Conventions vary on whether "vector" applies to the group of singlewords pertaining to a particular trap, or to the group of groups pertaining to all traps. We will always use "vector" to refer to the series of singlewords for a particular trap, and will use "set of vectors" to refer to the consecutive vectors for the different traps of the same type.)

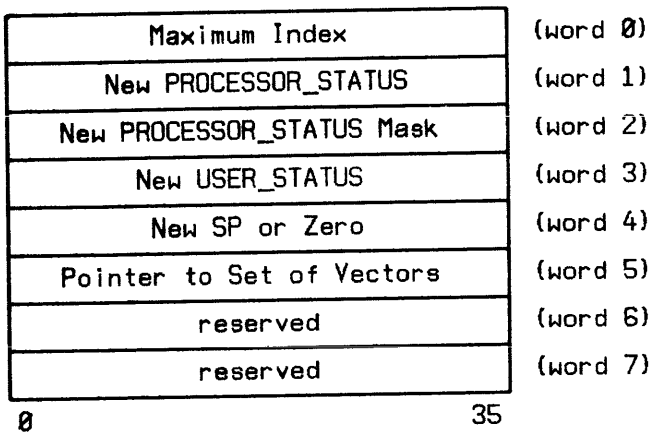
There are several classes of traps:

- Traps which can be handled by a process at its own level of privilege. These include *soft traps* caused by errors such as divide by zero.
- Traps caused by the TRPSLF instruction which can also be handled at the same level of privilege.
- Traps caused by the TRPEXE instruction, which are in effect calls to the executive.
- Traps caused by a CALLX instruction which uses gate-tagged pointers to make cross-ring calls (Section 1.9.5).
- Traps which must be handled by privileged code. These include *hard traps* caused by errors, such as page faults.
- Interrupts from I/O devices, all of which must be handled by privileged code.
- Interrupts from internal processor counters such as the real time clock.

Each type of trap has its own new processor state and set of vectors. A register called the *trap descriptor block pointer* (TDBP) contains the 34-bit physical address of a series of singlewords containing ordinary tagged pointers, each of which points to the first singleword of a *trap descriptor block entry*:

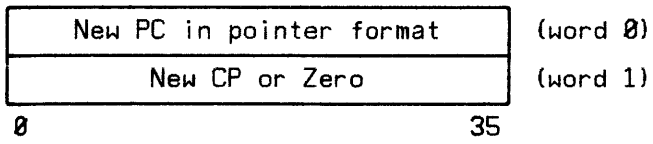
Singleword	Points to trap descriptor block entry for:
0	Ring 0 soft traps
1	Ring 1 soft traps
2	Ring 2 soft traps
3	Ring 3 soft traps
4	Ring 0 TRPSLF traps
5	Ring 1 TRPSLF traps
6	Ring 2 TRPSLF traps
7	Ring 3 TRPSLF traps
8	Ring 0 TRPEXE traps
9	Ring 1 TRPEXE traps
10	Ring 2 TRPEXE traps
11	Ring 3 TRPEXE traps
12	Ring 0 gate calls
13	Ring 1 gate calls
14	Ring 2 gate calls
15	Ring 3 gate calls
16	Hard traps
17	Interrupts from I/O
18	Interrupts from counters

The trap descriptor block entries contain the new context information that the processor loads before handling the trap as well as a pointer to the set of vectors for each of the traps of the type covered by the trap descriptor block entry. Each entry has the following format:



Note that a trap descriptor block entry may lie in any desired ring, and may point to a set of vectors in any ring that can be accessed from the ring containing the trap descriptor block entry. The set of vectors may in turn point to handlers in any ring which can be accessed from the ring containing the vectors. The trap descriptor block entry and the set of vectors for ring 3 soft traps may, for example, lie in ring 2 even though ring 3 cannot access ring 2; but the handlers must lie in ring 2 or ring 3, because ring 2 cannot access rings 0 or 1.

Each set of vectors is an array of doublewords having the following format:



1.9.1 How the Processor Responds to a Trap

When the processor responds to a trap, it follows these steps:

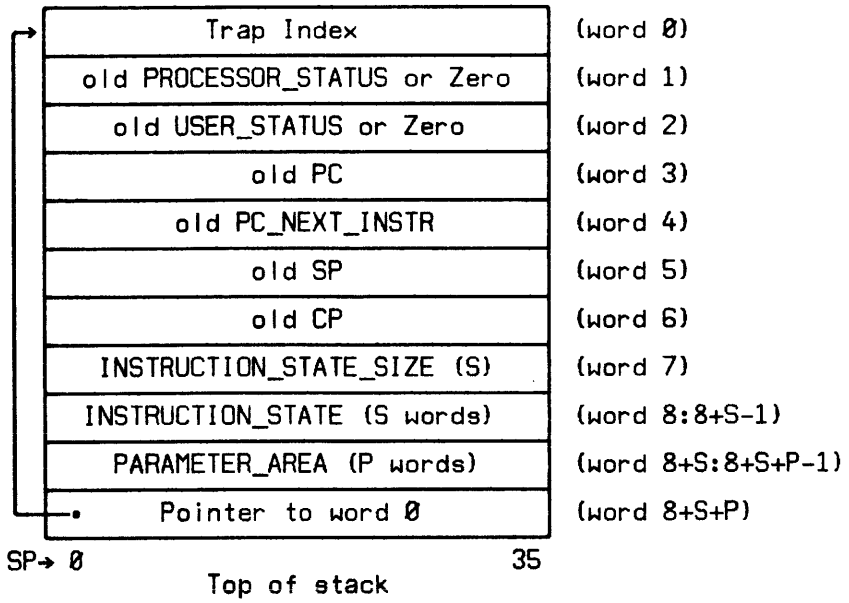
1. First determine a pair of numbers: the *trap type* and the *trap index*. The trap type indicates that the trap is a soft trap, TRPSLF, or a TRPEXE and what ring it came from, or that it is a gate call into a specific ring, or that it is a hard trap, an I/O interrupt or a counter interrupt. The trap index indicates within each trap type which particular TRPSLF, soft trap, hard trap, I/O memory number, counter, TRPEXE, or gate is meant.

The trap index for soft traps are enumerated in section 1.9.3. The trap index for TRPSLF and TRPEXE instructions comes from the value of the modifier 0 . . 63 on the instruction. The destination ring and trap index for gate calls comes for the gate-tagged pointer, which is described in section 1.9.5. The indices for hard traps are enumerated in section 1.9.6. The index for an I/O interrupt is the number of the I/O memory that generated the interrupt. The use of I/O memories is further described in section 1.10. The counters are described in section 1.9.9.

2. Locate the Trap Descriptor Block entry for this trap by adding the trap type times 32 bytes to the Trap Descriptor Block Pointer. If the trap index is greater than word 0 of the TDB entry, then abort the trap sequence and take a TRAP_INDEX_TOO_BIG hard trap. Note that a -1 in this word will effectively disable all traps of that type.

3. Next we begin to build an array of temporary locations which will ultimately be written on the stack. In word 0 write the trap index. In word 1 write the PROCESSOR_STATUS. In word 2 write the USER_STATUS. In word 3 write the PC. In word 4 write PC_NEXT_INSTR. Skip words 5 and 6 for now. In word 7 write the size (in words) of the instruction state. In words 8 through 8+S-1 write the INSTRUCTION_STATE. In words 8+S through 8+S+P-1 write the parameters of the trap. Skip word 8+S+P for now.

This the format of the stack entry which contains the information pushed on the stack:



4. Regardless of any other considerations, load the new PROCESSOR_STATUS from word 1 under mask of word 2 of the TDB entry. Loading under mask means to load only those PROCESSOR_STATUS bits whose corresponding bit positions are one in the mask.

5. In word 5 of the temporary stack array write the CP register. (We are now using the register file specified by the PROCESSOR_STATUS that was loaded under mask in step 4.) In word 6 write the SP register.

6. If word 4 of the TDB entry is non-zero then write it into the SP register.

7. Write the SP register into word 8+S+P of the temporary array.

8. Load word 3 of the TDB entry into USER_STATUS.

9. Locate the trap vector by indexing off of word 5 of the TDB entry by the trap index (as a doubleword index).

10. Validate word 0 of the trap vector. Set the PC to the resulting address and the ring of execution to the output validation level. If word 1 of the trap vector is non-zero, load it into the CP register.

11. If the ring of execution is not privileged then zero word 1 of the temporary stack array.

12. Push the temporary stack entry onto the stack pointed to by SP, leaving SP pointing to the first word after the pushed array.

INSTRUCTION_STATE_SIZE is the number of singlewords occupied by the INSTRUCTION_STATE portion of the stack entry. If it is zero, then INSTRUCTION_STATE

does not appear at all. INSTRUCTION_STATE itself stores instruction-dependent and implementation-dependent information required for restarting the instruction that was in process when the trap occurred. Some instructions are said to be *interruptable*, meaning that interrupts can occur during their execution. A vector arithmetic instruction, for example, may encounter a trap or interrupt part way through the processing of the vector. INSTRUCTION_STATE would, in such a case, contain the information needed to proceed with the remainder of the vector after handling the trap. In many cases it would be incorrect to start over at the beginning of the instruction.

PARAMETER_AREA contains information about the cause of the trap, and varies in content and size from one trap to another. The programmer may infer the size of this area in any particular instance by examining the last word in the stack entry pushed by the trap. The use of the PARAMETER_AREA by soft traps is described in section 1.9.3 and by hard traps in section 1.9.6. TRPSLF and TRPEXE traps use the PARAMETER_AREA for the description of their operand as described in section 1.9.4. A gate call will put one word in the PARAMETER_AREA, namely the gate pointer. Interrupts do not use the PARAMETER_AREA.

1.9.2 Returning from Traps

To return from a trap the RETS.{R,A} instruction is used. OP2 is not used and must be zero. The address of OP1 is the address of the stack entry made by the trap. The processor state is restored from this information.

Ordinarily, with the R (Retry) modifier, the RETS instruction is used to repeat the instruction that was in progress when the trap occurred (that is, the instruction at the PC stored in the stack entry). Whereas the A (Abort) modifier causes RETS to skip to the following instruction.

However, if the instruction that was in progress is interruptable--a vector arithmetic instruction, for example--and the instruction state in the stack entry is non-zero, RETS.R reprocesses the unfinished element of the vector whereas RETS.A skips that element and proceeds with the next.

Since all instructions may safely be restarted whenever they can be interrupted the R modifier is the correct one to use whenever it is undesireable to have a trap disturb the normal instruction sequence. However, it is sometimes necessary to alter the normal flow of operations. For example, after a divide by zero soft trap, one may choose to skip that instruction (or operation of a vector instruction), and in this case the RETS.A instruction is proper.

This is not a privileged instruction. If the ring of execution is not privileged then the old PROCESSOR_STATUS in the stack entry is ignored and the PROCESSOR_STATUS is unchanged.

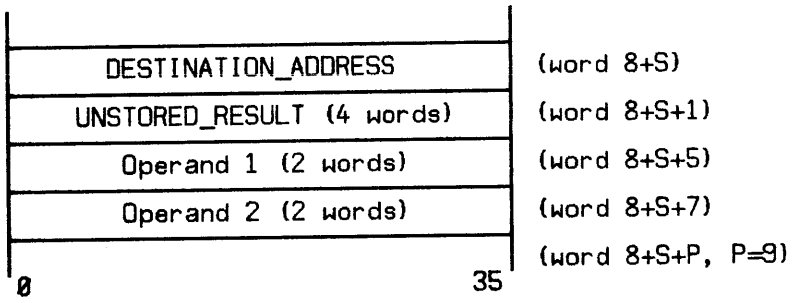
If the ring of execution is privileged and if the instruction state is non-zero, then the A modifier

will cause RETS.A to set the TRACE_PEND bit to match the TRACE_ENABLE bit in the old PROCESSOR_STATUS and the CALL_TRACE_PEND bit to match the saved CALL_TRACE_PEND bit. This is what the instruction would do if it were allowed to finish normally; thus, aborting an instruction does not erroneously disable tracing. The upshot of this behavior is that flow tracing will step through the instructions of a user trap handler such as for a TRPSLF, but will be unaffected by a trap to privileged code such as for a hard trap that returns with a RETS.A.

1.9.3 Soft Traps

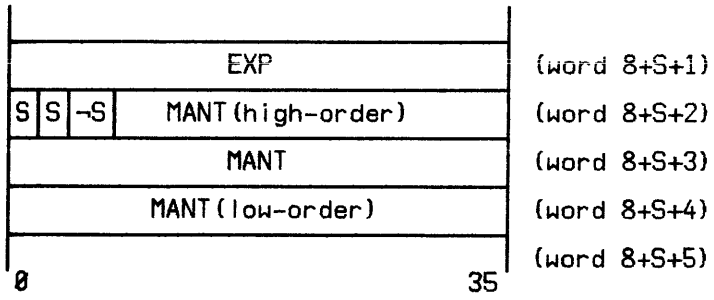
As mentioned earlier, soft traps are those which can be handled without increasing the level of privilege.

Soft traps supply the following information within the PARAMETER_AREA pushed onto the SP stack:



If the destination operand is a memory location, DESTINATION_ADDRESS is a standard pointer with tag and address fields. If the destination is a register, then DESTINATION_ADDRESS gives zero (fault) as its tag and the register address (in terms of quarterwords) as its address.

UNSTORED_RESULT is the result that would have been stored in the destination address if no trap had occurred. If it is an integer, it is sign-extended to be four singlewords long, with the most significant portion in the singleword having the lowest address. If it is a floating point value, it appears in the following format, where "S" is the one-bit sign and "¬S" is the hidden bit (Section 2.2.1):



"Operand 1" and "Operand 2" are the values of the source operands, sign-extended as necessary to be doublewords. If the instruction has only one operand aside from the destination, then "Operand 2" is undefined.

Soft traps include:

0: NO_FAULT No fault has occurred. This trap never occurs; it is defined simply so that software can use the value "0" to encode the absence of a trap.

1: FLT_OVFL_TRAP
Floating point overflow occurred with FLT_OVFL_MODE=0.

2: FLT_UNFL_TRAP
Floating point underflow occurred with FLT_UNFL_MODE=0.

3: FLT_NAN_TRAP
The floating point result was not a valid number and FLT_NAN_MODE=0.

4: INT_OVFL_TRAP
Integer overflow occurred in a signed arithmetic instruction and INT_OVFL_MODE=0.

5: INT_Z_DIV_TRAP
Integer division by zero occurred and INT_Z_DIV_MODE=0.

6: BOUNDS_TRAP
The BNDTRP instruction found its argument out of bounds.

7: FFT_TOO_LONG
An FFT instruction was required to operate on a vector whose size exceeded the maximum for this implementation.

8: LOST_PRECISION
An instruction such as FSIN or FCOS would deliver an imprecise result because its source operand is much larger than 1.

9: ILLEGAL_MATRIX_DIMENSION
The INTRAN or TRANSP insruction tried to use a number of rows or columns that is not a multiple of 8 (for doublewords) or 4 (for singlewords) or 2 (for halfwards).

10: TAG_TRAP The TAGTRP instruction found that the specified bit of the pointer field is 0.

11: UINT_OVFL_TRAP
Integer overflow occurred in an unsigned arithmetic instruction and UINT_OVFL_MODE=0.

1.9.4 TRPSLF and TRPEXE Traps

The TRPSLF and TRPEXE instructions effectively let the user add a number of software-defined instructions to the instruction set. Simply assign a trap vector number to each new instruction and provide a corresponding trap handler routine to implement the instruction. The trap handler is provided with access to the instruction's two operands.

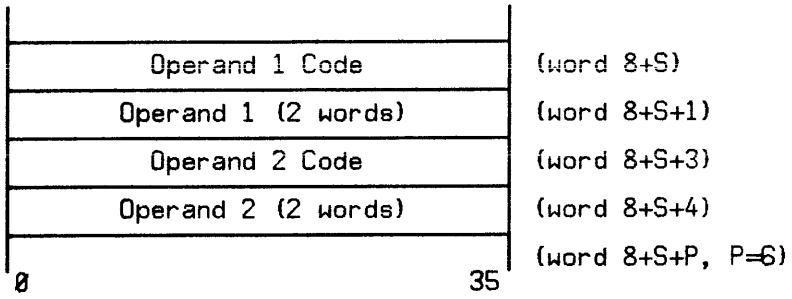
The TRPSLF and TRPEXE opcode modifier selects one of 64 trap vectors each of which may specify a handler address. Each of the two operands is evaluated as a normal XOP and produces a singleword code and a doubleword whose content depends on the code:

code = 0. The operand was an ordinary address. The high word of the doubleword is the address of the operand in pointer format.

code = 1. The operand was a register. The register number is in the high word of the doubleword.

code = 2. The operand was a constant. The doubleword is the constant.

The processor puts these six words in the trap parameter area on the stack in the following order:



This allows simulation of most XOP-like instructions. It doesn't lend itself to simulation of jump- or skip-like instructions.

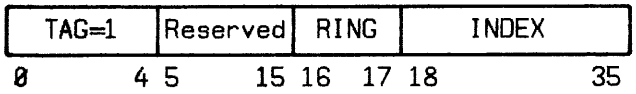
1.9.5 Cross-ring Calls

To simplify the user interface to the operating system, it is desirable to make the mechanism for calling operating system procedures appear identical with the mechanism for calling external procedures in general.

To achieve this, the architecture provides an instruction called CALLX, a special kind of pointer called a *gate pointer*, and a vector of entry points called *gates*. When the CALLX instruction employs a ring or user pointer to specify the called procedure, it cannot--due to the validation mechanism described in Section 1.8.2--call a procedure in a lower-numbered ring. When the CALLX instruction employs a gate pointer, however, it invokes the trap mechanism which permits calling a routine in a lower-numbered ring, but subjects the call to some protective mechanisms.

Thus, the only difference between calling an ordinary external procedure and calling an operating system procedure is in the TAG field of the pointer used to link to the procedure.

The gate pointer is used to determine the destination ring and gate index. The usual pointer ADDRESS field is redefined, so the format of a gate pointer is the following:



A typical operating system would rely on address validation checking to prevent higher-numbered rings from calling or jumping into lower-numbered rings arbitrarily; a user wishing to call into a privileged ring would have to use the gate mechanism. (If the operating system mapped itself into the same address space as the user, it would additionally use the STE.EB execute bracket mechanism to prevent the user from calling operating system routines except via gates.)

1.9.6 Hard Traps

Unless otherwise specified hard traps do not use the PARAMETER_AREA. Hard traps are:

0: NO_FAULT No fault has occurred. This trap never occurs; it is defined simply so that software can use the value "0" to encode the absence of a trap.

Virtual address translation errors: These hard traps are caused by the virtual to physical address translation process that is described in detail in section 1.7.

1: DSEG_SEGMENTITO_FAULT
The VALID field in the STE for a segmentito within the descriptor segment is zero, implying the required segmentito is not present in memory. The parameter area contains a pointer giving the virtual address being referenced.

2: DSEG_PAGE_FAULT
The VALID field in the PTE for a page within the descriptor segment is zero, implying the required page is not present in memory. The parameter area contains a pointer which is the virtual address being referenced.

3: SEGMENTITO_FAULT
The VALID field in the STE for a target segmentito is zero, implying the required segmentito is not present in memory. The parameter area contains a pointer giving the virtual address being referenced.

4: PAGE_FAULT
The VALID field in the PTE for a target page is zero, implying the required page is not present in memory. The parameter area contains a pointer which is the virtual address being referenced.

5: INSTRUCTION_ACCESS_VIOLATION
A word was fetched from the instruction cache that did not have execute permit access. The saved PC gives the virtual address that did not have access.

6: DATA_ACCESS_VIOLATION
A reference was made to either the data cache or an I/O memory and insufficient access was available for the reference. The parameter area contains two words which describe the error. The first is a pointer giving the virtual address being referenced, including the ring. The second word contains two bits (right justified and zero filled) which specify the type of reference. The two bits, MSB and LSB, respecively specify that a read and/or a write reference was made

to the data cache. The actual type of the fault must be determined by comparing the effective access field of the virtual address (see section 1.7.3 for details on how the effective access is calculated) with the two bits specifying the type of reference.

7: EXECUTE_BRACKET_FAULT

An instruction was fetched from a segment whose execute bracket was too low. The saved PC specifies the virtual address which faulted.

8: READ_WRITE_BRACKET_FAULT

A data cache reference was outside the read or write bracket for that segment. The parameter area contains two words. The first is the last base pointer used in the effective address calculation (that is, if you indirect to a second pointer and the second pointer get the fault, it will be that pointer which will appear here). The second word contains four bits of information right justified and zero filled. The two least significant bits are the validation level of the address computation. The upper two bits specify the type of reference that was being made. The MSB is set if a read is being attempted and the LSB is set if a write reference is being made. This information combined with the effective access field for the reference (see section 1.7.3 for details on how the effective access is calculated) can be used to determine the specific error that occured.

9: OUT_OF_BOUNDS

Accessing an operand would have violated segment bounds checking. The parameter area contains two words the first is the base pointer used in calculating the referencing address and the second word is the effective address of the reference.

10-11: Reserved.

Pointer errors: The following two faults result from references through pointers. See section 1.8 for more details.

12: BAD_POINTER_TAG

A MOVP or BASEPTR instruction tried to manipulate a pointer with a fault or reserved tag, or a BASEPTR instruction attempted to reference a NIL or gate tagged pointer, or in the course of pointer validation, a MOVP instruction, tried to alter a ring tag. The parameter area will contain the pointer which caused the fault, and a second word containing the validation level for the pointer.

13: BAD_ADDRESS_TAG

An address was encountered, during the preperation of an operand, that had a gate, NIL, fault or relative pointer tag, or whose validation level was in error. The parameter area contains two words. The first is the last base pointer used in the effective address calculation (that is, if you indirect to a second pointer

and the second pointer get the fault, it will be that pointer that will appear here). The second word contains four bits of information right justified and zero filled. The two least significant bits are the validation level of the address computation. The upper two bits specidy the type of reference that was being made. The MSB is set if a read is being attempted and the LSB is set if a write reference is being made.

14: ILLEGAL_RELATIVE_POINTER

A relative pointer was encountered in a register.

15: Reserved.

Illegal Instruction Traps:

16: ILLEGAL_INSTRUCTION

The instruction opcode is undefined.

17: PRIVILEGE_VIOLATION

A privileged instruction was encountered while in user mode.

18: RESERVED_ADDRESS_MODE

An OD and/or its associated EW has an undefined value. The parameter area will contain an singleword integer whose value is 1 or 2 indicating which OD was in error.

19: OPERAND_NOT_REQUIRED

An instruction was encountered that is defined not to use an operand but the corresponding OD field is not zero. The parameter area will contain a singleword integer whose value is 1 or 2 indicating which OD field was not zero and should have been.

20: ILLEGAL_REGISTER_OPERAND

An instruction specified a register as an operand, but a register is not allowed in this context. The parameter area contains a singleword whose value is 1 or 2 indicating which operand was at fault.

21: ILLEGAL_CONSTANT_OPERAND

An instruction specified a constant operand, but a constant is not allowed in this context. The parameter area contains a singleword whose value is 1 or 2 indicating which operand was at fault.

22-23: Reserved.

Bad Data Format Traps:

24: ALIGNMENT_ERROR

An operand was not properly aligned. The parameter area contains a singleword integer whose value is 1 or 2 indicating which OD was in error.

25: ILLEGAL_BYTE_PTR

The position or offset field of a byte pointer was invalid. The illegal byte pointer returned in the parameter area. If the byte pointer is an immediate byte pointer it will appear in the parameter area as a regular byte pointer but tag and address will form a NIL pointer.

26: ILLEGAL_SHIFT_ROTATE

The bit count for a shift, rotate, or bit reversal instruction was outside the permitted range for the instruction in question. The parameter area contains two words, the first is an integer specifying the precision, in bits, of the operand being shifted. This is the upperbound on the magnitude of the shift count. The second word is the invalid shift count.

27: ILLEGAL_REGISTER

One of the privileged register access instructions specified a register or register file number out of range. The parameter area will contain two words the first containing the register file being addressed and the second containing the number of the register being referenced.

28: ILLEGAL_COUNTER

One of the performance counter referencing instructions refered to a non-existent counter. The parameter area contains the singleword number of the counter that was referenced.

29-31: Reserved.

Debugging Traps: The machine provides several traps to aid in debugging. For more details on how address break points work see section 2.16. For more information on how trace, call trace and ring alarm traps work, see the description of the PROCESSOR_STATUS register in section 1.4

32: INSTRUCTION_BREAK_POINT

The processor fetched an instruction from one of the addresses enabled in the Instruction Break Point List. The old PC field of the stack entry is the address that caused the trap.

33: DATA_BREAK_POINT

The processor fetched an operand from an address enabled in the Data Break Point List. The parameter area contains two words. The first is the virtual address being referenced. The second is a two bit field, right justified, zero filled, which specifies the type of the reference. If the MSB is set a read reference was attempted; if the LSB is set a write reference was attempted. Both bits may be set.

34: TRACE_TRAP

The TRACE_PEND bit in PROCESSOR_STATUS is set.

35: CALL_TRACE_TRAP

The CALL_TRACE_PEND bit in PROCESSOR_STATUS is set.

36: RING_ALARM_TRAP

The ring of execution exceeded the RING_ALARM field of the PROCESSOR_STATUS.

37-39: Reserved.

Bad User or Processor Status Traps:

40: ILLEGAL_PROCESSOR_STATUS

An illegal processor status was loaded by some means. The parameter area contains the illegal processor status. The processor status saved in the trap area is the status before the failed attempt to load a bad processor status.

41: ILLEGAL_USER_STATUS

An illegal user status was loaded by some means. The parameter area contains the illegal user status. The user status saved in the trap area is the status before the failed attempt to load a bad user status.

42: ILLEGAL_TRACE_PEND

An instruction (such as SWITCH or RETS) is attempting to resume execution of an interruptable instruction which was left unfinished due to a trap. The PROCESSOR_STATUS.TRACE_PEND bit is set. Because the TRACE_PEND bit could not have been set at this point in the execution of the interruptable instruction, this indicates that privileged code must have erroneously set the bit some time between the interrupting of the instruction and the attempt to resume execution. The trap occurs on the instruction which attempts to transfer control back to the interruptable instruction, not on the interruptable instruction itself.

43: ILLEGAL_PRIORITY

The WIPND instruction specified a priority level outside the range 0 . . 31.

44-45: Reserved.

Trap Mechanism Traps:

46: TRAP_INDEX_TOO_BIG

A trap occured where the trap index was larger than the maximum index specified in the trap descriptor block entry. The parameter area has two words the first is the trap type, and the second is the trap index that was too big. See section 1.9.1 for a description of trap types and trap indices.

47: Reserved

Memory System Traps:

48: ILLEGAL_IOMEM

An I/O reference to an I/O memory was illegal because the virtual address presented as an operand to the I/O instruction mapped to an illegal I/O memory number. The first word of the parameter area is the virtual address that mapped to an illegal I/O memory number. The second word contains the illegal I/O memory number.

1.9.7 Interrupts

There is one interrupt vector for each I/O memory associated with the processor. Interrupts do not push any PARAMETER information within the stack entry. Interrupts are described further in Section 1.10.

1.9.8 Recursive Traps

When a trap attempts to push information onto the SP stack, a hard trap may occur due to stack overflow, a page fault, an access violation, and so on.

If the original trap was not a hard trap, the SP is left at its original position preceding the soft trap while the hard trap occurs. If the handler for the hard trap solves the stack problem and returns

with a RETS.R instruction, the operation which caused the soft trap is restarted and presumably the original trap will recur, this time completing without encountering a hard trap.

While the processor is trying to locate the trap vector for a trap it may encounter a translation or pointer error. In the case of all traps except a hard trap this will be converted into a trap of the appropriate type. This will allow for instance, the ring 3 soft trap descriptor block entry to be in paged user memory.

In either of these cases, however, if the original trap was a hard trap, the processor will halt. The front end processor must take appropriate action, since this situation indicates a serious operating system failure.

1.9.9 Performance Counter Interrupts

Instructions for manipulating the processor's performance counters are described in section 2.15.

A counter interrupts whenever its value goes from negative to positive. Thus to generate an interrupt in 15000 machine cycles, counter number one should be set to -15000. The processor will increment this every cycle and when it overflows from -1 to zero it will generate an interrupt from counter one. The precise allotment of counter functions to counter numbers is implementation dependent, but the architecture will define counter zero to be a monotonically increasing count of machine cycles. With the appropriate initial setting and multiplication factor this counter's value can be converted to a high resolution wall time clock. Counter one is also defined to count machine cycles, but is allocated for generating real time interrupt, by setting it to the desired negative value.

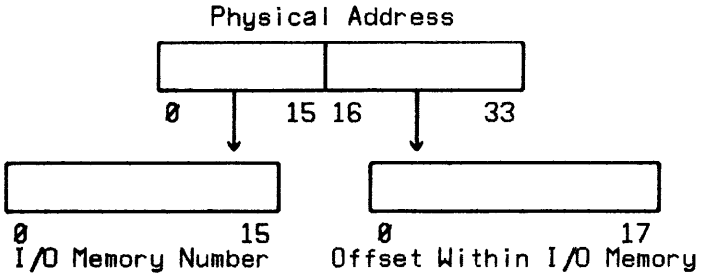
1.10 Input/output

An S-1 processor performs I/O by reading and writing one or more I/O memories, each of which is shared between the S-1 processor and an I/O processor (IOP). The architecture places few constraints on the IOP, which might be a commercially available minicomputer or specially designed hardware. Similarly, the architecture does not dictate how to use the memory to control devices, or how many devices to control through each memory. Instead, these details are determined by the IOP and by the device handler software within the S-1 processor.

An I/O memory appears to the S-1 processor as one or more pages of 36-bit singlewords. The IOP itself may have a much different memory format, because both the hardware and the I/O instructions themselves can provide transformations between the S-1 processor memory format and that of the IOP.

For proper operation, the S-1 processor must set the IO_PAGE bit within the ACCESS field of each of the STEs and PTEs corresponding to an I/O memory page. This permits I/O instructions to access the page and prevents non-I/O instructions from accessing it. The S-1 processor must also set the READ_PERMIT and WRITE_PERMIT bits to grant the access desired. The RB and WB fields in each STE entry will also restrict access to I/O pages.

Each I/O memory has a unique number in the range $0 \ldots 2^{16}-1$. (In a multiprocessor system, the numbers are unique throughout the system, and an attempt by a uniprocessor to refer to an I/O memory not connected to that uniprocessor causes an ILLEGAL_IOMEM hard trap.) When an I/O instruction addresses an operand on an I/O page, the usual virtual-to-physical address translation occurs, and the resulting physical address provides the I/O memory number and the address within that I/O memory:



A vector I/O transfer performs this translation once for the first element of the vector. It obtains succeeding elements from succeeding I/O memory locations, without translating their virtual addresses, even if those elements lie on different pages which might specify different I/O memories or even main memory. If the length of the vector causes it to overrun the end of the I/O memory, the result is undefined.

Each I/O memory has one interrupt whose number is the same as that of the I/O memory, an ENABLE bit which is controlled by the S-1 processor, and a priority ranging from 1 . . 31, which is controlled by the associated IOP. The S-1 processor itself can have a priority ranging from 0 . . 31, specified by the PRIORITY field in PROCESSOR_STATUS. When an interrupt occurs, the S-1 processor traps through the interrupt vector corresponding to the I/O memory number only if the

ENABLE bit is true and the priority of the memory is greater than that of the S-1 processor. Otherwise, the interrupt remains pending until those conditions become true.

If multiple interrupts satisfy those conditions at once, the S-1 processor services them in descending order of priority. When multiple interrupts have the same priority, the S-1 processor services them in a consistent order, but the order is implementation-dependent.

Note that setting the S-1 processor priority to 0 permits every I/O memory to interrupt, while setting it to 31 prevents any I/O memory from interrupting.

Section 1.9 explains how the processor reacts to an interrupt, obtaining a new context from the trap descriptor block entry for I/O interrupts and pushing its old context onto the SP stack. Note that the PRIORITY field in the new PROCESSOR_STATUS obtained from the interrupt vector is ignored. Instead, the processor priority is set to match the priority level of the interrupt and, unless otherwise altered, remains at that level until the interrupt handler returns and restores the old PROCESSOR_STATUS.

1.10.1 I/O Memory Translation

The I/O processors used with the S-1 need not have 9-bit bytes. Regardless of its own rules about byte size, each IOP is connected to the I/O memory so that each byte in the IOP maps onto a byte in the I/O memory (with zero-padding or truncation as need be), and so that as the byte address within the IOP increases, the address of the corresponding byte within the I/O memory likewise increases.

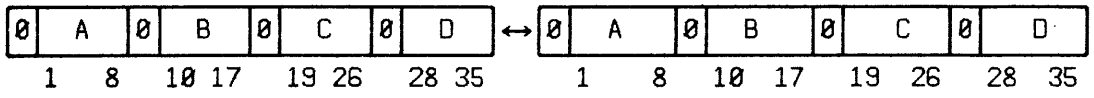
Thus, moving character data between the IOP and the S-1 requires no special treatment.

Moving wider pieces of data between the S-1 processor and an IOP having 8-bit bytes requires the shifting of bits, and may also require reordering of bytes if the two processors disagree on that issue (e.g., MSB first in the S-1 versus LSB first in the IOP).

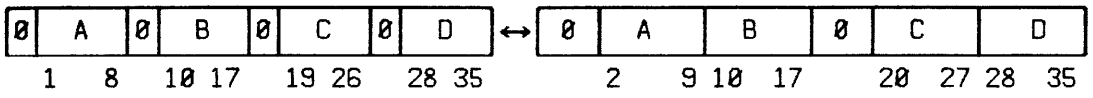
For IOPs with 8-bit bytes, the I/O instructions offer the translations shown in the following illustration. You can think of the words on the left as being in the I/O memory and those on the right as being in the S-1 main memory, although the translations are also available with instructions that move data within the main memory of the S-1. Bits marked "0" are ignored at the source and set to zero at the destination.

Q, H, S, D: 8-bit data, MSB or LSB first

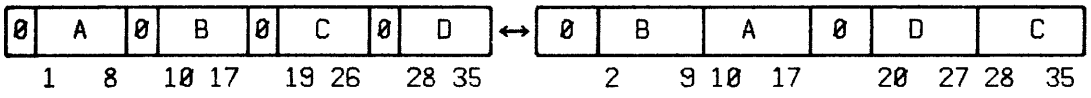
(Though the illustration shows an entire singleword, the precisions Q, H, S, and D move one, two, four, and eight bytes subject to the usual alignment rules.)



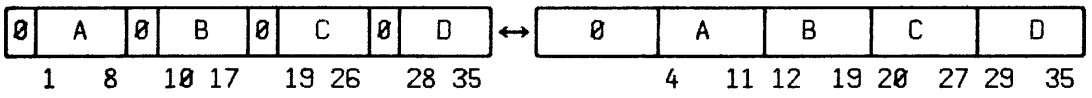
MSB16: 16-bit data, MSB first



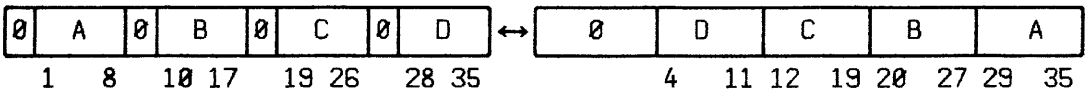
LSB16: 16-bit data, LSB first



MSB32: 32-bit data, MSB first

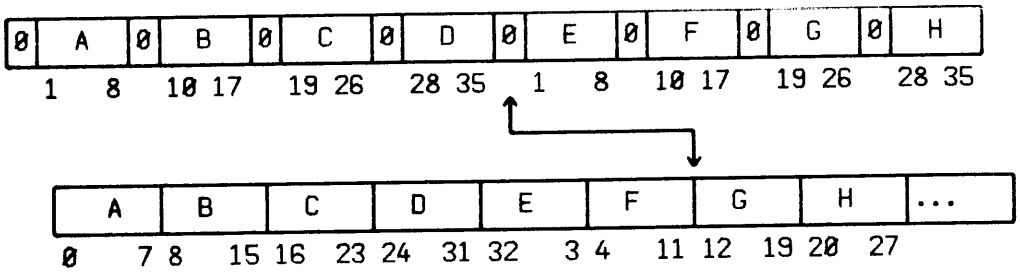


LSB32: 32-bit data, LSB first



B: Pack 9-bit S-1 bytes into 8-bit IOP bytes

(This translation is used only for purposes (such as storing S-1 words on a disk having 8-bit bytes) where the ordering of bytes does not matter so long as a read operation undoes any reordering performed by a write operaton. Thus, there is no LSB-first form.)


0 A 0 B 0 C 0 D 0 E 0 F 0 G 0 H
1 8 10 17 19 26 28 35 1 8 10 17 19 26 28 35
A B C D E F G H ...
0 7 8 15 16 23 24 31 32 3 4 11 12 19 20 27

1.11 Instruction Execution Sequence

The architecture divides the effect of an instruction into two halves, operand evaluation and instruction execution, and requires that the processor behave as if operand evaluation were complete before instruction execution begins.

Thus, unless otherwise stated, all operands required for execution are *prefetched*--that is, all address computations (including indirection) are done and all source operands are available *before* the operation specified by the instruction is performed and *before* results are stored.

The second half, the instruction execution sequence, consists of the following steps:

1. Process interrupts: If an interrupt is pending and has sufficient priority, trap through the appropriate interrupt vector to the specified interrupt handler. On returning from the interrupt handler, start at the beginning of step 1 again, so that if further interrupts are pending, they will also be serviced.

2. Process trace traps and clear the TRACE_PEND bit: If the TRACE_PEND bit in PROCESSOR_STATUS is 1, set TRACE_PEND to 0 so that traps encountered in step 3 do not cause the instruction to be traced redundantly, and invoke the TRACE_TRAP handler. Next, if the CALL_TRACE_PEND bit in PROCESSOR_STATUS is 1, set CALL_TRACE_PEND to 0 so that traps encountered in step 3 do not cause the instruction to be traced redundantly, and invoke the CALL_TRACE_TRAP handler. Finally, if either handler was invoked, restart the instruction-execution sequence at step 1.

3. Process pre-operation traps: If any other traps (such as page faults or illegal memory accesses) that can be detected prior to the operation specified by the instruction are pending, invoke the appropriate trap handlers. On returning from the last trap handler, restart the instruction-execution sequence at step 1.

4. Save TRACE_ENB and CALL_TRACE_ENB: Save the values of the TRACE_ENB and CALL_TRACE_ENB bits internally.

5. Operation: Perform the specific operation defined for this instruction, after first examining the instruction state. Some lengthy instructions--vector instructions, for example--are said to be *interruptable*. This means that an interrupt can suspend execution during step 5, saving the state of the instruction execution on the SP stack in INSTRUCTION_STATE as described in Section 1.9. Thus, if the instruction is known to be interruptable, and INSTRUCTION_STATE indicates the instruction is in such a state of suspended execution, step 5 will pick up where execution left off; otherwise, step 5 will start from the beginning.

When an instruction is interrupted in the fashion just described, the processor proceeds to execute the instructions of the trap handler, following this sequence for each one. On returning from the trap handler, the processor reencounters the interrupted instruction, and begins processing it again from step 1. Only when the processor reaches step 5 and

interrogates INSTRUCTION_STATE does it become clear that this is the resumption of a suspended instruction.

6. Process post-operation traps: If any traps (such as arithmetic overflow) resulted from step 5, invoke the appropriate trap handlers.

7. Set TRACE_PEND and CALL_TRACE_PEND: If the value of TRACE_ENB saved in step 4 is 1, set TRACE_PEND to 1. Thus, if tracing was enabled when this instruction commenced or if this instruction itself sets TRACE_PEND during step 5, a trace trap will occur on the following instruction even if the following instruction disables tracing.

Similarly, if the value of CALL_TRACE_ENB saved in step 4 is 1, and the instruction just executed in step 5 was a call or return (Section 2.11 defines these), then set CALL_TRACE_PEND to 1.

8. Clear the instruction state.

2 Instruction Set

This section describes the S-1 native mode instruction set. For conciseness, it assumes familiarity with the architecture as described in Section 1; for example, instead of explicitly stating the number and types of operands for each instruction, it simply classifies each instruction as an XOP, TOP, HOP, SOP, or JOP. Similarly, it avoids restating again and again the rules given in Section 1 for vector operands.

2.1 Integer Arithmetic

Signed integer arithmetic instructions interpret their operands--whether quarterwords, halfwords, singlewords, or doublewords--as two's complement data. For any given precision, we call the largest positive integer *MAXNUM* and the negative integer with the largest magnitude *MINNUM*.

Precision	MINNUM	MAXNUM
Quarterword	-256	255
Halfword	-131 072	131 071
Singleword	-34 359 738 368	34 359 738 367
Doubleword	-2 361 183 241 434 822 606 848	2 361 183 241 434 822 606 847

The unsigned integer data type uses no sign bit, making all bits of the word available for representing magnitude. Thus, whereas a signed quarterword ranges from -2^8 to 2^8-1, an unsigned quarterword ranges from 0 to 2^9. We call the largest positive unsigned integer *UMAXNUM*.

The primary difference between the signed and unsigned arithmetic is in the treatment of overflow. We define separate overflow traps and bits for signed and unsigned arithmetic.

2.1.1 Integer Arithmetic Exceptions

Inside the USER_STATUS register, four bits called CARRY, INT_OVFL (signed integer overflow), UINT_OVFL (unsigned integer overflow), and INT_Z_DIV (integer division by zero) record the *side effects* or *exceptions* that occur during integer arithmetic. INT_OVFL, UINT_OVFL and INT_Z_DIV are *sticky*--that is, integer arithmetic operations may set them but never clear them, so once one of these bits is set it remains set until explicitly cleared by manipulating USER_STATUS. CARRY is not sticky; instructions which affect CARRY will clear it if they do not set it.

CARRY Carry-out or borrow-in from integer arithmetic.

INT_OVFL Signed integer overflow (that is, the result is greater than MAXNUM or the result is less than MINNUM).

UINT_OVFL Unsigned integer overflow (that is, the result is greater than UMAXNUM or the result is less than zero).

INT_Z_DIV Integer division by zero.

For example, the following three instructions set CARRY, INT_OVFL, and INT_Z_DIV:

```
INC.S RTA,#-1    ; -1+1 invokes CARRY
INC.S RTA,#[377777,,777777] ; MAXNUM+1 invokes INT_OVFL
RECIP.S RTA      ; Remainder(RTA/0) invokes INT_Z_DIV
```

Three additional fields called INT_OVFL_MODE, UINT_OVFL mode and INT_Z_DIV_MODE tell the processor how to respond to the INT_OVFL and INT_Z_DIV exceptions respectively--whether to trap or what to use as the result of the arithmetic operation which encountered the exception. (Note that setting one of the exception bits by manipulating USER_STATUS will not produce the specified response; the bit must be set by integer arithmetic):

INT_OVFL_MODE

0	Invoke INT_OVFL_TRAP soft trap without storing a result.
1	Retain as many low-order bits of the result as will fit in the operand (9 bits for quarterwords, 18 for halfwords, etc.)

UINT_OVFL_MODE

0	Invoke UINT_OVFL_TRAP soft trap without storing a result.
1	Retain as many low-order bits of the result as will fit in the operand (9 bits for quarterwords, 18 for halfwords, etc.)

INT_Z_DIV_MODE

0	Invoke INT_Z_DIV_TRAP soft trap without storing a result.
1	Use 0 as the result.

2.1.2 CARRY Algorithm

To determine whether a particular instruction sets CARRY, evaluate the following formula. X1, X2, and X3 are the values shown for that instruction in the following table, and C_IN is the state of CARRY at the beginning of the instruction:

$$CARRY = (X1<0 \wedge X2<0) \vee [(X1<0 \vee X2<0) \wedge (X1+X2+X3 \geq 0)]$$

In the following table, "¬" means *one's-complement*; and "-1" is the two's-complement of 1.

Instruction	X1	X2	X3
ADD	S1	S2	0
ADDC	S1	S2	C_IN
SUB	S1	¬S2	1
SUBV	¬S1	S2	1
SUBC	S1	¬S2	C_IN
SUBCV	¬S1	S2	C_IN
INC	1	OP2	0
			(i.e., CARRY:=1 if OP2 = -1)
DEC	-1	OP2	0
			(i.e., CARRY:=1 if OP2 <> 0)
NEG	0	¬OP2	1
			(i.e., CARRY:=1 if OP2 = 0)
NEGC	0	¬OP2	C_IN
ABS	0	¬OP2	1
			(only if OP2 is negative)
UADD	S1	S2	0
USUB	S1	¬S2	1
USUBV	¬S1	S2	1

VBADD sets CARRY according to the last addition (i.e., the addition involving the first, high-order elements of the vector operands.) VBSUB, VBSUBV, and VBNEG behave in the analogous fashion.

2.1.3 Integer Rounding Modes

Integer rounding occurs during division, modulus, and conversion from a real number to an integer under control of the INT_RND_MODE field of the USER_STATUS register. Using the WRNDMD.INT instruction, it may be assigned any of the rounding modes explained in section 2.2.5 (which describes the rounding modes in the context of floating point numbers); however, any instruction which uses INT_RND_MODE, resets it to diminished-magnitude. This permits you to select an unusual rounding mode which remains in effect only until the end of the next instruction which invokes rounding.

2.1.4 Integer exceptions during vector instructions

When an integer arithmetic instruction can affect CARRY, the corresponding vector instruction (if any) leaves CARRY in an undefined state. When an integer arithmetic instruction can affect INT_OVFL, the corresponding vector instruction will leave INT_OVFL set to the logical OR of the settings for each of the scalar operations it performs.

2.1.5 Integer Arithmetic Instructions

ADD

Integer add

ADD . {Q,H,S,D} **TOP**
VADD . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: ADD performs DEST:=S1+S2. VADD adds each element of vector OP1 to the corresponding element of vector OP2, storing the result either back in vector OP1 or in the vector pointed to by SR0.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: For the scalar instruction, S1, S2, and DEST each have the precision specified by the modifier. For the vector instruction, each element of each vector has the precision specified by the modifier.

Carry is set by the following instruction. Note that 777 has the signed interpretation -1 and the unsigned interpretation 2^9-1:

```
ADD.Q RTA,#333,#777          ;RTA:=332 (QW)
```

ADDC — Integer add with carry

ADDC . {Q,H,S,D} TOP

Purpose: DEST:=S1+S2+CARRY

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: S1, S2, and DEST all have the precision specified by the modifier.

Carry is set after the execution of the first instruction, and cleared after the second:

```
ADD.Q RTA,#666,#777          ;RTA:=665 (QW)
ADDC.Q RTA,RTA,#1            ;RTA:=667 (QW)
```

Assume INT←OVFL←MODE is set not to trap. The following adds two "quadruple-word" integers at X and Y represented as a pair of DWs with the low-order DW having the higher address. The result is stored in X and X+8:

```
ADD.D X+8.,Y+8.
ADDC.D X,Y
```

Similarly, suppose that NUM1 and NUM2 are two blocks of singlewords, each of length N (N≥2) and representing an N-word integer, with lower-order words having higher addresses. These can be added and the result stored in an (N+1)-word block NUM3 in this manner:

```
        MOV.S.S RTB,#<N-1>                          ;RTB counts words
        ADD.S RTA,NUM1[RTB]↑2,NUM2[RTB]↑2           ;add low-order words
        MOV.S NUM3+4*1[RTB]↑2,RTA                   ;store low-order result
LOOP:   ADDC.S RTA,NUM1-4*1[RTB]↑2,NUM2-4*1[RTB]↑2 ;add next words plus carry
        MOV.S.S NUM3[RTB]↑2,RTA                     ;store next word
        DJMPZ.GTR RTB,LOOP                          ;DJMPZ does not alter carry!
        CMPSF.LSS.S RTA,NUM1,#0                     ;produce sign-extension of
        CMPSF.LSS.S RTB,NUM2,#0                     ; NUM1 and NUM2
        ADDC.S NUM3,RTA,RTB                         ;produce high-order result
```

SUB — Integer subtract

SUB . {Q,H,S,D}	**TOP**
SUBV . {Q,H,S,D}	**TOP**
VSUB . {SR,OP1} . {H,S,D}	**V:=VV**
VSUBV . {SR,OP1} . {H,S,D}	**V:=VV**

Purpose: SUB computes DEST:=S1-S2; SUBV computes DEST:=S2-S1. VSUB and VSUBV are the vector equivalents of SUB and SUBV respectively; they both put the result either back into OP1 or into the vector pointed to by SR0.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: For the scalar instructions, S1, S2, and DEST each have the precision specified by the modifier. For the vector instructions, each element of each vector has the precision specified by the modifier.

This example subtracts 1 from -1 to obtain -2. After execution, CARRY is set, INT_OVFL is unchanged, and RTA contains -2:

```
SUB.S RTA,#-1,#1          ;RTA:=-2
```

SUBC — Integer subtract with carry

SUBC . {Q,H,S,D} **TOP**
SUBCV . {Q,H,S,D} **TOP**

Purpose: SUBC computes DEST:=S1-S2-1+CARRY; SUBCV computes DEST:=S2-S1-1+CARRY.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: S1, S2, and DEST all have the precision specified by the modifier.

Let X and Y be two pairs of DWs representing a long integer with the low-order DW having the lower address. The following sets X to the difference of X and Y:

```
SUB.D X,Y
SUBC.D X+8.,Y+8.
```

MULT **Integer multiply**

MULT . {Q,H,S,D} **TOP**

Purpose: MULT computes DEST:=LOW_ORDER(S1*S2).

Restrictions: None

Exceptions: INT_OVFL

Precision: For the scalar instruction, S1, S2, and DEST all have the precision specified by the modifier.

INT_OVFL is set by the following instruction which multiplies 333 octal by 3, giving a result—1221 octal—which is larger than can fit in nine bits:

```
MULT.Q RTA,#[333],#3     ;RTA:=221 (QW)
```

MULTL — Integer multiply long, long result

MULTL . {Q,H,S} **TOP**

Purpose: DEST:=S1*S2

Restrictions: None

Exceptions: None

Precision: S1 and S2 have the same precision as the modifier. DEST has a precision *twice* that of the modifier and must be aligned accordingly.

The following instruction does *not* set INT_OVFL since the result fits in a halfword:

```
MULTL.Q RTA,#[333],#3    ;RTA:=001221 (HW)
```

UADD — Unsigned integer addition

UADD . {Q,H,S,D} TOP

Purpose: DEST := S1 + S2, where the operands are unsigned numbers.

Restrictions: None

Exceptions: UINT_OVFL occurs if the true result exceeds the precision of the destination.

Precision: S1, S2, and DEST all have the precision specified by the modifier.

The following forms the sum of the two constant operands. Since the true result is greater than 2^9-1, the CARRY and UINT_OVFL bits are set. RTA is assigned the low order bits of the result.

```
UADD.Q RTA,#777,#4        ; RTA := 3
```

USUB — Unsigned integer subtraction

USUB . {Q,H,S,D} TOP
USUBV . {Q,H,S,D} TOP

Purpose: USUB computes DEST:=S1-S2, and USUBV computes DEST:=S2-S1, where the operands are unsigned numbers.

Restrictions: None

Exceptions: UINT_OVFL occurs if the true result is less than zero.

Precision: S1, S2, and DEST all have the precision specified by the modifier.

The following computes the difference of the two constant operands. Since the true result is less than zero, the CARRY and UINT_OVFL bits are set. RTA is assigned the low order bits of the result.

```
USUB.Q RTA,#4,#777      ; RTA := 5
```

UMULT — Unsigned integer multiply

TOP

UMULT . {Q,H,S,D}

Purpose: DEST:=LOW_ORDER(S1*S2)

Restrictions: None

Exceptions: UINT_OVFL occurs if the true result exceeds the precision of the destination.

Precision: S1, S2, and DEST all have the precision specified by the modifier.

The following instruction puts the low order QW of the unsigned square of 2^9-1 in RTA. This value is the low-order nine bits of $2^{18}-2^{10}+1$, that is, 001. Since the full result is greater than 2^9-1, INT_OVFL is also set:

```
UMULT.Q RTA,#777,#777 (QW)
```

UMULTL — Unsigned integer multiply, long result

UMULTL . {Q,H,S} TOP

Purpose: DEST:=S1*S2

Restrictions: None

Exceptions: None

Precision: S1 and S2 have the same precision as the modifier. DEST has a precision *twice* that of the modifier and must align accordingly.

The following instruction puts the unsigned square of 2^9-1 in RTA. This value is $2^{18}-2^{10}+1$--that is, 776001:

```
UMULTL.Q RTA,#777,#777  ; RTA:=776001 (HW)
```

RECIP, MDIV, MDIVH — Integer division and modulus

RECIP . {Q,H,S,D}	**XOP**
MDIV . {FL,US} . {Q,H,S,D}	**XOP**
MDIVH . {FL,US} . {Q,H,S,D}	**TOP**
URECIP . {Q,H,S,D}	**XOP**
UMDIV . {Q,H,S,D}	**XOP**
UMDIVH . {Q,H,S,D}	**TOP**

Purpose: Integer division and modulus are accomplished with combinations of these instructions. We will describe in detail the RECIP, MDIV, and MDIVH instructions, which perform signed arithmetic. The URECIP, UMDIV, and UMDIVH instructions are the corresponding instructions for unsigned arithmetic. In the descriptions, "Precision" represents 9, 18, 36, or 72 bits, according to the {Q,H,S,D} modifier:

The RECIP instruction computes a scaled reciprocal with an error ϵ such that $0 <= abs(\epsilon) < 1$. OP2 has the precision of the {Q,H,S,D} modifier. OP1 becomes a fixed-point fraction, twice the precision of the {Q,H,S,D} modifier, whose binary point lies between bits 2 and 3. Thus -4 <= OP1 < 4:

```
OP1 := (2 ** (2 * <Precision> - 3) / OP2) - ε;
```

The MDIV instruction is used following a RECIP instruction and preceding an MDIVH instruction when both integer division and modulus are desired. It takes in OP2 the fixed-point fraction generated by RECIP, multiplies it by OP1, and undoes the scaling so as to put into RTA the result of an integer division and to put into RTB a partial result which MDIVH can operate on. OP2 and RTB have twice the precision of the {Q,H,S,D} modifier. OP1 and RTA have the same precision as the {Q,H,S,D} modifier. If the {FL,US} modifier is FL, the instruction uses the floor rounding mode. If the {FL,US} modifier is US, it uses the rounding mode from the INT_RND_MODE field of the USER_STATUS register, and resets resets that field to diminished-magnitude rounding; this permits you to select a particular rounding mode which remains in effect for a single operation and then returns to the standard mode (treating all numbers as integers):

```
RTA := Round(OP1 * OP2 / 2 ** (2 * <Precision> - 3));
RTB := OP1 * OP2 - (RTA * (2 ** (2 * <Precision> - 3)));
```

The MDIVH instruction performs the "first half" of the MDIV instruction, taking in S2 the fixed-point fraction generated by RECIP (or as the intermediate result of an MDIV instruction), multiplying it by S1, and undoing the scaling so as to put into DEST the result of an integer division. DEST and S1 have the precision of the {Q,H,S,D} modifier. S2 has twice the precision of the {Q,H,S,D} modifier. The {FL,US} modifier serves the same purpose as it does in MDIV:

```
DEST := Round(S1 * S2 / 2 ** (2 * <Precision> - 3));
```

URECIP is similar to RECIP, but $0 < abs(\epsilon) <= 1$, and the binary point lies to the left of bit 0 of the double-precision result, so that $0 <= OP1 < 1$:

```
OP1 := (2 ** (2 * <Precision>) / OP2) - ε;
```

UMDIV is similar to MDIV, but all operands are unsigned and only the floor rounding mode is appropriate. Note that UMDIV.D is precisely an unsigned multiply of doubleword OP2 times quadword OP1, storing a six-word result into the registers starting at RTA:

```
RTA := Floor(OP1 * OP2 / 2 ** (2 * <Precision>));
RTB := OP1 * OP2 - (RTA * 2 ** (2 * <Precision>));
```

UMDIVH is similar to MDIVH, but all operands are unsigned and only the floor rounding mode is appropriate:

```
DEST := Floor(S1 * S2 / 2 ** (2 * <Precision>));
```

Restrictions: None

Exceptions: INT_OVFL can occur with the signed instructions; INT_Z_DIV can occur with both the signed and unsigned instructions.

Precision: See the discussion under "Purpose".

To obtain both integer division and modulus using the Floor rounding mode:

```
RECIP.S RTB,DIVISOR      ; RTB is DW
MDIV.FL.S DIVIDEND,RTB   ; RTA := DIVIDEND DIV DIVISOR
MDIVH.FL.S RTB,DIVISOR,RTB
                         ; RTB := DIVIDEND MOD DIVISOR
```

To obtain both integer division and modulus using any desired rounding mode (note that in this sequence, you should apply the desired rounding mode to the MDIV instruction but *not* to the MDIVH instruction, which should use the Floor rounding mode in order to obtain the correct modulus):

```
RECIP.S RTB,DIVISOR      ; RTB is DW
WRNDMD.TMP <RNDMODE>
MDIV.US.S DIVIDEND,RTB   ; RTA := DIVIDEND DIV DIVISOR
MDIVH.FL.S RTB,DIVISOR,RTB
                         ; RTB := DIVIDEND MOD DIVISOR
```

To obtain integer division, using the Floor rounding mode:

```
RECIP.S RTA,DIVISOR      ; RTA is DW
MDIVH.FL.S RTA,DIVIDEND,RTA
                         ; RTA := DIVIDEND DIV DIVISOR
```

To obtain integer division, using any desired rounding mode:

```
RECIP.S RTA,DIVISOR      ; RTA is DW
WRNDMD.TMP <rounding mode>
MDIVH.US.S RTA,DIVIDEND,RTA
                         ; RTA := DIVIDEND DIV DIVISOR
```

One advantage of these sequences of instructions in comparison with explicit DIV and MOD instructions is that whenever DIVISOR is constant, the RECIP operation (which is by far the most expensive one) can be done at compile time:

```
MDIVH.FL.S RTA,[HRECIP36 ? LRECIP36],DIVIDEND
                         ; RTA := DIVIDEND DIV 36. where the
                         ;   concatenation of HRECIP36 with
                         ;   LRECIP36 is the doubleword value
                         ;   which RECIP X,36. would compute
```

INC — Integer increment

INC . {Q,H,S,D} **XOP**

Purpose: OP1:=OP2+1

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1 and OP2 have the same precision as the modifier.

The following adds one to RTB and stores the result in RTA.

```
INC.S RTA,RTB    ;RTA:=RTB+1
```

If the source and destination are identical, ADD is preferable from a performance standpoint:

```
ADD.S RTA,#1     ;RTA:=RTA+1
```

DEC — Integer decrement

DEC . {Q,H,S,D} **XOP**

Purpose: OP1:=OP2-1

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1 and OP2 have the same precision as the modifier.

The following subtracts one from A and puts the result in B:

```
DEC.S B,A        ;B:=A-1
```

If the source and destination are identical, SUB is preferable from a performance standpoint:

```
SUB.S B,#1       ;B:=B-1
```

TRANS — Signed integer translate

TRANS . {Q,H,S,D} . {Q,H,S,D} **XOP**
VTRANS . {Q,H,S,D} . {Q,H,S,D} **V:=V**

Purpose: TRANS copies a signed integer from OP2 to OP1, converting its precision if necessary by sign-extending or by discarding high order bits.

VTRANS performs TRANS on individual elements of vector OP2 and stores the result in vector OP1. If the source and destination vectors have the same precision, the vectors may overlap; the instruction guarantees not to alter any element of the source until it has copied that element to the destination.

If the source vector's precision exceeds that of the destination vector, the two vectors may be identical, but must not otherwise overlap.

If the source vector's precision is less than that of the destination vector, the two vectors may not overlap at all.

Restrictions: None

Exceptions: INT_OVFL

Precision: OP1 has the precision of the first modifier and OP2 has the precision of the second modifier.

The second instruction illustrates the sign-extension of TRANS:

```
MOV.H.Q RTA,#-1          ;RTA:=000777 (HW)
TRANS.H.Q RTA,#-1        ;RTA:=777777 (HW)
```

NEG **Integer negate**

NEG . {Q,H,S,D} **XOP**
VNEG . {H,S,D} **V:=V**

Purpose: For NEG, OP1:=*two's-complement*(OP2).

VNEG performs NEG on each element of the vector beginning with OP2 and stores the results in the vector beginning with OP1.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1 and OP2 have the same precision as the modifier.

The following negates the value in RTA:

```
NEG.S RTA                ;RTA:=-RTA
```

This piece of code jumps to TWOPOWER if the non-negative singleword integer in HUNOZ is an exact power of two (where zero is considered to be such a power):

```
NEG.S RTA,HUNOZ          ;RTA:=-HUNOZ
ANDCT.S RTA,HUNOZ        ;RTA:=(-RTA)∧HUNOZ
JMPZ.EQL.S RTA,TWOPOWER ;jump if RTA now is zero
```

The BITCNT instruction can be used to do the same thing if zero is not to be considered a power of two.

NEGC — Integer negate with carry

NEGC . {Q,H,S,D} **XOP**

Purpose: OP1:=*one's-complement*(OP2) + CARRY.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1 and OP2 have the same precision as the modifier.

The following negates a quadword integer in R8:

```
NEG.D R8,R8          ; negate low order part
NEGC.D R10,R10       ; negate high part
```

ABS — Integer absolute value

ABS . {Q,H,S,D} **XOP**
VABS . {H,S,D} **V:=V**

Purpose: For ABS, OP1:=*abs*(OP2).

VABS performs ABS on each element of the vector beginning at OP2 and stores the results in the vector beginning at OP1.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1 and OP2 have the same precision as the modifier.

The following takes the absolute value of RTB and puts it in RTA:

```
ABS.S RTA,RTB    ;RTA:=|RTB|
```

MIN — Integer minimum

MIN . {Q,H,S,D} **TOP**
VMIN . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: MIN stores in DEST the smaller of the signed integers S1 and S2.

VMIN performs MIN on a series of pairs: one element from the vector beginning with OP1 and the corresponding element of the vector beginning with OP2. If the first modifier is OP1, results go back into the vector beginning with OP1; if it is SR, they go into the vector pointed to by SR0.

Restrictions: None

Exceptions: None

Precision: For MIN, operands S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VMIN, the elements of each vector have the precision specified by the {H,S,D} modifier.

The following sets RTA to 0 if RTA is positive:

```
MIN.S RTA,RTA,#0
```

MAX — Integer maximum

MAX . {Q,H,S,D} **TOP**
VMAX . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: MAX places in DEST the larger of the signed integers S1 and S2.

VMAX performs MAX on a series of pairs: an element from the vector beginning with OP1 and the corresponding element of the vector beginning with OP2. If the first modifier is OP1, the instruction stores the results back into the elements of vector OP1; if the modifier is SR, it stores the results into the vector pointed to by SR0.

Restrictions: None

Exceptions: None

Precision: For MAX, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VMAX, the elements of each vector have the precision specified by the {H,S,D} modifier.

The following sets RTA to 100 if RTA is less than 100:

```
MAX.S RTA,RTA,#[100]
```

Suppose that A and B are two byte pointers. Then the following instruction puts in RTA the byte pointer which indicates the byte starting higher in memory than the other; or, if they start at the same bit, whichever points to the longer byte. (This is a consequence of the representation of byte pointers--see Section 2.9). Similarly, all D-precision integer comparison instructions—such as MIN.D, CMPSF.D, SKP.D, etc.--can be used to compare byte pointers in this fashion:

```
MAX.D RTA,A,B          ;RTA := pointer to higher byte
```

LMINMAX — Lengthwise integer minimum and maximum

LMINMAX . {H,S,D} **SS:=V**

Purpose: Select the minimum and maximum elements of a vector of signed integers whose first element is OP1. Put the minimum in RTA and the maximum in RTB. If the vector length is zero or negative RTA will be set to MAXNUM and RTB will be set to MINNUM.

Restrictions: None

Exceptions: None

Precision: RTA, RTB, and each element of vector OP1 have the precision of the modifier.

The following sets RTA to -4 and RTB to 16:

```
MOV.S.S SIZEREG,#7
LMINMAX.S [7 ? 12. ? -2 ? -4 ? 8. ? 16. ? 3]
```

ADDSUB — Integer add-and-subtract

ADDSUB . {Q,H,S,D}	**XOP**
ADDSUBV . {Q,H,S,D}	**XOP**
VADDSUB . {Q,H,S,D}	**VV:=VV**
VADDSUBV . {Q,H,S,D}	**VV:=VV**

Purpose: ADDSUB computes the sum and difference of a pair of integers:

```
TEMP := OP1 + OP2;
OP2 := OP1 - OP2;
OP1 := TEMP;
```

ADDSUBV reverses the roles of OP1 and OP2 on input:

```
TEMP := OP2 + OP1;
OP2 := OP2 - OP1;
OP1 := TEMP;
```

VADDSUB and VADDSUBV perform the analogous operations on successive elements of vectors.

Restrictions: None

Exceptions: CARRY, INT_OVFL

Precision: OP1, OP2, and each element of each vector have the precision specified by the modifier.

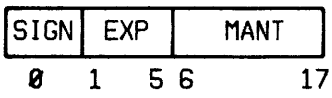
Compute the sum and difference of 4 and 5:

```
MOV.S.S RTA,4
MOV.S.S RTB,5
ADDSUB.S RTA,RTB        ; RTA:=9; RTB:=-1
```

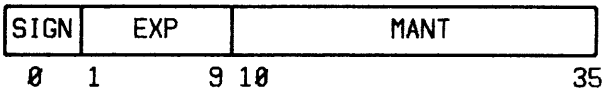
2.2 Floating Point Arithmetic

2.2.1 Floating Point Data Format

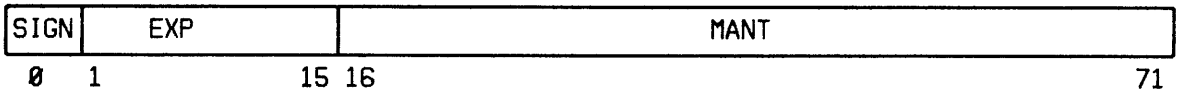
Floating point data can occur in three of the four standard precisions: halfword, singleword, or doubleword. The floating point representation is made up of three fields: SIGN, EXP, and MANT.



Halfword floating point format



Singleword floating point format



Doubleword floating point format

SIGN is 1 if the floating point number is negative.

EXP is the exponent, expressed in excess-16 format in halfword precision, excess-256 format for singleword precision, or excess-16384 format for doubleword precision. If SIGN is 1 (that is, the number is negative, EXP is one's complemented.

MANT represents only part of the true mantissa of the number; to obtain the entire mantissa, concatenate the sign bit, a *hidden bit*, a binary point, and the MANT field and treat as a 2's complement number:

<SIGN><hidden bit>.<MANT>

The result of concatenating these fields is a two's complement number. This number is always normalized to obey the following:

$$1 \leq \text{mantissa} < 2$$
or
$$-2 \leq \text{mantissa} < -1$$

As a result, <hidden bit> and SIGN are always opposites, and it is possible to omit <hidden bit> from the floating point representation and infer its value from that of SIGN.

To summarize: if the sign is zero, the number is:

$$\text{<entire mantissa>} * 2^{(\text{<exponent>} - \text{<excess>})}$$

whereas if the sign is one, the number is:

$$\text{<entire mantissa>} * 2^{(\textit{one's complement}(\text{<exponent>}) - \text{<excess>})}$$

Converting to floating point format: While the FLOAT instruction automatically converts an integer to floating point format, the following description of an algorithm for doing so may help make the format clear:

1. Set the SIGN field of the floating point version to 0.

2. Multiply a copy of the number by 2^x, where you choose x so the result is greater than or equal to 1 but less than 2. Set the EXP field to (-x+16) for a quarterword, (-x+256) for a singleword, or (-x+16384) for a doubleword.

3. Starting with the most significant bit of the original number, discard bits until you encounter the first 1-bit. Discard it, too. Place the remaining bits into the MANT field, left-justified.

To convert zero to floating point format, set the entire word to 0 (zero is an exceptional case).

To convert a negative integer to floating point format, take its absolute value and represent that according to the steps just given for positive integers. Then take the two's complement negation of the entire floating point representation, without regard to format.

(For the skeptical, here is an outline for a proof that two's-complement negation works correctly on floating point numbers. If MANT <> 0 then no carry from the two's-complement operation can reach the EXP field, since it will be absorbed by the right-most, non-zero MANT bit. Therefore, the EXP field will be one's-complemented. If MANT = 0 then there are three cases. Case 1: The floating point number was originally negative. The mantissa was, therefore, -2.0 and the floating point number was $-2^{\text{exponent}+1}$. When this number is two's-complemented, the MANT field is still zero but the EXP field is two's-complemented. The mantissa becomes 1 and the carry from the fraction has increased the exponent by one. This gives $1*2^{\text{exponent}+1}$ or $2^{\text{exponent}+1}$, the negative of the original number. Case 2: The floating point number was originally zero. The two's-complement of zero is zero. Case 3: The floating point number was originally positive. The mantissa was, therefore 1 and the floating point number was $1*2^{\text{exponent}}$. When this number is two's-complemented, the MANT field is still zero but the EXP field is two's complemented. The mantissa becomes -2.0 and the carry from the fraction has decreased the exponent by one. (It increased the EXP but decreased the one's-complement of the EXP). This gives $-(2.0)*2^{\text{exponent}-1}$ or -2^{exponent}, the negative of the original number.)

Here are a few examples of the floating point format for halfwords:

Halfword 10.0
SIGN=0
EXP=-(-3)+16=19=23_8
MANT=(hidden 1)010 000 000 000_2=2000_8
Result: 232 000_8

Halfword -10.0
Two'sComplement(232 000_8)=546 000_8

Halfword 3.1415
SIGN=0
EXP=-(-1)+16=17=21_8
MANT=(hidden 1)100 100 100 010_2=4442_8
Result: 214 442_8

2.2.2 Integrity of Floating Point Arithmetic

The architecture specifies that floating point arithmetic will be performed so that the following equalities hold for all floating point values A and B:

A+0.0=A
A+B=B+A
-(-A)=A
A+(-B)=A-B=-(B-A)
A*1.0=A
A*B=B*A
A*0.0=0.0 unless A is NAN
-(A*B)=(-A)*B

2.2.3 Floating Point Exception Values

Besides zero, five floating point numbers have special meanings. The positive floating point number with the greatest magnitude (in a given precision) is called *OVF* (overflow). The two's-complement of OVF is called *MOVF* (minus overflow). The smallest positive floating point

number is called *UNF* (underflow). The largest negative floating point number is called *MUNF* (minus underflow). The floating point number with the sign bit set to 1 and all other bits set to 0 is called *NAN* (not a number); all floating point instructions consider it illegal.

OVF, MOVF, UNF, MUNF, and NAN correspond to side effects or exceptions that occur during floating point arithmetic. One happy consequence of the floating point format is that each of the special floating point values has the same bit representation as an easily recognizable integer, as the following table shows:

Name	Meaning	Integer with identical bit representation
OVF	Positive overflow	MAXNUM
MOVF	Negative overflow	MINNUM + 1 (i.e., -MAXNUM)
UNF	Positive infinitesimal	+1
MUNF	Negative infinitesimal	-1
NAN	Indeterminate ("not a number")	MINNUM

The range of values representable in the three floating point precisions is approximately the following:

Precision	Underflow	Overflow	Digits
Halfword	$1.53 * 10^{-5}$	$6.55 * 10^{4}$	3.91
Singleword	$8.63 * 10^{-78}$	$1.16 * 10^{77}$	8.13
Doubleword	$8.41 * 10^{-4933}$	$1.19 * 10^{4932}$	17.16

2.2.4 Comparing Floating Point Values

Another happy consequence of the floating point format is the ability to compare floating point numbers as if they were signed integers, without decoding the format. Thus, the architecture does not need a separate set of test and branch instructions for floating point numbers (although it does provide such instructions to enhance performance).

Integer comparisons will treat the floating point exception values in an intuitively reasonable fashion, too. For example, they will treat MUNF as greater than any other negative value but less than zero. The only exception is NAN, which will be treated not as an illegal value but as a value that is less than any other floating point value.

Every integer which is not identical with a floating point special symbol is identical with a legal floating point value. For example, the following table expresses in octal the integers corresponding to certain halfword floating point values:

OVF	377777 octal
Maximum legal positive value	377776
Minimum legal positive value	000002
UNF	000001
Zero	000000
MUNF	777777
Maximum legal negative value	777776
Minimum legal negative value	400002
MOVF	400001
NAN	400000

2.2.5 Floating Point Rounding Modes

During floating point operations, rounding of the result may be necessary. The FIX instruction includes a modifier that specifies how it rounds; all other floating point instructions which round their results do so according to the field *FLT_RND_MODE* in the *USER_STATUS* register. Instructions RRNDMD.FLT and WRNDMD.FLT (Section 2.2) read and write that field.

Let F be the magnitude of the difference between a true floating point result, R, and the greatest representable floating point number N which is less than or equal to R, expressed as a fraction of the least-significant representable bit of R. The bits of FLT_RND_MODE have the following functions (reversals of rounding direction accumulate):

Bit	Value	Effect
0	0	Round as specified by FLT_RND_MODE<1:4>
	1	Reserved.
1	0	If F <> 0, round as specified by FLT_RND_MODE<2:4> else deliver R exactly.
	1	If F = 1/2 then round as specified by FLT_RND_MODE<2:4> else round to the floating point number nearest to R.
2	0	Round toward negative infinity.
	1	Round toward positive infinity.
3	0	No effect.
	1	If the least significant bit of the mantissa of N is one, reverse the rounding direction.
4	0	No effect.
	1	If and only if R is negative, reverse the rounding direction.

Various combinations of the above bits provide a variety of rounding modes. Some of the more common modes are:

FLT_RND_MODE (octal)	Function
0	Floor
1	Diminished magnitude
4	Ceiling
5	Augmented magnitude
12	Stable
14	Half rounds toward positive infinity (PDP-10 FIXR)
15	Approximate PDP-10 FLTR rounding

Inexact rounding: Certain instructions exhibit inexact rounding--that is, the uncertainty in their rounding behavior slightly exceeds the uncertainty specified for floating point computations in general. The list of instructions which exhibit this characteristic is implementation dependent.

2.2.6 Floating Point Exception Handling

In the USER_STATUS register, four bits record "side effects" or exceptions by floating point arithmetic operations:

FLT_OVFL Floating point overflow (that is, the result of the instruction is greater than or equal to OVF or less than or equal to MOVF).

FLT_UNFL Floating point underflow (that is, the result of the instruction is less than or equal to UNF and greater than or equal to MUNF, but not equal to zero).

FLT_NAN Floating point result is "not a number" (NAN).

These bits are "sticky"--that is, floating point instructions may set them but not clear them, so once a bit is set it will remain set until explicitly cleared via manipulation of USER_STATUS.

In the following example, the first instruction sets FLT_OVFL, the second sets FLT_UNFL, and the third sets FLT_NAN:

```
FSUBV.H RTA,#0,#[400001]        ; OP2 is MOVF to begin with
FSC.H RTA,#[010000],#-1         ; Result too small to represent
FDIV.H RTA,#0                   ; Division by 0 is undefined
```

In addition to these exception bits, USER_STATUS contains fields called FLT_OVFL_MODE, FLT_UNFL_MODE, and FLT_NAN_MODE which tell the processor how to react to FLT_OVFL, FLT_UNFL, and FLT_NAN exceptions respectively. (Note that setting an exception bit by manipulating USER_STATUS will not invoke the specified behavior; the bit must be set during floating point arithmetic):

FLT_OVFL_MODE<0:1>

0	Invoke FLT_OVFL_TRAP soft trap without storing a result.
1	If the result was positive, use OVF as the result; if it was negative, use MOVF as the result.
2	Retain the sign and mantissa but replace the EXP field with a wrapped-around exponent.
3	Undefined. Attempting to set this value in the user status register causes an ILLEGAL_USER_STATUS hard trap.

FLT_UNFL_MODE<0:1>

0	Invoke FLT_UNFL_TRAP soft trap without storing a result.
1	If the result was positive, use UNF as the result; if it was negative, use MUNF as the result.
2	Retain the mantissa and sign of the result, but replace the EXP field with a wrapped-around exponent.
3	Use floating point 0.0 as the result.

FLT_NAN_MODE

0	Invoke FLT_NAN_TRAP soft trap without storing a result.
1	Use NAN as the result.
2, 3	Undefined. Attempting to set these values in the user status register causes an ILLEGAL_USER_STATUS hard trap.

2.2.7 Propagating Floating Point Exceptions

If either operand of a floating point instruction is one of the exception values, the instruction propagates the exceptional condition according to a precisely defined algorithm.

The tables in this section describe the standard propagation algorithm for all operations. (The algorithm is implemented in tables in RAM within the S-1 processor, so a front end processor could dictate a different algorithm if desired.)

In the tables, X and Y are assumed to be "ordinary" positive floating point numbers--that is, greater than UNF and less than OVF--which do not in themselves invoke exceptions.

Unary operations

A ↓	FNEG(A)	FABS(A)	FIX(A)	FTRANS(A)
MOVF	OVF	OVF	INT_OVFL	MOVF
MUNF	UNF	UNF	0	MUNF
UNF	MUNF	UNF	0	UNF
OVF	MOVF	OVF	INT_OVFL	OVF
NAN	NAN	NAN	INT_OVFL	NAN

Addition (A+B)

A ↓ B→	MOVF	-Y	MUNF	Ø	UNF	Y	OVF	NAN
MOVF	MOVF	MOVF	MOVF	MOVF	MOVF	MOVF	NAN	NAN
-X	MOVF	-X-Y	-X	-X	-X	-X+Y	OVF	NAN
MUNF	MOVF	-Y	MUNF	MUNF	Ø	Y	OVF	NAN
Ø	MOVF	-Y	MUNF	Ø	UNF	Y	OVF	NAN
UNF	MOVF	-Y	Ø	UNF	UNF	Y	OVF	NAN
X	MOVF	X-Y	X	X	X	X+Y	OVF	NAN
OVF	NAN	OVF	OVF	OVF	OVF	OVF	OVF	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Multiplication (A*B)

A ↓ B→	MOVF	-Y	MUNF	Ø	UNF	Y	OVF	NAN
MOVF	OVF	OVF	NAN	Ø	NAN	MOVF	MOVF	NAN
-X	OVF	X*Y	UNF	Ø	MUNF	-X*Y	MOVF	NAN
MUNF	NAN	UNF	UNF	Ø	MUNF	MUNF	NAN	NAN
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	NAN
UNF	NAN	MUNF	MUNF	Ø	UNF	UNF	NAN	NAN
X	MOVF	-X*Y	MUNF	Ø	UNF	X*Y	OVF	NAN
OVF	MOVF	MOVF	NAN	Ø	NAN	OVF	OVF	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Division (A/B)

A ↓ B→	MOVF	-Y	MUNF	Ø	UNF	Y	OVF	NAN
MOVF	NAN	OVF	OVF	NAN	MOVF	MOVF	NAN	NAN
-X	UNF	X/Y	OVF	NAN	MOVF	-X/Y	MUNF	NAN
MUNF	UNF	UNF	NAN	NAN	NAN	MUNF	MUNF	NAN
Ø	Ø	Ø	Ø	NAN	Ø	Ø	Ø	NAN
UNF	MUNF	MUNF	NAN	NAN	NAN	UNF	UNF	NAN
X	MUNF	-X/Y	MOVF	NAN	OVF	X/Y	UNF	NAN
OVF	NAN	MOVF	MOVF	NAN	OVF	OVF	NAN	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

The rules for the remaining instructions are simple enough to state without using additional tables:

FSUB The algorithm behaves as if the processor applied FNEG to the second argument and then performed FADD.

FMAX, FMIN If either argument is NAN, the result is NAN. Otherwise, the algorithm considers MOVF<-X<MUNF<0<UNF<X<OVF for any unexceptional positive number X.

FSC The exponentiation portion of the instruction FSC or FSCV is effectively done in infinite precision and will not produce an exception; the subsequent multiplication follows the rules given in the tables.

2.2.8 Floating point exceptions during vector instructions

A vector floating point instruction affects these bits in USER_STATUS as would a loop which applied the corresponding scalar floating point instruction to the elements of the vector(s) in order.

2.2.9 Floating Point Arithmetic

FADD — Floating point add

FADD . {H,S,D} **TOP**
VFADD . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FADD computes DEST:=S1+S2. VFADD adds each element of OP1 to the corresponding element of OP2 and stores the result either back into OP1 or into the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For the scalar instruction, S1, S2, and DEST all have the precision specified by the modifier. For the vector instruction, each element of each vector has the precision specified by the modifier.

The first instruction adds 1.0 to RTA. The second instruction doubles RTA; alternatively, FMULT, FSC, or FDIV might be used:

```
FADD.S RTA,#[1.0]
FADD.S RTA,RTA          ;RTA:=2.0*RTA; FSC RTA,#1 is preferable
```

FSUB — Floating point subtract

FSUB . {H,S,D}	**TOP**
FSUBV . {H,S,D}	**TOP**
VFSUB . {SR,OP1} . {H,S,D}	**V:=VV**
VFSUBV . {SR,OP1} . {H,S,D}	**V:=VV**

Purpose: FSUB calculates DEST:=S1-S2. FSUBV calculates S2-S1. VFSUB and VFSUBV are the analogous vector instructions; they both put the result either back into OP1 or into the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For the scalar instructions, S1, S2, and DEST all have the precision specified by the modifier. For the vector instructions, each element of each vector has the precision specified by the modifier.

The following subtracts a floating point value of one from RTA:

```
FSUB.S RTA,#[1.0]                ;RTA:=RTA-1.0
```

FMULT — Floating point multiply

FMULT . {H,S,D} **TOP**
VFMULT . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FMULT computes DEST:=S1*S2. VFMULT multiplies each element of OP1 by the corresponding element of OP2 and stores the result either back into OP1 or into the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For the scalar instruction, S1, S2, and DEST all have the precision specified by the modifier. For the vector instruction, each element of each vector has the precision specified by the modifier.

The following instruction doubles the value in RTA. Alternatively, FSC, FADD, or FDIV might be used:

```
FMULT.S RTA,#[2.0]        ;RTA:=RTA*2.0
```

FMULTL — Floating point multiply, long result

FMULTL . {H,S}

TOP

Purpose: DEST:=S1*S2. Note that the long result format will have more than twice as many mantissa bits as either operand.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. (These can occur only if one of the operands was a floating point exception value to begin with. The operation of multiplication itself cannot overflow or underflow because DEST has such a large exponent field.)

Precision: S1 and S2 have the same precision as the modifier. DEST has precision *twice* that of the modifier and must align accordingly.

The following instruction will place in RTA all significant bits of the square of X:

```
FMULTL.S RTA,X,X          ;RTA:=X**2
```

FDIV — Floating point divide

FDIV . {H,S,D}	**TOP**
FDIVV . {H,S,D}	**TOP**
VFDIV . {SR,OP1} . {H,S,D}	**V:=VV**
VFDIVV . {SR,OP1} . {H,S,D}	**V:=VV**

Purpose: FDIV computes the floating point quotient, S1 divided by S2, and stores it in DEST.

FDIVV swaps the roles of S1 and S2.

VFDIV divides each element of the vector beginning with OP1 by the corresponding element of the vector beginning with OP2 and stores the results either in the vector pointed to by SR0 (if the modifier is SR) or back into the vector beginning with OP1 (if the modifier is OP1).

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For FDIV and FDIVV, S1, S2, and DEST all have the precision specified by the modifier. For VFDIV, the elements of all three vectors have the precision specified by the modifier.

The following instruction doubles the value in RTA. Alternatively, FADD, FMULT or FSC might be used:

```
FDIV.S RTA,#[0.5]        ;RTA:=RTA/0.5=2.0*RTA
```

FRECIP — Floating point reciprocal

FRECIP . {H,S,D} **XOP**
VFRECIP . {H,S,D} **XOP**

Purpose: OP1 := 1.0 / OP2. In most implementations, FRECIP offers higher performance than FDIV but inexact rounding.

VFRECIP is a vector version of FRECIP. Assuming that "i" increments by the precision of the modifier, they compute:

```
FOR i := 0 TO SIZEREG-1 DO
    OP1[i] := 1.0 / OP2[i]
```

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: OP1 and OP2 have the same precision as the modifier.

The following instruction reciprocates 2.0:

```
FRECIP.S RTA,#2.0       ; RTA := 0.5
```

FDIVL

Floating point divide, long dividend

FDIVL . {H,S} **TOP**
FDIVLV . {H,S} **TOP**

Purpose: FDIVL divides S1 by S2 in floating point and stores the result in DEST.

FDIVLV, the reverse form, divides S2 by S1 instead.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For FDIVL, S2 and DEST have the precision of the modifier. S1 has precision *twice* that of the modifier and must align accordingly.

For FDIVLV, S1 and DEST have the precision of the modifier and S2 has twice that precision

The following uses a doubleword 1.0 to reciprocate a singleword in RTA. Note that this is NOT the same constant that would be used for FDIV:

```
FDIVL.S RTA,#[200000,,0 ? !0],RTA        ; RTA:=1.0 (DW) / RTA
```

FSC — Floating point scale

FSC . {H,S,D} **TOP**
FSCV . {H,S,D} **TOP**

Purpose: DEST := S1 * 2^{S2}. S1 is a floating point number and S2 is a signed integer.

FSCV computes the floating point number S2 * 2^{S1}, where S2 is a floating point number and S1 is a signed integer.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. (FLT_OVFL and FLT_UNFL are not set during the exponentiation, which is done with unlimited precision.)

Precision: For FSC, S1 and DEST have the same precision as the modifier and S2 is a singleword. For FSCV, S2 and DEST have the precision of the modifier and S1 is a singleword.

The following instruction may be used to double the value in RTA. Alternatively, FADD, FMULT, or FDIV might be used:

```
FSC.S RTA,#1     ;RTA:=RTA*2**(1)=2.Ø*RTA
```

FIX — Convert floating point to fixed (integer)

FIX . {Q,H,S,D} . {H,S,D} **XOP**
VFIX . {H,S,D} . {H,S,D} **V:=V**

Purpose: FIX converts the floating point number specified by OP2 into an integer and stores it in OP1. VFIX converts each element of the vector beginning with OP2 to an integer and stores the result in the corresponding element of the vector beginning with OP1.

These instructions use the rounding mode given in the INT_RND_MODE field of the USER_STATUS register, and then reset it to diminished-magnitude rounding.

For VFIX, if the two vectors have equal precision, they may overlap. If the precision of the source vector exceeds that of the destination, the two vectors may be identical but must not otherwise overlap. If the precision of the destination vector exceeds that of the source, the two vectors must not overlap at all. Violating these rules produces undefined results.

Restrictions: None

Exceptions: INT_OVFL

Precision: For FIX, OP1 has the precision of the second modifier and OP2 has the precision of the third modifier. For VFIX, the elements of OP1 have the precision of the first modifier and the elements of OP2 have the precision of the second.

The following converts a floating point value in RTA into an integer using floor rounding.

```
WRNDMD.INT #0
FIX.US.S.S RTA,RTA
```

FLOAT — Convert to floating point

FLOAT . {H,S,D} . {Q,H,S,D} **XOP**
VFLOAT . {H,S,D} . {Q,H,S,D} **V:=VS**

Purpose: FLOAT converts the integer OP2 into a floating point number in OP1:

```
OP1 := FLOAT(OP2);
```

VFLOAT converts each element of vector OP2 into a floating point number, multiplies it by a floating point scaling factor in RTA, and stores the result in the corresponding element of OP1:

```
FOR I := 0 TO SIZEREG DO
    OP1[I] := RTA * FLOAT(OP2[I]);
```

If the two vectors have the same precision, they may overlap. If the precision of the source vector exceeds that of the destination vector, the two vectors may be identical but may not otherwise overlap. If the precision of the destination vector exceeds that of the source, the vectors must not overlap. Violating these rules produces undefined results.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. (Provided you use the scalar instruction or set RTA=1.0 when using the vector instruction, the only possible exception is FLT_OVFL, and it can occur only on FLOAT.H.S and FLOAT.H.D; for all other conversions, the floating point format can express the corresponding integer with--at worst--only the loss of the least significant bits.)

Precision: OP1 has the precision of the first modifier. OP2 and RTA have the precision of the second modifier.

The following converts an integer to a floating point number without scaling:

```
FTRANS.S.S RTA,#[1.0]
FLOAT.S.S FLOATING,INT
```

The following converts a halfword integer to a halfword floating point number, scaling it so that overflow cannot occur:

```
FTRANS.H.H RTA,#[0.125]
FLOAT.H.H FLOATING,INT
```

FTRANS — Floating point translate

FTRANS . {H,S,D} . {H,S,D} **XOP**
VFTRANS . {H,S,D} .{H,S,D} **V:=V**

Purpose: FTRANS copies a floating point number from OP2 to OP1, converting its precision if necessary.

VFTRANS performs FTRANS on individual elements of vector OP1 and stores the result in vector OP2. If the source and destination vectors have the same precision, the vectors may overlap; the instruction guarantees not to alter any element of the source until it has copied that element to the destination.

If the source vector's precision exceeds that of the destination vector, the two vectors may be identical, but must not otherwise overlap.

If the source vector's precision is less than that of the destination vector, the two vectors may not overlap at all.

In some implementations FTRANS.S.S will offer better performance than MOV.S.S when operating on floating point data because a series of floating point instructions permits the processor to maintain the data in an internal format that is easier to handle.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. If OP2 has no greater precision than OP1, then these can occur only if OP2 is one of the floating point exception values.

Precision: OP2 has the precision of the second modifier. OP1 has the precision of the first modifier.

The following illustrates the precision alteration possible with FTRANS. The exact values produced will, in general, depend on the rounding mode defined in USER_STATUS.FLT_RND_MODE:

```
FTRANS.S.D RTA,#[200000,,0 ? !0]        ; Funny constant is 1.0 DW
```

FSELECT — Floating point conditional move

FSELECT {RTA,RTB} . {H,S,D} TOP

Purpose: This instruction selects one of two values to be stored into the destination based on a flag in RTA or RTB. That is, it performs: IF {RTA,RTB} <> 0, then DEST := S2 else DEST := S1.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. These can occur only if OP2 is one of the floating point exception values.

Precision: OP1 and OP2 have the precision of the modifier; the RTx operand is a singleword.

The following shows to select one of two values for a variable depending on a condition.

```
CMPSF.LEQ.S RTA,I,J      ; IF I <= J ...
FSELECT.RTA.S.S RTA,R,S  ; ... THEN P := S ELSE P := R
FTRANS.S.S P,RTA
```

FNEG — Floating point negate

FNEG . {H,S,D} **XOP**
VFNEG . {H,S,D} **V:=V**

Purpose: FNEG negates the floating point number in OP2 and stores the result in OP1. VFNEG performs NEG on each element of the vector beginning at OP2 and stores the results in the vector beginning at OP1.

The difference between NEG and FNEG is that FNEG handles floating point exceptions.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: OP1 and OP2 have the same precision as the modifier.

These examples show how floating point exceptions are propagated by FNEG.

```
FNEG.H RTA,#-1          ;RTA:=MUNF, signal FLT_UNFL
FNEG.H RTA,#677777      ;RTA:=OVF, signal FLT_OVFL
FNEG.H RTA,#400000      ;RTA:=NAN, signal FLT_NAN
```

FABS — Floating point absolute value

FABS . {H,S,D} **XOP**
VFABS . {H,S,D} **V:=V**

Purpose: FABS takes the floating point absolute value of OP2 and stores it in OP1. In comparison with ABS, FABS handles floating point exceptions.

VFABS performs FABS on each element of the vector OP2 and stores the results in the vector OP1.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: OP1 and OP2 have the same precision as the modifier.

These examples show how the uses of FABS and ABS on floating point numbers differ.

```
ABS.H RTA,#[-1]          ;RTA:=-1, no side effects
FABS.H RTA,#[-1]         ;RTA:=MUNF, signal FLT_UNFL
ABS.H RTA,#[377777]      ;RTA:=MAXNUM, no side effects
FABS.H RTA,#[377777]     ;RTA:=OVF, signal FLT_OVFL
ABS.H RTA,#[-400000]     ;RTA:=NAN, signal INT_OVFL
FABS.H RTA,#[-400000]    ;RTA:=NAN, signal FLT_NAN
```

FMIN — Floating point minimum

FMIN . {H,S,D} **TOP**
VFMIN . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FMIN places in DEST the smaller of the floating point numbers S1 and S2. The primary difference between MIN and FMIN is that FMIN properly propagates the floating point exception values.

VFMIN performs FMIN on a series of pairs: an element of the vector beginning with OP1 and the corresponding element of the vector beginning with OP2. If the first modifier is OP1, the results go back into the elements of vector OP1; if it is SR, they go into the elements of the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For FMIN, S1, S2, and DEST all have the precision specified by the {H,S,D} modifier. For VFMIN, the elements of vector OP1, vector OP2, and the vector pointed to by SR0 all have the precision specified by the {H,S,D} modifier.

This instruction sets RTA to the smaller of X and 43.0:

```
FMIN.S RTA,X,#[43.0]
```

FMAX — Floating point maximum

FMAX . {H,S,D} **TOP**
VFMAX . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FMAX places in DEST the larger of the floating point numbers S1 and S2. The primary difference between MAX and FMAX is that FMAX properly propagates the floating point exception values.

VFMAX performs FMAX on a series of pairs: an element of the vector beginning with OP1 and the corresponding element of the vector beginning with OP2. If the first modifier is OP1, the results go back into the elements of vector OP1; if it is SR, they go into the elements of the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For FMAX, S1, S2, and DEST all have the precision specified by the {H,S,D} modifier. For VFMAX, the elements of vector OP1, vector OP2, and the vector pointed to by SR0 all have the precision specified by the {H,S,D} modifier.

This sequence of instructions takes the number F and "clips" it to be within the window of [0.0,1.0]:

```
FMAX.S RTA,F,#0.0       ;larger of F and 0.0 to RTA
FMIN.S F,RTA,#1.0       ;smaller of that and 1.0 to F
```

FSQR — Square

FSQR . {H,S,D} **XOP**
VFSQR . {H,S,D} **V:=V**

Purpose: Compute the square of a floating point number:

OP1 := OP2 * OP2

VFSQR performs FSQR on each element of vector OP2 and places the results in vector OP1.

Restrictions: None

Exceptions: FLT_OVL, FLT_UNFL, FLT_NAN

Precision: Both OP1 and OP2 have the precision specified by the modifier.

The following leaves the square root of 625.0 in RTA:

```
FSQR.S  RTA,#25.0       ; RTA := 625.0
```

FADDSUB — Floating point add and subtract

FADDSUB . {H,S,D}	**XOP**
FADDSUBV . {H,S,D}	**XOP**
VFADDSUB . {H,S,D}	**VV:=VV**
VFADDSUBV . {H,S,D}	**VV:=VV**

Purpose: FADDSUB computes the sum and difference of a pair of floating point numbers:

```
TEMP := OP1 + OP2;
OP2 := OP1 - OP2;
OP1 := TEMP;
```

FADDSUBV computes:

```
TEMP := OP1 + OP2;
OP2 := OP2 - OP1;
OP1 := TEMP;
```

VFADDSUB and VFADDSUBV perform the analogous operations on vectors.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN.

Precision: OP1, OP2, and each element of each vector have the precision specified by the modifier.

Compute the sum and difference of 4.0 and 5.0:

```
MOV.S.S RTA,4.0
MOV.S.S RTB,5.0
FADDSUB.S RTA,RTB        ; RTA:=9.0; RTB:=-1.0
```

2.3 Complex Arithmetic

Certain instructions operate on halfword or singleword complex numbers in either signed integer or floating point format. A complex number actually consists of two consecutive integers or floating point numbers; the one at the lower memory or register address is the real part and the one at the higher address is the imaginary part.

For scalar complex numbers, the real and imaginary parts must align to form a single entity of twice the precision. Thus, a halfword complex number occupies two halfwords or one singleword (and must align as a singleword).

For vectors, alignment is unnecessary. The first integer or floating point number in the vector is considered real, the second is considered imaginary, the third is considered real, and so on. SIZEREG must contain the number of complex entities in the vector, not the number of individual integers or floating point numbers in the vector. Thus, if the precision of a vector of complex numbers is "S" and SIZEREG=n, the vector contains $2n$ singlewords.

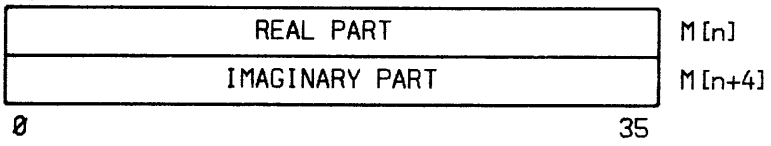


Figure 2-1
A singleword complex number

CADD — Complex add

CADD . {H,S} **TOP**
FCADD . {H,S} **TOP**

Purpose: CADD adds complex integers and FCADD adds complex floating point numbers:

```
FIRST(DEST):=FIRST(S1) + FIRST(S2); (* Real part *)
SECOND(DEST):=SECOND(S1) + SECOND(S2); (* Imaginary part *)
```

Restrictions: None

Exceptions: CARRY and INT_OVFL for the integer instruction; FLT_OVFL, FLT_UNFL and FLT_NAN for the floating point instruction.

Precision: DEST, S1, and S2 obey the precision and alignment rules for complex numbers.

The following leaves in RTA and RTA1 the sum of the complex numbers 4+i5 and 3+i12.:

```
CADD.S  RTA, [4 ? 5], [3 ? 12.]    ; RTA := 7; RTA1 := 17.
```

CSUB — Complex subtract

CSUB . {H,S} **TOP**
CSUBV . {H,S} **TOP**
FCSUB . {H,S} **TOP**
FCSUBV . {H,S} **TOP**

Purpose: CSUB subtracts complex integers and FCSUB subtracts complex floating point numbers:

```
FIRST(DEST):=FIRST(S1) - FIRST(S2); (* Real part *)
SECOND(DEST):=SECOND(S1) - SECOND(S2); (* Imaginary part *)
```

CSUBV and FCSUBV reverse the roles of S1 and S2.

Restrictions: None

Exceptions: CARRY and INT_OVFL for the integer instructions; FLT_OVFL, FLT_UNFL and FLT_NAN for the floating point instructions.

Precision: DEST, S1, and S2 obey the precision and alignment rules for complex numbers.

The following leaves in RTA and RTA1 the difference of the two complex numbers 4+i5 and 3+i12.:

```
CSUB.S  RTA, [4 ? 5], [3 ? 12.]    ; RTA := 1; RTA1 := -7
```

FCMULT — Complex multiply

FCMULT . {H,S} **TOP**
VFCMULT . {SR,OP1} . {H,S} **V:=VV**

Purpose: FCMULT multiplies complex floating point numbers:

```
FIRST(TEMP):=FIRST(S1) * FIRST(S2) -
     SECOND(S1) * SECOND(S2); (* Real part *)
SECOND(TEMP):=FIRST(S1) * SECOND(S2) +
     SECOND(S1) * FIRST(S2); (* Imaginary part *)
DEST:=TEMP;
```

VFCMULT multiplies the vector at OP1 with the vector at OP2, putting the results either back into OP1 or into to the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_NAN, FLT_OVFL, and FLT_UNFL

Precision: For the scalar instruction, DEST, S1, and S2 obey the precision and alignment rules for complex numbers; for the vector instruction, OP1 and OP2 obey the precision rules for complex vectors.

The following leaves in RTA and RTA1 the result of multiplying the complex numbers 4.0+i5.0 and 3.0+i12.0:

```
FCMULT.S RTA,[4.0 ? 5.0],[3.0 ? 12.0] ; RTA := -48.0; RTA1 := 63.0
```

FCDIV — Complex divide

FCDIV . {H,S}	**TOP**
FCDIVV . {H,S}	**TOP**
VFCDIV . {SR,OP1} . {H,S}	**V:=VV**
VFCDIVV . {SR,OP1} . {H,S}	**V:=VV**

Purpose: FCDIV divides complex floating point numbers (using additional precision internally to avoid overflow):

```
FIRST(TEMP) := (FIRST(S1) * FIRST(S2) + SECOND(S1) * SECOND(S2)) /
     (FIRST(S1) ** 2) + (SECOND(S2) ** 2);
SECOND(TEMP) := (SECOND(S1) * FIRST(S2) - FIRST(S1) * SECOND(S2)) /
     (FIRST(S1) ** 2) + (SECOND(S2) ** 2);
DEST := TEMP;
```

FCDIVV swaps the roles of SR1 and SR2.

VFCDIV divides the vector at OP1 by the vector at OP2, putting the result either back into OP1 or into the vector pointed to by SR0. VFCDIVV divides OP2 by OP1, putting the results either back into OP1 or into the vector pointed to by SR0.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: For the scalar instructions, DEST, S1, and S2 obey the precision and alignment rules for complex numbers. For the vector instructions, OP1 and OP2 obey the precision rules for complex vectors.

The following leaves in RTA and RTA1 the result of dividing the complex numbers 4.0+i5.0 and 3.0+i12.0:

```
FCDIV.S RTA,#[4.0 ? 5.0],[3.0 ? 12.0] ; RTA:=0.47; RTA1:=0.22
```

CNEG

Complex negate

CNEG . {H,S} **XOP**
FCNEG . {H,S} **XOP**

Purpose: CNEG negates a complex integer and FCNEG negates a complex floating point number:

FIRST(OP1):=*two's-complement*(FIRST(OP2));
SECOND(OP1):=*two's-complement*(SECOND(OP2));

Restrictions: None

Exceptions: CARRY, INT_OVFL for the integer instructions; FLT_OVFL, FLT_UNFL, FLT_NAN for the floating point instructions.

Precision: OP1 and OP2 obey the precision and alignment rules for complex numbers.

The following leaves in RTA and RTA1 the result of negating the complex number 4+i5:

```
CNEG.S  RTA, [4 ? 5]      ; RTA := -4; RTA1 := -5;
```

VCCONJ — Complex conjugate

VCCONJ . {H,S} **V:=V**
VFCCONJ . {H,S} **V:=V**

Purpose: VCCONJ computes the conjugate of a vector of complex integers while VFCCONJ computes the conjugate of a vector of complex floating point numbers:

```
FOR i := 0 TO SIZEREG-1 DO
    BEGIN
    FIRST(OP1[i]) := FIRST(OP2[i]);
    SECOND(OP1[i]) := two's complement(SECOND(OP2[i]));
    END;
```

Restrictions: None

Exceptions: CARRY, INT_OVFL for the integer instruction; FLT_OVFL, FLT_UNFL, FLT_NAN for the floating point instruction.

Precision: OP1 and OP2 obey the precision rules for complex vectors.

The following computes the complex conjugate of 25 complex integers; SOURCE is a vector of 50. singlewords, and so is RESULT:

```
MOV.S.S SIZEREG,#25.
VCCONJ.S RESULT,SOURCE
```

FCMAG — Complex magnitude

FCMAG . {H,S} **XOP**
VFCMAG . {H,S} **V:=V**

Purpose: This computes the magnitude of a floating point number:

OP1 := SquareRoot(FIRST(OP2) ** 2 + SECOND(OP2) ** 2)

VFCMAG is a vector version of CMAG. The destination vector OP1 contains half as many quarterwords as the source vector OP2, and thus SIZEREG (consistent with the rule for complex vector instructions) can be thought of as either the number of {halfwords, singlewords} in the destination or the number of complex entities in the source.

Restrictions: None

Exceptions: INT_OVFL for the integer instructions; FLT_NAN, FLT_OVFL, and FLT_UNFL for the complex instructions.

Precision: For the scalar instructions, OP1 has the precision of the modifier while OP2 obeys the precision and alignment rules for a complex entity of that precision. For the vector instructions, OP1 is a vector having the precision of the modifer and OP2 is a vector of complex entities of that precision.

The following finds the length of the hypotenuse of a right triangle whose sides have lengths of 3 and 4:

```
FCMAG.S RTA, [3,0? 4,0] ; RTA := 5.0
```

CMAGSQ — Complex magnitude squared

CMAGSQ.{H,S}	**XOP**
FCMAGSQ.{H,S}	**XOP**
VCMAGSQ.{H,S}	**V:=V**
VFCMAGSQ.{H,S}	**V:=V**

Purpose: Compute the square of the scalar magnitude of a complex number.

CMAGSQ regards the complex number as a pair of signed integers, while FCMAGSQ regards it as a pair of floating point numbers.:

```
OP1 := FIRST(OP2)**2 + SECOND(OP2)**2
```

VCMAGSQ and VFCMAGSQ are vector versions of CMAGSQ and FCMAGSQ. Assuming that "i" increments by the precision of the modifier, they compute:

```
FOR i := 0 TO SIZEREG-1 DO
    OP1[i] := FIRST(OP2[2*i])**2 + SECOND(OP2[2*i])**2
```

Restrictions: None

Exceptions: INT_OVFL (for CMAGSQ and VCMAGSQ); FLT_NAN, FLT_OVFL, and FLT_UNFL (for FCMAGSQ and VFCMAGSQ)

Precision: For CMAGSQ and FCMAGSQ, OP1, FIRST(OP2), and SECOND(OP2) have the precision specified by the modifier. FIRST(OP2) and SECOND(OP2) must align together to form an entity having twice that precision.

For VCMAGSQ and VFCMAGSQ, the elements of all three vectors have the precision specified by the modifier.

The following finds the sum of the squares of 3 and 4:

```
CMAGSQ.S        RTA, [3 ? 4]     ; RTA := 25
```

2.4 Mathematics

FSQRT — Square root

FSQRT . {H,S,D} **XOP**
VFSQRT . {H,S,D} **V:=V**

Purpose: Compute the principal square root in floating point: OP1:=SquareRoot(OP2).

VFSQRT performs FSQRT on each element of vector OP2 and places the results in vector OP1.

The implementation is guaranteed to be monotonic--that is, if x≥y then SQRT(x)≥SQRT(y). Attempting to take the square root of a negative number invokes FLT_NAN, which will result in either a FLT_NAN_TRAP soft trap or NAN, depending on the setting of USER_STATUS.

Restrictions: None

Exceptions: FLT_NAN

Precision: Both OP1 and OP2 have the precision specified by the modifier.

The following leaves the square root of 25 in RTA:

```
FSQRT.S RTA,#25.0       ; RTA := 5.0
```

FLOG

Floating point logarithm (base 2)

FLOG . {H,S,D} **XOP**
VFLOG . {H,S,D} **V:=V**

Purpose: FLOG computes the base 2 logarithm of OP2 and stores the result in OP1. The results are guaranteed to be monotonic--that is, if x≥y then FLOG(x)≥FLOG(y).

VFLOG performs FLOG on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. Taking the logarithm of a non-positive number invokes FLT_NAN, resulting in either NAN or a FLT_NAN_TRAP soft trap, depending on the setting of USER_STATUS.

Precision: OP1 and OP2 have the precision specified by the modifier.

The following leaves RTA set to the base 2 logarithm of 32:

```
FLOG.S  RTA,#32.0       ; RTA := 5.0
```

Using the rule that $\log_b z = \log_2 z / \log_2 b$, the following instructions compute the base 10 logarithm of 1000.0:

```
FLOG.S  RTB,#10.0       ; RTB := base 2 log of 10.0
FLOG.S  RTA,#1000.0     ; RTA := base 2 log of 1000.0
FDIV.S  RESULT,RTA,RTB  ; RESULT := 3.0
```

FEXP — Floating point exponential (base 2)

FEXP . {H,S,D} **XOP**
VFEXP . {H,S,D} **V:=V**

Purpose: Raise 2.0 to a power: FEXP computes OP1:=2.0**OP2. VFEXP performs FEXP on each element of OP2 and places the result in the corresponding element of OP1. The results are guaranteed to be monotonic--that is, if x≥y then FEXP(x)≥FEXP(y).

Restrictions: None

Exceptions: FLT_NAN, FLT_OVFL, FLT_UNFL

Precision: OP1 and OP2 have the precision specified by the modifier.

Using the rule that x**y = 2**(y * $\log_2$x), the following raises 81.0 to the power 0.25:

```
FLOG.S  RTA,#81.0
FMULT.S RTA,#0.25       ; RTA := 0.25 * FLOG(81.0)
FEXP.S  RTB,RTA         ; RTB := 3.0
```

FSIN — Floating point sine

FSIN . {H,S,D} **XOP**
VFSIN . {H,S,D} **V:=V**

Purpose: FSIN computes OP1:=Sine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

VFSIN performs FSIN on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_NAN

Precision: Both operands have the precision specified by the modifier.

The following computes the sine of an angle expressed in degrees:

```
MOV.S.S ANGLE,#30.0          ; 30 degrees
FDIV.S  RTA,ANGLE,#360.0     ; convert to cycles
FSIN.S  RTA                  ; RTA := 0.5
```

FCOS — Floating point cosine

FCOS . {H,S,D} **XOP**
VFCOS . {H,S,D} **V:=V**

Purpose: FCOS computes OP1:=Cosine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

VFCOS performs FCOS on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_NAN

Precision: Both operands have the precision specified by the modifier.

The following computes the cosine of an angle expressed in degrees:

```
MOV.S.S ANGLE,#60.0              ; 60 degrees
FDIV.S  RTA,ANGLE,#360.0         ; convert to cycles
FCOS.S  RTA                      ; RTA := 0.5
```

FSINCOS — Floating point sine and cosine

FSINCOS . {H,S,D} **XOP**

Purpose: Computes FIRST(OP1):=Cosine(OP2) and SECOND(OP1):=Sine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

Note that because the cosine appears in the first anyword of the pair and the sine in the second, the result can be used as a complex number.

Restrictions: None

Exceptions: FLT_NAN

Precision: FIRST(OP1), SECOND(OP1), and OP2 have the precision specified by the modifier. FIRST(OP1) and SECOND(OP1) must align together to form an entity having twice that precision.

The following computes both the sine and the cosine of an angle expressed in degrees;

```
MOV.S.S ANGLE,#60.0              ; 60 degrees
FDIV.S  RTA,ANGLE,#360.0         ; convert to cycles
FSINCOS.S RTA                    ; RTA := 0.866...; RTA1 := 0.5
```

FATAN — Floating point arctangent

FATAN . {H,S,D} **TOP**
FATANV . {H,S,D} **TOP**
VFATAN . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FATAN computes DEST:=Arctangent(S1/S2). Expressing the tangent as a quotient instead of a single value allows the instruction to determine the correct quadrant for the result, which lies between -0.5 and 0.5 inclusive. Multiplying the result by 2*PI yields a value in radians.

FATANV, the reverse form, swaps the roles of S1 and S2.

VFATAN performs FATAN on each pair of elements, one from vector OP1 and the other from vector OP2, and places the result in the corresponding element of either vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=Arctangent(OP1[i]/OP2[i])
    ELSE
        SR0@[i]:=Arctangent(OP1[i]/OP2[i])
```

Restrictions: None

Exceptions: FLT_NAN

Precision: All three operands have the precision specified by the {H,S,D} modifier.

The following computes an arctangent in degrees:

```
FATAN.S RTA,#1.0,#1.0
FMULT.S RTA,#360.0      ; RTA := 45.0 degrees
```

VF2DSQ, VF3DSQ

Vector 2- or 3-dimensional distance squared

VF2DSQ . {SR,OP1} . {H,S,D} **V:=VV**
VF3DSQ . {SR,OP1} . {H,S,D} **V:=VVV**

Purpose: Compute the sum of squares of a series of coordinate pairs or triples.

VF2DSQ deals with coordinate pairs, where the vector beginning with OP1 holds the first coordinate of each pair and the vector beginning with OP2 holds the second. Depending on the first modifier, these instructions put the result back in vector OP1 or in the vector pointed to by SR0.

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN OP1[i]:=OP1[i]**2 + OP2[i]**2
    ELSE SR0@[i]:=OP1[i]**2 + OP2[i]**2
```

VF3DSQ deals with coordinate triples, where the vector beginning with OP1 holds the first coordinate of each triple, the vector beginning with OP2 holds the second, and the vector pointed to by SR0 holds the third. Depending on the first modifier, these instructions put the result back in vector OP1 or in the vector pointed to by SR1.

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN OP1[i]:=OP1[i]**2 + OP2[i]**2 + SR0@[i]**2
    ELSE SR1@[i]:=OP1[i]**2 + OP2[i]**2 + SR0@[i]**2));
```

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, and FLT_NAN

Precision: Each element of each vector has the precision specified by the second modifier.

The following example illustrates the use of V2DSQ:

```
MOV.S.S SIZEREG,#3        ; Specify length of vectors
MOVP.P.A SR0,RESULT       ; Set up SR0 to point to result
VF2DSQ.SR.S [1.0 ? 2.0 ? 3.0],[4.0 ? 5.0 ? 6.0]
                          ; RESULT now holds [17.0 ? 29.0 ? 45.0]
```

VF2DIS, VF3DIS

Vector 2- or 3-dimensional distance

VF2DIS . {SR,OP1} . {H,S,D} **V:=VV**
VF3DIS . {SR,OP1} . {H,S,D} **V:=VVV**

Purpose: Compute the square root of the sum of squares for a series of coordinate pairs or triples.

VF2DIS operates on coordinate pairs, where the vector beginning with OP1 contains the first coordinate of each pair and the vector beginning with OP2 contains the second. Depending on the first modifier, the resulting vector goes back into OP1 or into the vector pointed to by SR0.

```
FOR i:=1 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=SquareRoot(OP1[i]**2 + OP2[i]**2)
    ELSE
        SR0@[i]:=SquareRoot(OP1[i]**2 + OP2[i]**2)
```

VF3DIS operates on triples, with the vector beginning at OP1 containing the first coordinate of each triple, the vector beginning at OP2 containing the second, and the vector pointed to by SR0 containing the third. Depending on the first modifier, the result goes back into the vector starting at OP1 or into the vector pointed to by SR1:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=SquareRoot(OP1[i]**2 + OP2[i]**2 + SR0@[i]**2)
    ELSE
        SR1@[i]:=SquareRoot(OP1[i]**2 + OP2[i]**2 + SR0@[i]**2)
```

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, and FLT_NAN

Precision: Each element of each vector has the precision specified by the second modifier.

Suppose X_DISP and Y_DISP represent a drawing as a series of line segments, describing each segment as a pair of displacements in the X and Y directions from the endpoint of the preceding segment. The following program fragment converts this data to represent each segment as an angle and magnitude:

```
        ; Obtain a vector of angles
        MOV.S.S SIZEREG,#0
NEXT:   FATAN.S RTA,X_DISP[SIZEREG]↑2Y_DISP[SIZEREG]↑2
        MOV.S.S ANGLE[SIZEREG]↑2,RTA
        ISKP.LSS SIZEREG,LENGTH,NEXT
```

```
                                        ; Now SIZEREG = length of vector
VF2DSQ.OP1.S X_DISP,Y_DISP              ; X_DISP becomes a vector
                                        ; of magnitudes
```

VFDOT — Dot product

VFDOT . {H,S,D} **S:=VV**

Purpose: Compute the dot product of two vectors:

```
RTA:=0;
FOR i:=0 TO SIZEREG-1 DO
    RTA:=RTA + OP1[i] * OP2[i]
```

To avoid overflow and underflow problems, the processor accumulates the sum with as much precision as it can, regardless of the {H,S,D} modifier. If that modifier is "H", the result goes into RTA as a singleword, and if the modifier is "S", RTA is a doubleword. If the modifier is "D", however, the result is still a doubleword.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, and FLT_NAN.

Precision: The elements of each vector have the precision specified by the modifier. RTA has twice that precision unless the modifier is D, in which case RTA is a doubleword.

Suppose that singleword vector V contains the results from sampling a voltage waveform at 100 Hz for one second. The following computes the RMS voltage:

```
MOV.S.S SIZEREG,#100.    ; Put length in SIZEREG
VFDOT.S V,V              ; Sum of squares
FDIV.D  RTA,#100.0       ; Mean
FSQRT.D RTA,RTA          ; Root
```

FCONV — Convolution

FCONV . {H,S,D} **V:=VV**

Purpose: "Sort of" convolve two vectors, OP1 and OP2. The vector starting at OP1 is assumed to be at least as long as the vector starting at OP2. SR1 is the length of the OP2 vector and SIZEREG is the length of the result vector. If N is the length of the OP1 vector, then the following holds: SIZEREG=N-SR1 + 1. The assumption that the OP1 vector is no shorter than the OP2 vector gives N≥SR1 ⇒ SIZEREG>0. To do a "real" convolution of two vectors, say A_i and B_i of lengths N_A and N_B ($N_A \geq N_B$) respectively, simply build a new vector A′ which is the original vector A with N_B zeros concatenated before and after A, and a new vector B′ which is a end-for-end reversed copy of B. Then do a FCONV with OP1=A′, OP2=B′, SR1=N_B, and with SIZEREG = $(N_B - 1 + N_A + N_B - 1) - N_B + 1 = N_A + N_B - 1$.

```
FOR i:=0 TO SIZEREG-1 DO
    BEGIN
    SR0@[i]:=0;
    FOR j:=0 TO SR1-1 DO
        SR0@[i]:=SR0@[i] + OP2[j] * OP1[i + j]
    END;
```

Restrictions: None.

Exceptions: FLT_OVFL, FLT_NAN

Precision: SR1 and the elements of each of the vectors have the precision specified by the modifier.

FRFLT2 — Second order recursive filter

FRFLT2 . {H,S,D} **V:=V**

Purpose: Apply a second order recursive filter to the vector whose first element is OP2 and leave the results in the vector whose first element is OP1. The instruction obtains the coefficients of the filter from the five element vector pointed to by SR0. The result is actually two elements shorter than SIZEREG indicates, since it begins at OP1[2] instead of OP1[0]. The user must initialize the first two elements of the OP1 vector to start the recursion properly.

```
FOR i:=0 TO SIZEREG - 3 DO
    OP1[i+2]:=SR0@[0] * OP1[i]
        + SR0@[1] * OP1[i+1]
        + SR0@[2] * OP2[i]
        + SR0@[3] * OP2[i+1]
        + SR0@[4] * OP2[i+2]
```

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, and FLT_NAN

Precision: The coefficients and the elements of each vector have the precision specified by the modifier.

The following example filters the signal in vector SENSE_IN:

```
MOVP.P.A SR0,COEFFICIENTS       ; Pointer to five coefficients
FTRANS.D.D RESULT,[1.73476 ? 1.73476]
                                ; Initialize the recursion
MOV.S.S SIZEREG,#1000.          ; Specify length of SENSE_IN
FRFLT2.S RESULT,SENSE_IN
```

INTRAN — In-place square matrix transpose

INTRAN . {H,S,D} **V:=V**

Purpose: Transpose a square two-dimensional matrix without moving the matrix to a different area of memory. (The TRANSP instruction can operate on a matrix which is not square, but must move the matrix to a new, non-overlapping area of memory as it does so.)

OP1 is the first element of the matrix, which must be stored in row major order (second subscript varying more rapidly than the first). R3 gives the number of rows (which is, of course, the same as the number of columns) in the matrix, and must be a multiple of 8 for halfword precision (or a multiple of 4 for singlewords, or a multiple of 2 for doublewords).

Restrictions: None

Exceptions: ILLEGAL_MATRIX_DIMENSION

Precision: Every element of the matrix has the precision specified by the modifier. R3 is a singleword.

To transpose the following matrix:

```
0  1  2  3
4  5  6  7
8  9  10 11
12 13 14 15
```

one could use the INTRAN instruction like this:

```
DSPACE
                ; Expressions separated by "?" assemble
                ; successive singlewords in memory
FOURBY:         0 ? 1 ? 2 ? 3 ? 4 ? 5 ? 6 ? 7 ? 8
                9 ? 10 ? 11 ? 12 ? 13 ? 14 ? 15

ISPACE
        INTRAN.S FOURBY,#4.
                ; Now FOURBY = 0 ? 4 ? 8 ? 12 ? 1 ? 5 ? 9 ? 13
                ;              ? 2 ? 6 ? 10 ? 14 ? 3 ? 7 ? 11 ? 15
```

TRANSP — Matrix transpose

TRANSP . {H,S,D} **V:=V**

Purpose: Transpose a two-dimensional matrix, moving it to a different, non-overlapping area of memory in the process. (The INTRAN instruction transposes a matrix without moving it, but requires that the matrix be square.)

The instruction expects the matrix to be stored in row major order with its first element at OP2. The result of the transposition appears in row major order with its first element at OP1.

Registers R0 and R1 respectively specify the number of rows and columns in the source matrix. Registers R2 and R3 specify the number of columns to ignore between each row in the source and destination matrices respectively. To transpose an entire matrix, one sets R2 and R3 to zero; to transpose a submatrix, one sets R2 and R3 to skip over the columns that lie outside the submatrix.

The number of rows (and the number of columns) in the source matrix must be a multiple of 8 for halfword precision (or 4 for singlewords, or 2 for doublewords.)

Restrictions: None

Exceptions: ILLEGAL_MATRIX_DIMENSION

Precision: All elements of the source and destination matrices have the precision specified by the modifier. R0, R1, R2, and R3 are singlewords.

To transpose the following matrix:

```
0  1  2  3
4  5  6  7
```

use the TRANSP instruction like this:

```
; Assume the matrix is stored as a series of doublewords
; in the following order: 0 1 2 3 4 5 6 7
ISPACE
        MOV.S.S R0,#2           ; Number of rows
        MOV.S.S R1,#4           ; Number of columns
        MOV.S.S R2,#0           ; Do not skip anything
        MOV.S.S R3,#0
        TRANSP.D NEWPLACE,TWOBY4
; The result is a series of doublewords in the following
; order: 0 4 1 5 2 6 3 7
```

As an example of how to use R2 and R3 to transpose a submatrix, suppose we have the following matrices (in Pascal notation):

```
VAR A: ARRAY [0..ARows, 0..ACols-1] OF INTEGER;
    B: ARRAY [0..BRows, 0..BCols-1] OF INTEGER;
```

and we want to transpose the submatrix of A whose origin is A[Ax,Ay] and whose size is SRows by SCols, storing the result in the submatrix of B whose origin is B[Bx,By]. Assuming the submatrices are proper (that is, they truly fit within A and B) we can use the following instructions:

```
MOV.S.S R0,#SRows       ; Number of rows in submatrix
MOV.S.S R1,#SCols       ; Number of columns in submatrix
MOV.S.S R2,#ACols
SUB.S R2,#SCols         ; Skip (ACols-SCols) columns between
                        ; source rows
MOV.S.S R3,#BCols
SUB.S R3,#SCols         ; Skip (BCols-SCols) columns between
                        ; dest rows

MOV.S RTA,AY
ARRIND.RTA AX,ACOLS
MOVP.P.A RTA,A[RTA]↑2

MOV.S RTB,BY
ARRIND.RTB BX,BCOLS
MOVP.P.A RTB,B[RTB]↑2
```

FMATMUL — Matrix multiply

FMATMUL . {H,S,D} **V:=VV**

Purpose: Multiply two 2-dimensional matrices stored in memory in row major order. OP1 is the first singleword of a 9-singleword block which describes the two source matrices and the destination matrix.

Word	Meaning
0	Number of rows in source matrix 1
1	Number of columns in source matrix 1
2	Number of columns in source matrix 2
3	Number of columns to skip between rows of source matrix 1
4	Number of columns to skip between rows of source matrix 2
5	Number of columns to skip between rows of destination matrix
6	Pointer to origin of source matrix 1
7	Pointer to origin of source matrix 2
8	Pointer to origin of destination matrix

The third, fourth, and fifth elements of the OP1 block are used when multiplying submatrices. To multiply entire matrices, one ordinarily sets these to zero.

Like VFDOT, FMATMUL accumulate results internally in the greatest feasible precision regardless of the precision of the result.

Restrictions: OP1 may not be a register or constant.

Exceptions: FLT_NAN, FLT_OVFL, and FLT_UNFL (for FMATMUL)

Precision: Every element of each matrix has the precision specified by the modifier. OP1 is the first element of a block of 9 singlewords.

The following example multiplies the two matrices shown and stores the result in matrix D:

```
A= 1.0  2.0  3.0        B= 1.0  2.0
   3.0  2.0  1.0           3.0  3.0
                           2.0  1.0

   MOV0S.S PBLOCK,#2              ; Rows in source matrix 1
   MOV.S.S PBLOCK+4,#3            ; Columns in source matrix 1
   MOV.S.S PBLOCK+8.,#2           ; Columns in source matrix 2
   MOV.S.S PBLOCK+12.,#0
   MOV.S.S PBLOCK+16.,#0
   MOV.S.S PBLOCK+20.,#0
```

```
MOVP.P.A PBLOCK+24.,A          ; Pointer to source matrix 1
MOVP.P.A PBLOCK+28.,B          ; Pointer to source matrix 2
MOVP.P.A PBLOCK+32.,D          ; Pointer to destination matrix
FMATMUL.S PBLOCK
```

As an example of how to multiply submatrices, assume we have the following matrices (in Pascal notation):

```
VAR A: ARRAY [0..ARows-1, 0..ACols-1] OF REAL;
    B: ARRAY [0..BRows-1, 0..BCols-1] OF REAL;
    D: ARRAY [0..DRows-1, 0..DCols-1] OF REAL;
```

and that we want to multiply the submatrix whose origin is at A[Ax,Ay] with the submatrix whose origin is at B[Bx,By], storing the result in D[Dx,Dy]. The submatrix of A has R rows by S columns and the submatrix of B has S rows by T columns. Assuming further that the submatrices are proper (that is, they fit inside the corresponding matrices), we can use the following code:

```
MOV.S.S DESC,R
MOV.S.S DESC+4*1,S
MOV.S.S DESC+4*2,T
SUB RTA,ACols,S
MOV DESC+4*3,RTA
SUB RTA,BCols,T
MOV DESC+4*4,RTA
SUB RTA,DCols,T
MOV DESC+4*5,RTA

MOV RTA,Ay
ARRIND.RTA ACols,Ax
MOVP.P.A DESC+4*6,A[RTA]↑2

MOV RTA,By
ARRIND.RTA BCols,Bx
MOVP.P.A DESC+4*7,B[RTA]↑2

MOV RTA,Dy
ARRIND.RTA DCols,Dx
MOVP.P.A DESC+4*8,D[RTA]↑2

FMATMUL.S DESC
```

FFT **In-place complex FFT and inverse FFT**

CFFT . {H,S}	**V:=V**
FCFFT . {H,S}	**V:=V**
CFFTV . {H,S}	**V:=V**
FCFFTV . {H,S}	**V:=V**

Purpose: Compute the fast Fourier transform (FFT) or inverse fast Fourier transform of a vector of complex numbers.

CFFT and FCFFT compute the FFT, with CFFT operating on complex signed integers and FCFFT on complex floating point numbers.

CFFTV and FCFFTV compute the inverse FFT, with CFFTV operating on complex signed integers and FCFFTV on complex floating point numbers.

For all four instructions, OP1 designates the first element of the vector to be transformed. In each case, the instruction puts its results back into the original source vector. The number of elements in the vector must be a power of 2; R3 contains that power (i. e., the base 2 logarithm of the number of elements). If R3 is not positive, the instruction leaves the vector untouched.

If the source vector exceeds the maximum allowable length, an FFT_TOO_LONG soft trap occurs. (This limit is implementation-dependent; see Section 4.2.) If desired, one can provide a software trap handler that operates transparently to the user on vectors of arbitrary size, transforming a lengthy vector by repeatedly applying the instruction to subvectors.

The last step of the FFT algorithm is a "scrambling" operation which swaps elements of the vector whose indices within the vector are bit reversals of each other. (For example, in a 16-element vector where indices range from 0 to 15, this scrambling would swap element 12 with element 3 because reversing the bits of the four-bit binary representation of 12 yields 3. Similarly, the scrambling would swap element 1 with element 8, and so on.) Because this step represents a considerable fraction of the time required for the total FFT, the architecture does not incorporate it in the FFT instructions themselves, but provides a separate instruction called BADREV to perform it.

Similarly, "scrambling" is the first step of the complete inverse FFT algorithm, but it is omitted from the inverse FFT instructions, which expect their source arrays to be scrambled.

Thus, a complete FFT would require the CFFT instruction (for example) followed by the BADREV instruction. A complete inverse FFT would require the BADREV instruction followed by (for example) the CFFTV instruction.

Providing a separate instruction for swapping elements saves time in many applications where one wants to transform a signal, operate on it, and transform it back. Because the FFT instructions produce a scrambled result and the inverse FFT instructions expect a scrambled input, one can

simply omit to unscramble and rescramble between them--provided the operations that take place between the FFT and inverse FFT instructions preserve the scrambled order.

Restrictions: None

Exceptions: INT_OVFL, (for CFFT and CFFTV); FLT_OVFL, FLT_UNFL, and FLT_NAN (for FCFFT and FCFFTV), FFT_TOO_LONG

Precision: Every element of the vector has the precision specified by the modifier. R3 is a singleword.

Consider a simple filtering operation where one transforms the input signal, multiplies it by a vector of selected filter coefficients, and transforms it back. One could write:

```
MOVP.P.A R0,OUTPUT
CFFT.S INPUT,LOGSIZE
BADREV.D INPUT,LOGSIZE
MOV.S.S SIZEREG,SIZE
V"XY".SR.S INPUT,COEFFIC
BADREV.D OUTPUT,LOGSIZE
CFFTV.S OUTPUT,LOGSIZE

BADREV.D COEFFIC,LOGSIZE

MOVP.P.A R0,OUTPUT
CFFT.S INPUT,LOGSIZE
V"XY".SR.S INPUT,COEFFIC
CFFTV.S OUTPUT,LOGSIZE
```

But by scrambling the coefficient vector itself (an operation which need be performed only once no matter how many signals are to be passed through the same filter),

```
BADREV.D COEFFIC
```

one can remove both BADREV operations from the preceding sequence:

```
MOV.S.S R3,LOGSIZE      ;Define number of elements in vector
MOVP.P.A RTA,COEFFIC    ; Point to scrambled coefficients
CFFT.S INPUT            ; FFT
V"SX".S OUTPUT,INPUT    ; Filter using scrambled coefficients
CFFTV.S OUTPUT          ; Inverse FFT
```

The following example uses the FCFFT, BADREV, and INTRAN instructions together to perform a two-dimensional FFT:

```
;2DFFT - Two dimensional complex FFT
; halfword floating point
;Transform complex 2D array whose origin is in ORG
;Size of array is 2**LOGSIZE by 2**LOGSIZE
;
; Called via JSR PC,2DFFT
;
2DFFT:  MOV.S.S R3,LOGSIZE       ;Define number of elements in vector
        SHF.LF.S RTA,#1,LOGSIZE  ;Get number of rows (and columns)
        MOV.S.S ESIZE,RTA        ;Save number of elements in rows and columns
        SHF.LF.S SIZE,RTA,#2     ;Convert to halfword complex size and save
        MOVP.P.P T,ORG           ;Initialize row pointer to first row
        MOV.S.S RTA,ESIZE        ;Loop counter
2dfft1: FCFFT.H (T)              ;Transform a row
        BADREV.S (T)             ;Un-bit-reverse this row
        ADD.S T,SIZE             ;Step to next row
        DJMPZ.GTR RTA,2dfft1     ;Last row?
        MOV.S.S R3,ESIZE
        INTRAN.S (ORG),ESIZE     ;Transpose array
        MOVP.P.P T,ORG
        MOV.S.S RTA,ESIZE
        MOV.S.S R3,LOGSIZE
2dfft2: FCFFT.H (T)              ;Transform a column
        BADREV.S (T)             ;Un-bit-reverse this column
        ADD.S T,SIZE             ;Step to next column
        DJMPZ.GTR RTA,2dfft2     ;Last column?
        MOV.S.S R3,ESIZE
        INTRAN.S (ORG),ESIZE     ;Transpose array back
        RETSR PC,(SP)            ;Return
```

BADREV — In-place bit address reversal

BADREV . {S,D} **V:=V**

Purpose: Within a vector, swap each pair of elements whose addresses represent bit-reversals of each other. The instruction is primarily useful in conjunction with the FFT and inverse FFT instructions.

The last step of the FFT algorithm is a "scrambling" operation which swaps elements of the vector whose indices within the vector are bit reversals of each other. (For example, in a 16-element vector where indices range from 0 to 15, this scrambling would swap element 12 with element 3 because reversing the bits of the four-bit binary representation of 12 yields 3. Similarly, the scrambling would swap element 1 with element 8, and so on.)

OP1 is the first element of the vector to be scrambled; the instruction puts the results back into the same vector. The number of elements in the vector must be a power of 2. R3 specifies that power (or, in other words, the base 2 logarithm of the number of elements). If R3 is not positive, the instruction leaves the vector untouched.

Restrictions: None

Exceptions: None

Precision: The elements of the vector all have the precision specified by the modifier. R3 is a singleword.

Note that when one uses BADREV to complete an FFT operation, the precision must be twice that of the FFT instruction because the vector in question contains complex numbers and thus each data point comprises two values:

```
CFFT.S SIGNAL,LOGSIZE
BADREV.D SIGNAL,LOGSIZE
```

QPART — Quicksort partition inner loop

QPART V:=V

Purpose: Pipelined processors must predict with considerable accuracy whether conditional branch instructions will alter the flow of control, or execution speed suffers. Because sorting algorithms usually contain unpredictable conditional branches, the architecture provides an instruction to perform the inner loop of the Quicksort algorithm, eliminating branches.

OP1 is a pointer to the first element of a vector of records and OP2 is a pointer to the last record in the vector. Each record consists of a singleword key followed by a singleword of data (typically a pointer to a larger amount of data).

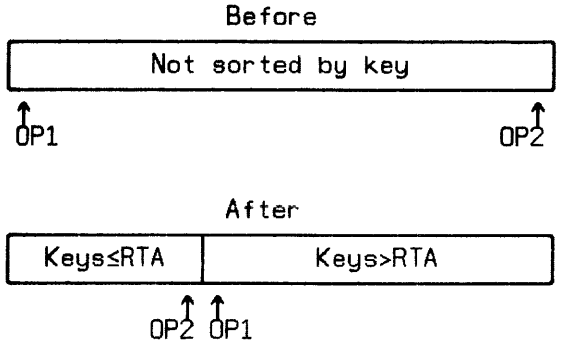
RTA contains a partitioning value.

The instruction rearranges the elements of the vector, segregating them into two groups so that all the records in one group have keys exceeding RTA and all the records in the other have keys less than or equal to RTA. Within each group, the records may still be disordered (though in moving records about to achieve the segregation, the instruction does attempt to order them locally); the instruction guarantees merely to partition the vector into two groups relative to the value in RTA.

When the instruction finishes, the first part of the vector contains the group of records with keys less than or equal to RTA (the "lower partition"), and OP2 points to the last record in that group. OP1 points to the next record, which is the first record in the group whose keys exceed RTA (the "upper partition"). RTA contains a code that reports the status of the two partitions:

0	The lower partition is sorted, but the upper one is not.
1	The upper partition is sorted, but the lower one is not.
2	Both partitions need sorting. The upper has fewer records.
3	Both partitions need sorting. The lower has fewer records.
4	Both partitions are sorted.

In simplified form, the instruction does the following:



Restrictions: None

Exceptions: None

Precision: Each element of the vector is a pair of singlewords, the first serving as a key and the second as data which the instruction moves along with the key. RTA is a singleword.

The following example illustrates how to use QPART to implement the complete quicksort algorithm:

```
; Quicksort
; Called via: JSR #-1,QUICKSORT
;  On entry :
;   LOW - pointer to first record of array to be sorted
;   HIGH - pointer to last record of array to be sorted
;    (HIGH must immediately follow LOW)
;  On exit :
;   Array between LOW and HIGH is completely sorted
QUICKSORT:
        MOVP.P.A SP,(SP)10              ;Reserve space to save HIGH and LOW
                                        ; later on
QUICK1: SUB.S RTB,HIGH,LOW              ;Calculate size of array - 8
        SHFA.RT.S RTB,#4                ;Get half the size
                                        ; (in doublewords)
        SEXCH.D (LOW),(HIGH)            ;Swap the first, last, and middle
        SEXCH.D (LOW)0[RTB]↑3,(HIGH)    ; words of the array as necessary
        SEXCH.D (LOW),(LOW)0[RTB]↑3     ;  so first<middle<low
        MOV.S.S RTA,(LOW)0[RTB]↑3       ;Partition array around middle's value
        MOV.D.D (SP)-10,LOW             ;Save high and low pointers
        QPART LOW,HIGH                  ;Do the partitioning
        JMPA QUICK2[RTA]↑3              ;Dispatch to correct routine

QUICK2:         ;Dispatch table.
                ;It is important that all sections (except the last)
                ; be two words long

;Sort upper half only => tail recursion
        MOV.S.S HIGH,(SP)-4
        JMPA QUICK1

;Sort lower half only => tail recursion
        MOV.S.S LOW,(SP)-10
        JMPA QUICK1

;Sort upper first then lower => full recursion
```

```
        EXCH.S HIGH,(SP)-4
        JMPA QUICKSORT

;Sort lower then upper => full recursion
        EXCH.S LOW,(SP)-10
        JMPA QUICKSORT

;All sorted
        MOVP.P.A SP,(SP)-10     ;Discard the HIGH and LOW just saved
        MOV.D.D LOW,(SP)-10     ;Restore previous HIGH and LOW
        ;If LOW is the -1 value pushed by the JSR that invoked the quicksort,
        ; we're finished, so return to the caller. Otherwise, tail recursion
        ; continues sorting.
        JMPZ.GEQ.S LOW,QUICK1   ;Tail recursion
        RET (SP)-4,(SP)         ;Discard -1 and return to original caller
```

VBADD — Vector bignum addition

VBADD . {SR,OP1} **V:=VV**

Purpose: VBADD allows the efficient implementation of bignum addition. Bignums are implemented as a vector of doublewords, which represents a two's-complement integer 72*SIZEREG bits long. This instruction adds the bignums starting at OP1 and OP2. The initial carry-in is taken from the CARRY flag in the user status the the carry-out of the addition if left in the CARRY flag at the end of the operation. The actual sequence of doubleword additions is done starting at the end of the OP1 and OP2 vector, since the additions are most convenient to do starting with the least significant bits.

A more precise description is:

```
VAR
 CARRY: 0..1; (* From USER_STATUS *)
 OP1,
 OP2,
 DEST: ARRAY[0..SIZEREG-1] OF 0..2**72-1;
 TMP: 0..2**73-1

BEGIN
 FOR I := SIZEREG-1 DOWNTO 1
  DO BEGIN
    TMP := A[I] + B[I] + CARRY;
    DEST[I] := TMP MOD 2**72;
    CARRY := TMP DIV 2**72;
    END;
END;
```

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are vectors of doublewords.

The following sequence illustrates the use of VBADD to add two bignums:

```
A:      BLOCK 100*10            ;addend
B:      BLOCK 100*10            ;addend
AB:     BLOCK 200*10            ;sum
SR0=%4*0
SIZEREG=%4*3
```

```
BIGADD: ADD SIZEREG,?0          ;clear carry flag (any register will do)
        MOV.S.S SIZEREG,#100    ;load SIZEREG
        MOVP.P.A SR0,AB
        VBADD A,B
BIGAD2: ;; 100*72. BIT SUM IN AB
```

VBMULT — Vector bignum multiply

VBMULT V:=VV

Purpose: VBMULT allows the efficient implementation of bignum multiply. Bignums are implemented as a vector of doublewords, which represents a twos-complement integer 72*SIZEREG bits long. This instruction multplies the bignum starting at OP2 by the 72 bit number in RTA and accumulates this intermediate product in the bignum starting at OP1. The doubleword carry out is returned in RTA. It has the significance of the doubleword before the beginning of OP2.

Sign correction information necessary to allow multiplying two signed bignums is taken from RTB. The sign bit of RTB is the sign correction for RTA, and the sign bit of RTB1 (i.e. R7) is the sign correction for OP2.

A more precise description is:

```
VAR
  A: -2**71..2**71;
  TMP: -2**143..2**143-1;
  CIN: -2**71..2**71;
  OP1: ARRAY[0..SIZEREG-1] OF 0..2**72-1;
  OP2: ARRAY[0..SIZEREG-1] OF 0..2**72-1;(* Except that OP2[0] is really
                                            signed, i.e. -2**71..2**71-1  *)
BEGIN
  A := RTA:RTA1;                          (* Double word multiplier *)
  IF RTB < 0 THEN A := A+1;               (* Where A is large enough that
                                             this can't overflow *)
                                          (* This is for sign correction *)
  IF RTB1 < 0 THEN CIN := A               (* Sign correction for OP2, the
              ELSE CIN := 0;                 multiplicand *)
  FOR I := SIZEREG-1 DOWNTO 1
    DO BEGIN
      TMP := A * OP2[I]                   (* Signed * Unsigned *)
                + CIN
                + OP1[I];
      OP1[I] := TMP MOD 2**72;
      CIN := TMP DIV 2**72;
      END;
  TMP := A * OP2[0]                       (* Signed * Signed *)
                + CIN
                + OP1[0];
  OP1[0] := TMP MOD 2**72;
  RTA:RTA1 := TMP DIV 2**72;
END;
```

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are vectors of doublewords. RTA is a doubleword. RTB and RTB1 are singlewords.

The following sequence illustrates the use of VBMULT to multiply two bignums:

```
A:       BLOCK 100*10              ;multiplicand
B:       BLOCK 100*10              ;multiplier
AB:      BLOCK 200*10              ;product
SIZEREG=%4*3
I=%4*8.
J=%4*9.

BIGMUL: MOV.S.S SIZEREG,#100      ;load SIZEREG
        MOV I,#100-1
        MOV J,#100
        VINI.D AB+100*10,#0       ;initialize the least significant half of
        MOV.D.D AB-10[J]↑3,#0     ;result plus one extra word
        MOV.D.D RTB,#0            ;clear sign bits of RTB & RTB1

BIGMU1: MOV.D.D RTA,B[I]↑3        ;get multiplier
        VBMULT AB[J]↑3,A          ;do most of the work
        ADD.D AB-10[J]↑3,RTA      ;save the carry out in next "digit"
        DJMPZ.LEQ J,BIGMU2        ;that was the last "digit"
          JMPZ.D.GEQ AB[J]↑3      ;sign extend result if negative
            MOV.D.D AB-10[J]↑3,?-1
          MOV.S.S RTB,B[I]↑3      ;copy sign correction for next iteration
          DJMPA I,BIGMU1          ;get index of next multiplier "digit"

BIGMU2: ;; 200*72. BIT PRODUCT IN AB
```

VBNEG — Vector bignum negation

VBNEG V:=V

Purpose: VBNEG allows the efficient implementation of bignum negation. Bignums are implemented as a vector of doublewords, which represents a twos-complement integer 72*SIZEREG bits long. This instruction negates the bignum starting at OP2 and puts the result into OP1. The initial carry-in is taken from the CARRY flag in the user status the the carry-out of the negation if left in the CARRY flag at the end of the operation. The actual sequence of doubleword negations is done starting at the end of the OP1 and OP2 vector, since the negation is most convenient to do starting with the least significant bits.

A more precise description is:

```
VAR
 CARRY: 0..1;    (* From USER_STATUS *)
 OP1,
 OP2: ARRAY[0..SIZEREG-1] OF 0..2**72-1;
 TMP: 0..2**73-1

BEGIN
 FOR I := SIZEREG-1 DOWNTO 1
  DO BEGIN
    TMP := -OP2[I] - 1 + CARRY;
    OP1[I] := TMP MOD 2**72;
    CARRY := TMP DIV 2**72;
    END;
END;
```

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are vectors of doublewords.

2.5 Chained Vectors

These instructions perform arithmetic on vectors, often combining two or more operations. This results in faster execution not only because it reduces the number of instructions the processor must fetch--a single multiply-and-add instruction can take the place of a multiplication followed by an addition, for example--but also because the processor can use its adder and multiplier in parallel.

Because the mnemonics for chained vector instructions explain themselves, and because the arithmetic operations are logical extensions of those for scalars, this section will not describe each instruction in detail.

Each mnemonic consists of a V followed by up to two letters defining the data type and then an equation within quotation marks:

```
V<data type>"<equation>"
```

For <data type>, a "CF" indicates complex floating point, a "C" alone indicates complex signed integer, and "F" alone indicates floating point. If <data type> is missing, the instruction deals with signed integers.

Within the equation, "X", "Y", and "Z" are the first, second, and third source vectors while "S" and "R" are the first and second source scalars. As in algebra, concatenating variables indicates multiplication.

Thus, for example, the instruction:

```
VF"X+SY".OP1
```

performs the operation:

```
FOR i:=0 TO SIZEREG-1 DO
        OP1[i]:=OP1[i] + RTA * OP2[i]
```

Two Vector Operands and One Scalar

S+X, S-X, SX

V"S+X" . {H,S,D} **V:=VS**
VF"S+X" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA + OP2[i]

V"S-X" . {H,S,D} **V:=VS**
VF"S-X" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA - OP2[i]

VF"SX" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA * OP2[i]

Three Vector Operands and One Scalar

X+SY, SX+Y, SY-X, SX-Y, SX+SY, SX-SY, S+XY, S-XY

VF"X+SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] + RTA * OP2[i]
    ELSE SR0e[i]:=OP1[i] + RTA * OP2[i]
```

VF"SX+Y" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA * OP1[i] + OP2[i]
    ELSE SR0e[i]:=RTA * OP1[i] + OP2[i]
```

VF"SY-X" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA * OP2[i] - OP1[i]
    ELSE SR0e[i]:=RTA * OP2[i] - OP1[i]
```

VF"SX-Y" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA * OP1[i] - OP2[i]
    ELSE SR0e[i]:=RTA * OP1[i] - OP2[i]
```

VF"SX+SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA * (OP1[i] + OP2[i])
    ELSE SR0e[i]:=RTA * (OP1[i] + OP2[i])
```

VF"SX-SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA * (OP1[i] - OP2[i])
    ELSE SR0e[i]:=RTA * (OP1[i] - OP2[i])
```

VF"S+XY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA + (OP1[i] * OP2[i])
    ELSE SR0e[i]:=RTA + (OP1[i] * OP2[i])
```

VF"S-XY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=RTA - (OP1[i] * OP2[i])
    ELSE SR0e[i]:=RTA - (OP1[i] * OP2[i])
```

Two Vector Operands and Two Scalars

S+RX

VF"S+RX" . {H,S,D} **V:=VSS**
VFC"S+RX" . {H,S} **V:=VSS**

FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA + RTB * OP2[i]

Four Vector Operands

X+YZ

VF"X+YZ" . {SR,OP1} . {H,S,D} V:=VVV

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] + OP2[i] * SR0e[i]
    ELSE SR1e[i]:=OP1[i] + OP2[i] * SR0e[i]
```

2.6 Data Moving

MOV — Logical move

MOV . {Q,H,S,D} . {Q,H,S,D} — XOP

Purpose: OP1:=OP2. If OP2 has greater precision than OP1, the low-order bits of OP2 are used. If OP2 has smaller precision than OP1, it is zero-extended to the left. This is best thought of as a "logical" or "unsigned" move operation. No condition bits (e.g., carry or integer overflow) are affected. Note that the TRANS instruction can be used to perform sign-extended or truncated integer moves.

It is preferable to use FTRANS rather than MOV on floating point numbers, because the former will execute faster on most implementations.

Restrictions: None

Exceptions: None

Precision: The two modifiers specify the precisions of OP1 and OP2 respectively.

The following copies the low-order QW of RTA into the high-order QW:

```
MOV.Q.Q RTA,RTA+3
```

The next example shows how MOV extends an integer with zeroes rather than sign bits:

```
MOV.H.Q RTB,#-1          ; RTB := 000777 octal
TRANS.H.Q RTB,#-1        ; RTB := 777777 octal
```

SELECT

Move conditionally

SELECT {RTA,RTB} . {Q,H,S,D,P} **TOP**

Purpose: This instruction selects one of two values to be stored into the destination based on a flag in singleword RTA or RTB. That is, it performs: IF {RTA,RTB} <> 0, then DEST := S2 else DEST := S1.

If the modifier for OP2 is "P", a pointer move is performed according to the description of MOVP.P.P; otherwise, a logical move is performed. To move floating point numbers, it is preferable to use FSELECT because it will be faster on most implementations.

Restrictions: None

Exceptions: None

Precision: The two modifiers specify the precisions of OP1 and OP2 respectively; the RTA or RTB operand is a singleword.

The following shows how to conditionally alter the value of a variable.

```
CMPSF.GEQ.S RTA,B,C     ; IF B < C ...
SELECT.RTA.S.S A,#3     ; ... THEN A := 3
```

MOVMQ — Move many quarterwords

MOVMQ . { 2 .. 32, 64} **XOP**

Purpose: Moves a block of quarterwords beginning with OP2 into the block of quarterwords beginning with OP1, so that OP1:=OP2, NEXT(OP1):=NEXT(OP2), and so on. The modifier specifies how many quarterwords to move. If the source and destination regions overlap, the result is undefined. Unlike vector instructions, MOVMQ can access the registers. Constant arguments are not permitted.

Restrictions: None

Exceptions: None

Precision: This instruction deals with quarterwords for both source and destination precisions.

The following copies the three high-order QWs from RTA into RTB:

```
MOVMQ.3 RTB,RTA
```

MOVMS — Move many singlewords

MOVMS . { 2 .. 32 } **XOP**

Purpose: Moves a block of singlewords beginning with OP2 into the block of singlewords beginning with OP1, so that OP1:=OP2, NEXT(OP1):=NEXT(OP2), and so on. The modifier specifies how many singlewords to move. If the source and destination regions overlap, the result is undefined. Unlike vector instructions, MOVMS can access the registers. If OP2 is a constant its value will be spread through the specified range of locations.

Restrictions: None

Exceptions: None

Precision: This instruction deals with singlewords for both source and destination precisions.

The following saves all the registers from RTA onward in a block starting at SAVEBK:

```
MOVMS.28 SAVEBK,RTA
```

The following clears the registers:

```
MOVMS.32 R0,#0
```

VINI — Vector initialize

VINI . {Q,H,S,D} V:=S

Purpose: Initialize each element of a vector OP1 to match the scalar OP2.

Restrictions: None

Exceptions: None

Precision: The elements of the vector OP1, like the scalar OP2, have the precision specified by the modifier.

The following stores in each element of A the value in R0:

```
VINI.S A,RØ
```

VREV — Vector reverse

VREV . {Q,H,S,D} **V:=V**

Purpose: Reverse a vector end-for-end by swapping the first element with the last, the second element with the next-to-last, and so on. OP2 is the first element of the source vector and OP1 is the first element of the destination. Either OP1 and OP2 must be identical or the two vectors must not overlap at all; otherwise, the result of the instruction is undefined.

Restrictions: None

Exceptions: None

Precision: The elements of the two vectors have the precision specified by the modifier.

The following stores in DOWN the reverse of the vector in UP:

```
MOV.S.S SIZEREG,#5
VTRANS.S.S UP,[1 ? 2 ? 3 ? 4 ? 5]
VREV.S  DOWN,UP ; DOWN := 5, 4, 3, 2, 1
```

EXCH — Exchange

EXCH . {Q,H,S,D} **XOP**
VEXCH . {Q,H,S,D} **V:=V**

Purpose: EXCH exchanges OP1 with OP2; VEXCH exchanges vector OP1 with vector OP2.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 each have the precision specified by the modifier.

The following swaps RTA and RTB:

```
EXCH.S RTA,RTB
```

One can contrive a situation where the result depends on two rules: the processor prefetches operands, and XOP instructions store OP1 after storing OP2:

```
MOV.S.S RTA,#5
MOV.S.S RTA1,#6
MOV.S.S RTB,#7
EXCH.D RTA,RTA1 ; RTA:=6; RTA1:=7; RTB:=6
                ; (first RTA1:=5 and RTB:=6; then
                ; RTA:=6 and RTA1:=7)
```

SEXCH, USEXCH — Signed and unsigned sorted exchange

SEXCH . {Q,H,S,D} **XOP**
USEXCH . {Q,H,S,D} **XOP**

Purpose: If OP1 > OP2 then exchange OP1 with OP2. The instruction requires read and write access to both OP1 and OP2 even if the inequality is false and no exchange takes place. SEXCH treats the operands as signed integers, whereas USEXCH treats them as unsigned integers.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 each have the precision specified by the modifier.

The following swaps RTA and RTB only if RTA > RTB:

```
SEXCH.S RTA,RTB
```

SLR **Save and load register**

SLR . { 0 .. 124 step 4 } **XOP**

Purpose: Loosely speaking, the instruction saves the contents of the register specified by the modifier in OP1 and then loads that register with OP2.

More precisely, note that the processor prefetches operands and that XOP instructions store into OP1 last. Thus SLR effectively does the following:

```
TEMP1:=Rn
TEMP2:=OP2
Rn:=TEMP2
OP1:=TEMP1
```

As illustrated below, one can contrive situations where this behavior makes a difference.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords. The modifier must be a multiple of 4 within the range 0 . . 124.

The first instruction moves RTA into RTB and zeros RTA. The second and third instructions show what happens when one of the operands is the register specified in the instruction. The fourth shows what happens when the operands are the same.

```
SLR.RTA RTB,#0  ;RTB:=RTA, RTA:=0
SLR.RTA RTA,F   ;essentially a NOP
                ; (TMPR:=REG; TMP2:=OP2; REG:=TMP2; OP1:=TMPR)
                ; (TMPR:=RTA; TMP2:=F; RTA:=TMP2; RTA:=TMPR)

SLR.RTA F,RTA   ;effectively MOV F,RTA
                ; (TMPR:=RTA; TMP2:=RTA; RTA:=TMP2; F:=TEMPR)

SLR.RTA F,F     ;effectively EXCH RTA,F
                ; (TMPR:=RTA; TMP2:=F; RTA:=TMP2; F:=TMPR)
```

SLRADR — Save and load register with address

SLRADR . { 0 .. 124 step 4 } **XOP**

Purpose: Loosely speaking, the instruction saves in OP1 the register specified by the modifier and then loads the register with ADDRESS(OP2) in the same manner as MOVP.P.A does.

Because the processor prefetches operands, and because XOP instructions store into OP1 last, it is more precise to say that:

```
TEMP1:=Rn
MOVP.P.A Rn,OP2
OP1:=TEMP1
```

As illustrated below, one can concoct examples where this behavior makes a difference.

Restrictions: OP2 must not be a register or constant.

Exceptions: None

Precision: All operands involved are singlewords. The modifier must be a multiple of 4 in the range 0 .. 124

The first instruction moves RTA into RTB and puts ADDRESS(F) in RTA. The second shows what happens when the first operand is the register specified in the instruction. The third shows what happens when the operands are the same.

```
SLRADR.RTA RTB,F        ;RTB:=RTA, RTA:=ADDRESS(F)
SLRADR.RTA RTA,F        ;effectively a NOP
                        ; (TMP:=REG; REG:=ADDRESS(OP2); OP1:=TMP)
                        ; (TMP:=RTA; RTA:=ADDRESS(F); RTA:=TMP)

SLRADR.RTA F,F          ;same as MOV F,RTA; MOVP.P.A RTA,F
```

ARRIND — Array index

ARRIND {AL.AR} . {RTA,RTB} — XOP

Purpose: These instructions are used to accumulate array indices into one of RTA or RTB. With the "AL" modifier, the computation performed is: RTx := (OP1 * RTx) + OP2. With the "AR" modifier, RTx := (OP1 * OP2) + RTx is performed.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

Given the following program fragment:

```
VAR
   I, J, K: INTEGER;
   TABLE: ARRAY [0..2, 0..4, 0..6] OF INTEGER;
BEGIN
  TABLE[I,J,K]:=25;
```

Either form of the instruction could be used to calculate the subscript in the assignment statement. Using the left associative form:

```
MOV.S.S RTA,I                    ; x := i
ARRIND.AL.RTA #3,J               ; x := (i*3) + j
ARRIND.AL.RTA #5,K               ; x := ((i*3) + j)*5) + k
MOV.S.S TABLE[RTA]↑2,#25.        ; TABLE[x] := 25
```

Using the right associative form:

```
MOV.S.S RTA,I                    ; x := i
ARRIND.AR.RTA #15,J              ; x := (i*15) + j
ARRIND.AR.RTA #5,K               ; x := (i*15) + (j*5) + k
MOV.S.S TABLE[RTA]↑2,#25.        ; TABLE[x] := 25
```

MOVP — Move pointer

MOVP . {P,R} . {P,A} **XOP**

Purpose: Move pointer, optionally transforming it.

This instruction transforms and moves pointers, performing checks to validate the integity of the pointers. It verifies that ring-tagged pointers do not lie in a ring less privileged than they address, and that the pointers do not have fault tags, which normally indicate that the pointer is uninitialized. Self-relative pointers are converted to ring-tagged pointers as they are moved.

The modifier for OP2 specifies how to obtain the pointer value to store, and the modifier for OP1 specifies the format in which the pointer is to be stored.

If the OP2 modifier is "P", then the value is the operand itself. If the pointer has a ring tag, it is validated according to the rules of Section 1.8.3, possibly invoking the BAD_POINTER_TAG hard trap. If the pointer has a user, NIL or gate tag, then the pointer and tag are preserved. If the pointer has a self-relative tag, then it is converted to a ring-tagged pointer having the address of the referenced location. Note that this must be in the same segment as OP2, or an OUT_OF_BOUNDS hard trap will occur. A self-relative pointer is illegal if it appears in a register. Any other tags cause a BAD_POINTER_TAG hard trap.

If the OP2 modifier is "A", then the value is the virtual address of the operand.

If the OP1 modifier is "P", then the value derived from OP2 is stored.

If the OP1 modifier is "R", then the value derived from OP2 is stored as a self-relative pointer. Note that this must be an address in the same segment and ring as OP1. The operand must not be a register or a ILLEGAL_REGISTER_OPERAND hard trap occurs.

Thus there are four cases:

MOVP.P.P Move one pointer value to another. This may be used to convert a self-relative pointer to non-self-relative format.

MOVP.P.A Store the address of OP2 in a pointer.

MOVP.R.P Convert a pointer to a self-relative pointer.

MOVP.R.A Store the address of OP2 in a self-relative pointer.

Restrictions: None

Exceptions: BAD_POINTER_TAG, OUT_OF_BOUNDS, ILLEGAL_REGISTER_OPERAND

Precision: Operands corresponding to an "A" modifier have no alignment requirements; in all other cases, the operands are singlewords

The following makes register R0 point to location DATA:

```
MOVP.P.A R0,DATA
```

SETTAG — Set pointer tag

SETTAG — TOP

Purpose: Set DEST<0:4> to S2<0:4>, and DEST<5:35> to S1<5:35>. The effect is to manufacture a pointer using the tag of S1 and the address of S2.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

The following example changes the tag of the pointer, P, to a NIL tag.

```
SETTAG P,#040000000000
```

VALIDP — Validate pointer

VALIDP — XOP

Purpose: Validate the pointer OP1 with respect to the ring containing OP2. The address for OP2 is computed following the usual address validation rules, but OP2 itself is not actually fetched. Then, OP1 is validated using the validation level of OP2 instead of that of OP1. If the tag of OP1 is a ring tag and the number of the ring is less than the validation level of OP2, a BAD_POINTER_TAG trap occurs; if the tag of OP1 is a fault, gate or NIL tag, a BAD_POINTER_TAG trap also occurs.

Sections 1.8.2 and 1.8.3 describe the address and pointer validation mechanisms.

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

Suppose a process executing in ring 3 has called a routine executing in ring 1, passing it a parameter in register R27. The routine in ring 1 could use the return address saved on the stack--which by definition specifies the caller's ring of execution--to assure that the pointer in R27 is trustworthy. That return address is within the stack entry pushed by CALLX during the gate crossing (Section 1.9.5) at (SP)-6*4:

```
VALIDP R27,(SP)-6*4
```

BASEPTR — Base pointer

BASEPTR — XOP

Purpose: Store in OP1 a pointer to the beginning of the segment containing OP2. (The instruction stores ADDRESS(OP2) in OP1 and then sets to zero the low order LOGSEGSIZE+LOGPGSIZE bits of OP1, where LOGSEGSIZE is the base 2 logarithm of the number of pages in the segment and LOGPGSIZE is the base 2 logarithm of the number of quarterwords in a page; see 1.7 which describes paging and segmentation.)

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

Make BP point to the beginning of the segment containing the following instruction:

```
        BASEPTR BP, .
```

SEGSIZE — Segment size

SEGSIZE — XOP

Purpose: Store in OP1 the size in quarterwords of the segment containing OP2. The value stored is rounded up to an even multiple of segmentito sizes, e.g. 16*1024*4 quarterwords.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword; OP2 requires only quarterword alignment.

Assign the size of the segment containing the PC to S.

```
        SEGSIZE S, .
```

RMW **Read/modify/write**

RMW **TOP**

Purpose: In one memory cycle (and hence indivisibly with respect to other processors in a multiprocessor system) DEST:=S1 and then S1:=S2. (More precisely, because the processor prefetches operands and because TOP instructions store DEST last, this instruction makes a temporary copy of S1, stores S2 in S1, and then stores the copy into DEST.)

Other atomic instructions are MOVCSF and MOVCSS.

Restrictions: None

Exceptions: None

Precision: S1, S2, and DEST are all singlewords.

The following illustrates the use of RMW to implement a test-and-set lock for interprocessor communication. The lock is a singleword flag which is -1 if some processor has seized the lock and 0 if the lock is free:

```
SEIZE:  RMW RTA,LOCK,#-1        ;attempt to seize lock
        JMPZ.NEQ.S RTA,SEIZE    ;busy-wait if someone else has it
        ...                     ;do ... if lock was zero (now I have it)
FREE:   MOV.S.S LOCK,#0         ;release the lock
```

MOVPHY — Move physical address

MOVPHY **XOP**

Purpose: OP1:=PHYSICAL_ADDRESS(OP2).

Restrictions: Illegal in user mode.

Exceptions: If accessing the source operand of MOVPHY through the virtual address translation process would cause an addressing trap, that trap will occur even though MOVPHY deals with physical addresses.

Precision: OP1 and OP2 are singlewords; OP2 must not be a constant or register.

The following loads RTA with the *physical* address of F:

```
MOVPHY RTA,F     ;RTA:=PHYSICAL_ADDRESS(F)
```

RPHYS, WPHYS — Read/write physically addressed location

RPHYS **XOP**
WPHYS **XOP**

Purpose: RPHYS reads into OP1 the singleword contents of a memory location whose physical address is specified by the 34 low order bits of R3. WPHYS writes OP1 into a memory location whose address is specified by the 34 low order bits of R3.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. R3 is a singleword whose 34 low order bits are a physical address. OP2 is unused.

The following moves SOURCE to DESTINATION even if the mapping tables are changed following the first two instructions:

```
MOVPHY R3,SOURCE
MOVPHY R2,DESTINATION
...
RPHYS  RTA
EXCH.S R3,R2
WPHYS  RTA
```

MOVMEM — Move to/from memory, bypassing cache

MOVFMEM . {N,C} . {1,16} **XOP**
MOVTMEM . {N,C} . {1,16} **XOP**

Purpose: These implementation-dependent instructions exist for use by memory diagnostics.

MOVFMEM reads a block of words directly from memory beginning with OP2, bypassing the cache, and writes them to a block beginning with OP1, using the cache. MOVTMEM reads from OP2, using the cache, and writes directly to OP1, bypassing the cache.

If the first modifier is N (for "no correction"), the instruction copies each singleword along with its associated error-correction bits into a doubleword, right-justified with leading zeros, instead of applying the error correction algorithm. If the first modifier is C ("correction"), the instruction copies source singlewords into destination singlewords, applying the correction algorithm and then discarding the error-correction bits.

The second modifier specifies the length of the source block in singlewords. Note that it offers a choice of 1 or 16, not the subrange 1 . . 16.

Restrictions: Illegal in user mode. Neither operand may be a constant. The operand which bypasses the cache may never be a register, and the operand which uses the cache may not be a register if the second modifier is 16.

Exceptions: None

Precision: For MOVFMEM, OP2 is the first element of a block of {1,16} singlewords, and must be aligned to a {1,16} singleword boundary. If the first modifier is "N", OP1 is the first element of a block of {1,16} doublewords; otherwise, OP1 is the first element of a block of {1,16} singlewords. For MOVTMEM, the precision requirements are reversed.

The following example copies a block of 16 singlewords into a block of 16 doublewords, revealing the error-correction bits:

```
MOVFMEM.N.16 DEST,SOURCE
```

2.7 Skip, Jump, and Comparison

Skip and jump instructions branch to locations other than that of the next sequential instruction. Skip instructions branch within a short range while jumps branch anywhere in the 2^{31} quarterword address space.

Many skips or jumps occur only if a condition specified by a modifier to the instruction is true. An *arithmetic condition* (ACOND) can be any of the following :

ACOND = {GTR,EQL,GEQ,LSS,NEQ,LEQ}

These correspond to the conditions >, =, >=, <, <>, <= respectively.

The SKP instruction may use a *logical condition* (LCOND) as well. The LCONDs are:

LCOND = {NON,ALL,ANY,NAL}

These correspond to the logical conditions that relate two operands (say OP1 and OP2) as shown in the table below. Here OP2 is considered to be a mask whose "1" bits select bits of OP1 to be tested.

Modifier	Condition	Meaning
NON	(OP1 ∧ OP2) = 0	If no masked bits are 1
ALL	(*one's-complement*(OP1) ∧ OP2) = 0	If all masked bits are 1
ANY	(OP1 ∧ OP2) <> 0	If any masked bit is 1
NAL	(*one's-complement*(OP1) ∧ OP2) <> 0	If not all masked bits are 1

Combining the ACONDs and the LCONDs gives the arithmetic and logical conditions (ALCONDs):

ALCOND = {GTR,EQL,GEQ,LSS,NEQ,LEQ,NON,ALL,ANY,NAL}

Note that jump instructions are subject to the pointer validation described in section 1.8.4 when they modify the PC.

SKP **Skip on condition**

SKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ,NON,ALL,ANY,NAL} . {Q,H,S,D} **SOP**

Purpose: If OP1 *condition* OP2 (where *condition* is taken from the first modifier) is true, control is transferred to the specified location that is within –8 . . 7 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: None

Precision: The precision of OP1 and OP2 is specified by the second modifier.

The following instructions compute the function "IF RTA is Odd THEN BEGIN RTA:=3*RTA+1 END; RTA:=RTA/2;" repeatedly while RTA>1. Note that FASM determines the SW offset automatically from the JUMPDEST operand:

```
THREEN:
        SKP.LEQ.S RTA,#1,DONE
        SKP.NON.S RTA,#1,RTAEVN ;skip if RTA has an even integer
        MULT.S RTA,#3           ;multiply by three
        ADD.S RTA,#1            ;add one - result must be even,
RTAEVN:                         ; so fall into even case
        SHFA.RT.S RTA,#1        ;this is better than QUO RTA,#2
        JMPA THREEN
DONE:   ...
```

ISKP — Increment, then skip on condition

ISKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **SOP**

Purpose: OP1:=OP1+1. CARRY is not affected. Then if OP1 ACOND OP2 (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to a location that is within -8 . . 7 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 and OP2 are both singlewords.

The following is a typical loop of the form, "FOR I:=M TO N DO ...". The inner part of the loop must not exceed 8 singlewords when assembled:

```
        MOV.S.S I,M
LOOP:
        ...
        ISKP.LEQ I,N,LOOP
```

DSKP — Decrement, then skip on condition

DSKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} SOP

Purpose: OP1:=OP1-1. CARRY is not affected. Then if OP1 ACOND OP2 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to a location that is within -8 . . 7 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 and OP2 are both singlewords.

The following instructions search an array of N singlewords starting at TABLE for the largest index I such that TABLE[I]=I. Assume that TABLE[0] contains 0 to ensure loop termination, and that N singlewords follow this entry. In the following, I must be a register. Note that since the loop is one instruction long the singleword skip offset is zero. The "-4" added to the base address TABLE compensates for the fact that the address calculation occurs *before* the decrementation operation, but the skip condition is tested *after* the decrementation operation. In turn, "N+1" is used instead of "N" in the initialization to compensate for this compensation:

```
        MOV.S.S I,#<N+1>
LOOP:   DSKP.NEQ I,<TABLE-4>[I]↑2,LOOP
```

JMP **Jump on condition**

JMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: If FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR, EQL, GEQ, LSS, NEQ, LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: None

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords.

The following loop searches down a chain of pointers for a specified tail pointer FPTR. Let P be a register and HEAD the address of the first link in the chain. Note that NEXT(P) is implicitly used by this routine to hold the comparison operand:

```
        MOV.D.D P,#<[HEAD ? FPTR]>      ;initialize P and NEXT(P)
                                        ;(this is an assembler literal
                                        ; whose address becomes a constant)
LOOP:   MOV.S.S P,(P)
        JMP.NEQ P,LOOP
```

JMPZ, FJMPZ — Jump on condition relative to zero

JMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} JOP
FJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {H,S,D} JOP

Purpose: If OP1 ACOND 0 is true (where ACOND∈{GTR, EQL, GEQ, LSS, NEQ, LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction. JMPZ tests integer/logical values, whereas FJMPZ tests floating point values.

While floating point values may be tested with the integer instructions, FJMPZ is to be preferred because it checks for the special floating point symbols and may be faster on some implementations.

Restrictions: None

Exceptions: In the case of FJMPZ, the FLT_NAN soft trap may occur.

Precision: OP1 has the precision specified by the second modifier.

By using the indexed constant addressing mode (Section 1.6.2), a programmer can use the JMPZ instruction to compare the contents of a register against any integer constant, not just against zero. For example, the following jumps to AWAY iff RTA≤1:

```
JMPZ.LEQ.S #[-1](RTA),AWAY
```

JMPA **Jump always**

JMPA **JOP**

Purpose: Jump unconditionally to JUMPDEST. For a simple jump to a label, the SJMP instruction is often more compact, but JMPA allows indexing and indirect addressing, usually at the expense of an extra singleword.

Restrictions: None

Exceptions: None

Precision: None

The following instruction jumps to the RTA-th address stored in a list of indirect pointers that begins at JVECTS:

```
JMPA  JVECTS[RTA]↑2
```

IJMP — Increment, then jump on condition

IJMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} JOP

Purpose: FIRST(OP1):=FIRST(OP1)+1. CARRY is not affected. Then if FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords.

The following is a typical loop of the form, "FOR I:=M TO N DO . . .". The inner part of the loop may be any length when assembled:

```
        MOV.D.D I,[M ? N]              ;M,N are assembly literals
LOOP:
        ...
        IJMP.LEQ I,LOOP
```

IJMPZ — Increment, then jump on condition relative to zero

IJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: OP1:=OP1+1. CARRY is not affected. Then if OP1 ACOND 0 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 is a singleword.

The following increments N and jumps to AWAY if N=0:

```
IJMPZ.EQL N,AWAY
```

IJMPA — Increment and jump always

IJMPA — JOP

Purpose: OP1:=OP1+1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 is a singleword.

The following is an extremely inefficient way to add RTA into RTB, assuming that integer overflow traps are disabled. However, it shows off the IJMPA instruction:

```
LOOP:   DSKP.EQL RTA,#-1        ;decrement RTA; skip next instruction if -1
        IJMPA RTB,LOOP          ;otherwise increment RTB and loop
```

DJMP **Decrement, then jump on condition**

DJMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: FIRST(OP1):=FIRST(OP1)-1. CARRY is not affected. Then if FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords.

The following is a typical loop of the form, "FOR I:=M DOWNTO N DO...". The inner part of the loop may be any length when assembled:

```
        MOV.D.D I,[M ? N]              ;M,N are assembly literals
LOOP:
        ...
        DJMP.GEQ I,LOOP
```

DJMPZ — Decrement, then jump on condition relative to zero

DJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} JOP

Purpose: OP1:=OP1-1. CARRY is not affected. Then if OP1 ACOND 0 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 is a singleword.

The following decrements N and jumps to AWAY if N=0:

```
DJMPZ.EQL N,AWAY
```

DJMPA — Decrement and jump always

DJMPA **JOP**

Purpose: OP1:=OP1-1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 is a singleword.

The following decrements N and jumps to AWAY:

```
DJMPA N,AWAY
```

SJMP **Simple jump**

SJMP **HOP**

Purpose: Unconditionally jump anywhere in the address space.

The HOP format performs a PC-relative jump using a 29 bit unsigned displacement field. Because the address calculation "wraps around" if it exceeds the maximum virtual address, it can reach any singleword in the virtual address space. No segment bounds checking occurs.

While SJMP never occupies more than 1 singleword, it allows only a direct memory address reference. One must use JMPA for any other addressing mode, such as indexing or indirect addressing.

Restrictions: None

Exceptions: None

Precision: None

Go to CRUNCH:

```
SJMP CRUNCH
```

MOVCSF, MOVCSS — Move conditionally, skip on failure/success

MOVCSF . {Q,H,S,D} **SOP**
MOVCSS . {Q,H,S,D} **SOP**

Purpose: For MOVCSF, IF OP1=OP2 THEN OP1:=R3 ELSE GOTO DEST.

For MOVCSS, IF OP1=OP2 THEN BEGIN OP1:=R3; GOTO DEST END.

In a multiprocessor system, these instructions are atomic (that is, they finish work on OP1 before any other processor can alter that operand). Another atomic instruction is RMW.

Restrictions: None

Exceptions: None

Precision: OP1, OP2, and R3 have the precision specified by the modifier.

Singleword LOCK represents a lock, which holds -1 if unlocked and 0 if locked. The following sequence seizes the lock, using busy-waiting if the lock is not free:

```
;;; Seize the lock stored in location LOCK.
        MOV.S.S R3,#-1          ;Prepare the value -1 to be stored.
LOOP:   MOVCSF.S LOCK,#0,LOOP   ;Store -1 when LOCK holds 0.
```

The following code sequence atomically turns on bit 35 of word F01.

```
;;; Turn on bit 35 of word F01.
LOOP:   MOV.S.S RTA,F01         ;Pick up a copy of the former value of F01.
        OR.S R3,RTA,#2          ;Turn on bit 35, creating the new value in R3.
        MOVCSF.S F01,RTA,LOOP   ;Store the new value if the value has not
                                ;changed since we began.
```

The following code sequence leaves in R3 a unique number; no two callers will ever be returned the same number even if they run this routine simultaneously from different processors. The location UNIQUE holds a number, whose value is increased by one atomically to get the new unique value.

```
;;; Return a unique value in R3 (well, until the counter wraps).
LOOP:   MOV.D.D RTA,UNIQUE      ;Get the old value of UNIQUE.
        ADD.D.D R3,RTA,#1       ;The new value should be one greater.
        MOVCSF.D UNIQUE,RTA,LOOP        ;Store the new value if the value
                                ;of UNIQUE has not changed in the meantime.
```

The following code sequence atomically adds a new element to a singly linked list. The pointer to the first list element is stored in location HEAD; the first word of each element contains a pointer to the next element. Register R3 contains a pointer to a new element to be added to the head of the list.

```
;;; Add the element in R3 to the list.
LOOP:   MOV.S.S RTA,HEAD          ;Pick up the pointer to the former first
                                  ;element of the list.
        MOV.S.S (R3),RTA          ;Make the new element point to it.
        MOVCSF.S HEAD,RTA,LOOP    ;Store the new pointer if the old one
                                  ;has not changed.
```

CMPSF, UCMPSF, FCMPSF — Compare and set flag

CMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} **TOP**
UCMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} **TOP**
FCMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {H,S,D} **TOP**

Purpose: If S1 *condition* S2 then DEST := -1 else DEST := 0, where *condition* is the first modifier. CMPSF performs a two's complement signed comparison; UCMPSF performs an unsigned comparison, and FCMPSF compares floating point numbers.

While floating point values can be compared using integer/logical comparisions, FCMPSF should be used in preference to CMPSF to obtain checking for the special floating point symbols. FCMPSF may also be faster on some implementations.

Restrictions: None

Exceptions: In the case of FCMPSF only, the FLT_NAN soft trap may occur.

Precision: S1 and S2 have the same precision as the modifier. DEST is a singleword.

Let X, Y, and Z be singlewords, with Y=NEXT(X). The following code implements setting RTA to X if Z≥0 and to Y otherwise. It uses indexing rather than a conditional jump or skip. Such use of indexing can often make more effective use of instruction pipelining than jumping or skipping:

```
CMPSF.GEQ.S RTA,Z,#0
MOV.S.S RTA,Y[RTA]↑2     ;indexing with flag result
```

CMPSF.LSS can be used to produce an extended-sign word for a number. TRANS or FTRANS can be used to sign-extend a number to one of the four standard precisions, but this trick is useful in dealing with numbers of very large precision:

```
CMPSF.LSS.S RTA,NUM,#0  ;all bits of RTA get the sign bit of NUM
```

Though instructions CMPSF.{NON,ALL,ANY,NAL} do not exist, their effect can be obtained by an AND or ANDCT followed by a CMPSF.EQL or CMPSF.NEQ:

```
ANDCT.S RTA,ARG1,ARG2    ;this behaves as would the fictional
CMPSF.EQL.S RTA,#0       ; instruction  CMPSF.ALL RTA,ARG1,ARG2
```

BNDSF — Bounds check and set flag

BNDSF . {RTA,RTB} . {B,M1,0,1} . {Q,H,S,D} **XOP**

Purpose: Check OP2 against a set of bounds determined by the second modifier together with OP1. If OP is within bounds then RTA := -1 else RTA := 0.

The second modifier specifies the following bounds checks:

Modifier	Meaning
B ("both")	FIRST(OP1) ≤ OP2 ≤ SECOND(OP1)
M1	-1 ≤ OP2 ≤ OP1
0	0 ≤ OP2 ≤ OP1
1	1 ≤ OP2 ≤ OP1

The BNDSF instruction lacks some of the modifiers available on the BNDTRP instruction because they would duplicate the operation of the CMPSF instruction.

Restrictions: None

Exceptions: None

Precision: OP2, RTA or RTB have the precision of the third modifier; OP1 has the precision of the third modifier except in the "B" case; there, FIRST(OP1) and SECOND(OP1) must each have the precision of the third modifier, and must align together to form a single entity with twice that precision.

This is a typical use of BNDSF:

```
BNDSF.RTA.Ø.S LIMIT,X          ; Ø  ≤ X ≤ LIMIT ?
```

Even if the lower bound is not 0, 1, or -1, one can still use the indexed constant addressing mode to scale OP2 and thereby avoid using the more elaborate "B" mode, provided the lower bound is indeed a constant:

```
BNDSF.RTA.Ø.S #<LIMIT-3>,#[-3](X)      ; 3 ≤ (X) ≤ LIMIT
```

One can use constant addressing modes to obtain other limits if the limits are both constant. This makes use of the rule that a singleword instruction which expects a FIRST/SECOND operand pair will expand a constant to twice the specified precision and use half for the FIRST part and half for the SECOND part:

```
BNDSF.RTA.B.S #[LIMIT ? !Ø],X    ; #LIMIT ≤ X ≤ Ø ?
```

BNDTRP — Bounds check and trap

BNDTRP . {B,M1,0,1,NEQ,EQL,GEQ,GTR,LSS,LEQ} . {Q,H,S,D} **XOP**

Purpose: Check OP2 against a set of bounds determined by the first modifier together with OP1, and cause a BOUNDS_TRAP soft trap if it is not within bounds.

The modifier specifies the bounds check as follows:

Modifier	Trap occurs when:
B ("both")	¬(FIRST(OP1) ≤ OP2 ≤ SECOND(OP1))
M1	¬(-1 ≤ OP2 ≤ OP1)
0	¬(0 ≤ OP2 ≤ OP1)
1	¬(1 ≤ OP2 ≤ OP1)
NEQ	OP2 <> OP1
EQL	OP2 = OP1
GEQ	OP2 >= OP1
GTR	OP2 > OP1
LSS	OP2 < OP1
LEQ	OP2 <= OP1

Restrictions: None

Exceptions: None

Precision: OP2 has the precision of the second modifier; OP1 has the precision of the second modifier except in the "B" case; there, FIRST(OP1) and SECOND(OP1) must each have the precision of the modifier, and must align together to form a single entity with twice that precision.

This is a typical use of BNDTRP to test that an array index is within bounds:

```
BNDTRP.0.S LIMIT,X              ; 0  <= X <= LIMIT ?
```

STRCMP String compare

STRCMP . {RTA,RTB} XOP

Purpose: Consider OP1 and OP2 to be blocks of quarterwords--in other words, strings of characters--whose quarterword length is specified by SIZEREG. Signed comparison is used, and each quarterword character is compared separately. The result of the comparison is computed as shown in the following table and is stored into {RTA,RTB}. The result values are designed to have two useful properties. First, the result (as a signed integer) bears the same relation to zero that STRING1 does to STRING2. Second, the value can be used as an index into the string no matter what the result, because indexing arithmetic "wraps around" the address space.

Neither string may lie in the registers, but if one string is a constant, that constant will be replicated SIZEREG times.

Condition	Result
STRING1 = STRING2	0
STRING1 > STRING2	n
STRING1 < STRING2	-2^{35}+n (i.e. MINNUM+n)

(n is the position of the first character to differ indexing from 1)

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are quarterword blocks, not vectors, and thus may designate registers. RTA and RTB are single words.

The following sets RTA to the result of comparing the eighty-character blocks at X and Y.

```
MOV.S.S SIZEREG,#80.
STRCMP.RTA X,Y
```

Because the constant operand gets replicated to match the length of the string, the following tests whether STRINGA is entirely blank:

```
MOV.S.S SIZEREG,#80.
STRCMP.RTA STRINGA,#40
```

The following illustrates a more general sort of comparison. Assume that XLENGTH contains the length of a string beginning at X and YLENGTH that of string at Y. For the purposes of this comparison we will imagine that appended to the two strings are infinitely many imaginary

characters defined to be "less than" all real characters. We will then define the result of the comparison as the result of a STRCMP performed on these extended strings. (This definition is similar to that used in some high-level languages):

```
        MIN.S RTA,XLENGTH,YLENGTH       ;set RTA to minimum real length
        MOV.S.S SIZEREG,RTA
        INC.S RTB,RTA                   ;save one greater in RTB for
                                        ;unequal case
        STRCMP.RTA X,Y                  ;do comparison
        JMPZ.NEQ.S RTA,DONE             ;difference found
        SKP.NEQ.S XLENGTH,YLENGTH       ;done if strings are equal length
        JMPA DONE
        MOV.S.S RTA,RTB                 ;RTB is index of "imaginary"
                                        ;character
        SKP.LEQ.S XLENGTH,YLENGTH,DONE  ;set high-order bit if necessary
        OR.S RTA,#<400000,,0>           ;or  DIBYT RTA,#1,#1  to save a word!
DONE:   ...                             ;RTA contains result
```

SKPTAG — Test pointer tag and skip

SKPTAG — SOP

Purpose: Test bit S2<S1<0:4>>. If it is 1, then control is transfered to a location that is within -8 . . 7 singlewords of the PC.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

Skip if the pointer has a fault tag.

```
SKPTAG P,#400000000020,BADPTR
```

JMPTAG — Set pointer tag

JMPTAG . {1 .. 30,RING,FAULT} . {EQL,NEQ} **JOP**

Purpose: Jump to JUMPDEST if OP1<0:4> matches (EQL), or does not match, (NEQ) the specified tag. If the modifier is one of 1 . . 30, then OP1<0:4> must exactly equal the tag. If the modifier is RING, then a match occurs if OP1<0:4> is any ring tag (4 . . 7). If the modifier is FAULT, then OP4<0:4> must be a fault tag (0 or 31).

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following branches to the location pointed to by P unless P is NIL.

```
JMPTAG.2.NEQ P,(P)Ø
```

CMPTAGSF — Check pointer tag and set flag

CMPTAGSF TOP

Purpose: If bit S2<S1<0:4>> is 1 then DEST := -1, else DEST := 0.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

In the following example, set the flag in RTA if the pointer is not NIL.

```
        CMPTAGSF RTA,P,#677777777760
```

TAGTRP Check pointer tag

TAGTRP **TOP**

Purpose: If bit S2<S1<0:4>> is 1, then copy S1 to DEST; otherwise, a TAG_TRAP soft trap occurs.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

In the following example, move the pointer, P, into RTA if P has a user tag in the range 12 . . 16.

```
TAGTRP RTA,P,#000074000000
```

2.8 Shift, Rotate, and Bit Manipulation

These instructions all manipulate bits within a word, either by shifting, by rotating, or by performing bitwise logical functions. Note that a left shift (or rotate) by N is equivalent to the corresponding right shift (or rotate) by -N.

NOT — Logical bit-wise NOT

NOT . {Q,H,S,D} **XOP**
VNOT . {H,S,D} **V:=V**

Purpose: NOT computes OP1:=(¬OP2), where "¬" signifies *one's complement*

VNOT performs NOT on each element of the vector beginning with OP2 and stores the result in the corresponding element of the vector beginning with OP1.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 (or the elements of vectors OP1 and OP2) have the same precision as the modifier.

The following is an alternate to NEG RTA:

```
NOT.S RTA,#[-1](RTA)      ;RTA := -RTA
```

AND — Logical bit-wise AND

AND . {Q,H,S,D} **TOP**
VAND . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: AND computes DEST:=S1∧S2.

VAND performs AND on each element of vector OP1 and the corresponding element of OP2. It puts the results either back into vector OP1 or into the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] ∧ OP2[i]
    ELSE SR0@[i]:=OP1[i] ∧ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For AND, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VAND, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of AND:

```
AND.Q RTA,#3,#5          ;RTA:=1 (QW)
```

ANDTC — Logical bit-wise AND true/complement

ANDTC . {Q,H,S,D} **TOP**
VANDTC . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: DEST:=S1∧(¬S2). Note that the "TC" in ANDTC means "True-Complement" and refers to the fact that S1 and *one's-complement*(S2) respectively are operands to the AND function. The reverse form of ANDTC is ANDCT, *not* ANDTCV.

VANDTC performs ANDTC on pairs of corresponding elements in the vectors beginning at OP1 and OP2. It puts the results back into the vector OP1 or into the vector pointed to by SR0, depending on the first modifier.

```
FOR I:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] ∧ (¬OP2[i])
    ELSE SR0@[i]:=OP1[i] ∧ (¬OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For ANDTC, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VANDTC, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ANDTC:

```
ANDTC.Q RTA,#3,#5              ;RTA:=2 (QW)
```

Suppose that MASK is a mask whose "1" bits select certain (possibly non-contiguous!) bits of WORD. These bits are to be regarded as a "field", and the contents of that field *decremented* as an integer "in place" in WORD, without affecting non-selected bits of WORD. This can be done as follows:

```
AND.S RTA,WORD,MASK     ;RTA:=WORD with non-selected bits zeroed
SUB.S RTA,#1            ;zeroed bits propagate the borrow
AND.S RTA,MASK          ;mask out non-selected bits
ANDTC.S WORD,MASK       ;mask out SELECTED bits in WORD
OR.S WORD,RTA           ;merge the two results
```

ANDCT — Logical bit-wise AND complement/true

ANDCT . {Q,H,S,D} **TOP**
VANDCT . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: ANDCT computes DEST:=(¬S1)∧S2. Note that the "CT" in ANDCT means "Complement-True" and refers to the fact that *one's-complement*(S1) and S2 respectively are operands to the AND function. The reverse form of ANDCT is ANDTC, *not* ANDCTV.

VANDCT performs ANDCT on pairs of elements from the vectors beginning at OP1 and OP2. It puts the results back into the vector OP1 or into the vector pointed to by SR0, depending on the first modifier.

```
FOR I:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=(¬OP1[i]) ∧ OP2[i]
    ELSE SR0@[i]:=(¬OP1[i]) ∧ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For ANDCT, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VANDCT, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ANDCT:

```
ANDCT.Q RTA,#3,#5                ;RTA:=4 (QW)
```

OR — Logical bit-wise OR

OR . {Q,H,S,D} **TOP**
VOR . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: OR computes DEST:=S1∨S2.

VOR performs OR on pairs of elements from the vectors OP1 and OP2, putting the results into vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] ∨ OP2[i]
    ELSE SR0@[i]:=OP1[i] ∨ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For OR, S1, S2, and DEST all have the precision specified by the modifier {Q,H,S,D}. For VOR, the elements of the vectors all have the precision specified by the modifier {H,S,D}.

The following instruction illustrates the effect of OR:

```
OR.Q RTA,#3,#5          ;RTA:=7 (QW)
```

ORTC — Logical bit-wise OR true/complement

ORTC . {Q,H,S,D} **TOP**
VORTC . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: ORTC computes DEST:=S1∨(¬S2). Note that the "TC" in ORTC means "True-Complement" and refers to the fact that S1 and *one's-complement*(S2) respectively are operands to the OR function. The reverse form of ORTC is ORCT, *not* ORTCV.

VORTC performs ORTC on pairs of elements of the vectors OP1 and OP2, putting the results in either vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] ∨ (¬OP2[i])
    ELSE SR0@[i]:=OP1[i] ∨ (¬OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For ORTC, S1, S2, and DEST all have the precision specified by the second modifier. For VORTC, the elements of the vectors all have the precision specified by the second modifier.

The following instruction illustrates the effect of ORTC:

```
ORTC.Q RTA,#3,#5        ;RTA:=773 (QW)
```

Suppose that MASK is a mask whose one-bits select certain (possibly non-contiguous!) bits of WORD. These bits are to be regarded as a "field", and the contents of that field *incremented* as an integer "in place" in WORD, without affecting non-selected bits of WORD. This can be done as follows:

```
ORTC.S RTA,WORD,MASK    ;RTA:=WORD with non-selected bits set to "1"
ADD.S RTA,#1            ;"1" bits propagate the carry
AND.S RTA,MASK          ;mask out non-selected bits
ANDTC.S WORD,MASK       ;mask out SELECTED bits in WORD
OR.S WORD,RTA           ;merge the two results
```

ORCT — Logical bit-wise OR complement/true

ORCT . {Q,H,S,D} **TOP**
VORCT . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: ORCT computes DEST:=(¬S1)∨S2. Note that the "CT" in ORCT means "Complement-True" and refers to the fact that *one's-complement*(S1) and S2 respectively are operands to the OR function. The reverse form of ORCT is ORTC, *not* ORCTV.

VORCT performs ORCT on pairs of elements of vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=(¬OP1[i]) ∨ OP2[i]
    ELSE SR0@[i]:=(¬OP1[i]) ∨ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For ORCT, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VORCT, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ORCT:

```
ORCT.Q RTA,#3,#5          ;RTA:=775 (QW)
```

NAND — Logical bit-wise NAND

NAND . {Q,H,S,D} **TOP**
VNAND . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: NAND computes DEST:=¬(S1∧S2).

VNAND performs NAND on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=¬(OP1[i]) ∧ OP2[i]
    ELSE SR0@[i]:=¬(OP1[i] ∧ OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For NAND, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VNAND, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of NAND:

```
NAND.Q RTA,#3,#5          ;RTA:=776 (QW)
```

NOR — Logical bit-wise NOR

NOR . {Q,H,S,D} **TOP**
VNOR . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: NOR computes DEST:=¬(S1∨S2), where "¬" signifies *one's complement.*

VNOR performs NOR on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=¬(OP1[i] ∨ OP2[i])
    ELSE SR0@[i]:=¬(OP1[i] ∨ OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For NOR, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VNOR, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of NOR:

```
NOR.Q RTA,#3,#5          ;RTA:=770 (QW)
```

XOR — Logical bit-wise XOR

XOR . {Q,H,S,D} **TOP**
VXOR . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: XOR computes DEST:=(S1∧¬(S2)) ∨ (¬(S1)∧S2), where "¬" represents the one's complement operation.

VXOR performs XOR on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=ExclusiveOR(OP1[i],OP2[i])
    ELSE SR0@[i]:=ExclusiveOR(OP1[i],OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For XOR, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VXOR, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of XOR:

```
XOR.Q RTA,#3,#5          ;RTA:=6 (QW)
```

The following code exchanges the two words QUUX and ZTESCH. (A better way to do this is with the EXCH instruction, but this example demonstrates an interesting information-preserving property of XOR.)

```
XOR.S QUUX,ZTESCH
XOR.S ZTESCH,QUUX
XOR.S QUUX,ZTESCH
```

EQV — Logical bit-wise equivalence

EQV . {Q,H,S,D} **TOP**
VEQV . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: EQV computes DEST:=(S1∧S2) ∨ (¬(S1)∧(¬S2)), where "¬" represents the one's complement operation.

VEQV performs EQV on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=EQV(OP1[i],OP2[i])
    ELSE SR0@[i]:=EQV(OP1[i],OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For EQV, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VEQV, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of EQV:

```
EQV.Q RTA,#3,#5          ;RTA:=771 (QW)
```

The following code exchanges the two words QUUX and ZTESCH. (A better way to do this is with the EXCH instruction, but this example demonstrates an interesting information-preserving property of EQV.)

```
EQV.S QUUX,ZTESCH
EQV.S ZTESCH,QUUX
EQV.S QUUX,ZTESCH
```

SHFA — Shift arithmetically

SHFA . {LF,RT} . {Q,H,S,D} **TOP**
SHFAV . {LF,RT} . {Q,H,S,D} **TOP**
VSHFA . {LF,RT} . {H,S,D} **V:=VS**

Purpose: SHFA computes DEST:=S1 arithmetically shifted {left,right} by S2. Shifts to the (true) left introduce "0" bits; shifts to the (true) right replicate the sign bit and discard bits shifted out the low end. This is equivalent to a multiplication or division by a power of two, where it is understood that such a division rounds towards negative infinity. Note that a left shift by S1 is equivalent to a right shift by -S1. If the absolute value of S2 exceeds the width of the anyword being shifted, an ILLEGAL_SHIFT_ROTATE hard trap occurs.

SHFAV swaps the roles of S1 and S2.

VSHFA performs SHFA on each element of the vector beginning at OP2 and stores the results in the corresponding elements of OP1. RTA specifies how far to shift each element.

Restrictions: None

Exceptions: INT_OVFL on certain left shifts (the instruction behaves exactly as would a multiplication by a power of 2)

Precision: For SHFA, S2 is a singleword, and DEST and S1 have the precision specified by the second modifier.

For SHFAV, S1 is a singleword, and DEST and S2 have the precision specified by the second modifier.

For VSHFA, the elements of vectors OP1 and OP2 have the precision of the modifier and RTA is a scalar singleword.

The following two instructions illustrate the difference between SHF.RT and SHFA.RT:

```
SHF.RT.Q RTA,#-1,#1      ;RTA:=377
SHFA.RT.Q RTA,#-1,#1     ;RTA:=777
```

SHF — Logical shift

SHF . {LF,RT} . {Q,H,S,D} **TOP**
SHFV . {LF,RT} . {Q,H,S,D} **TOP**
VSHF . {LF,RT} . {H,S,D} **V:=VS**

Purpose: SHF computes DEST:=S1 logically shifted {left,right} by S2. Bits shifted in are "0" bits; bits shifted out are lost. Note that a left shift by S2 is identical to a right shift by -S2. If the absolute value of S2 exceeds the width of the anyword being shifted, an ILLEGAL_SHIFT_ROTATE hard trap occurs.

SHFV, the reverse form, behaves identically except that it swaps the roles and precisions of S1 and S2.

VSHF performs SHF on each element of the vector beginning with OP2 and stores the results in the corresponding elements of the vector beginning with OP1. RTA specifies the number of bit positions by which to shift.

Restrictions: None

Exceptions: None

Precision: For SHF, S2 is a singleword; DEST and S1 have the precision specified by the second modifier. For SHFV, S1 is a singleword; DEST and S2 have the precision of the second modifier. For VSHF, RTA is a singleword; the elements of OP1 and OP2 have the precision specified by the modifier.

The following shows the effect of a positive left-shift argument:

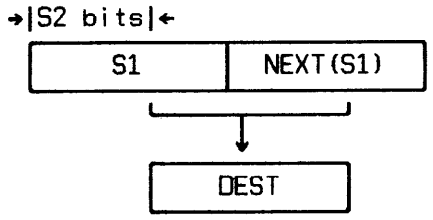
```
SHF.LF.Q RTA,#-1,#1      ;RTA:=-2 (QW)
```

DSHF — Extended logical shift

DSHF . {LF,RT} **TOP**
DSHFV . {LF,RT} **TOP**

Purpose: Just as a programmer can use the ADDC instruction repeatedly to add numbers of arbitrarily great precision, the programmer can use the DSHF instruction repeatedly to shift an arbitrarily long string of bits. Ordinary logical shift instructions are difficult to chain in this fashion because they shift zeros into the word. DSHF solves the problem by shifting in bits from the adjacent word in memory instead.

More precisely, DSHF.LF concatenates S1 and NEXT(S1), logically shifts the resulting double precision entity left by S2 bits and stores in DEST the high order singleword (corresponding to Q, H, or S precisions). DSHF.RT logically shifts the entire entity right by S2 bits and stores in NEXT(DEST) the low order singleword.




```
DSHF.LF DEST,S1,S2
```

Careful use of DSHF even permits in-place shifting--that is, leaving the result of the shifting in the original memory locations: right shifts must start at the right end of the series of words, and long left shifts must start at the left end.

DSHFV, the reverse form, swaps the roles of S1 and S2.

See also the vector instruction VDSHF.

Restrictions: DSHF S1 and DSHFV S2 may not be constants.

Exceptions: An ILLEGAL_SHIFT_ROTATE hard trap occurs if S2 is negative or exceeds the width of the singleword being shifted or if S2 < 0.

Precision: For DSHF, operands S1, NEXT(S1), and DEST (or NEXT(DEST)) singlewords.

For DSHFV, the same is true except that the roles of S1 and S2 are swapped.

Suppose that a 30-word block of bits MARKERS is to be logically shifted in place three bits to the left. While using VDSHF provides better performance, the following example illustrates the

use of DSHF within an explicit loop:

```
        MOV.S.S RTB,#0                  ;RTB indexes MARKERS left to right
LOOP:   DSHF.LF.S MARKERS[RTB]↑2,#3     ;produce one result word
        ISKP.LSS RTB,#29.,LOOP          ;increment RTB and loop if < 29.
        SHF.LF.S MARKERS+29.*4,#3       ;do last word in single precision
```

The same block of bits can be logically shifted three bits to the *right* as follows. Note that the operation must proceed in the other direction within the block, i.e. from right to left:

```
        MOV.S.S RTB,#29.                ;RTB indexes MARKERS right to left
LOOP:   DSHF.RT.S MARKERS[RTB]↑2,#3  ;produce one result word
        DSKP.GTR RTB,#0,LOOP            ;decrement RTB and loop if > 0
        SHF.RT.S MARKERS,#3             ;do last word in single precision
```

The same block of bits can be *arithmetically* shifted three bits to the right by using the same loop but changing the last SHF.RT instruction to SHFA.RT.

VDSHF — Lengthwise vector logical shift

VDSHF . {LF,RT} **V:=VS**

Purpose: Logically shift an arbitrarily long series of bits. OP2 is the first word of the source vector, OP1 is the first word of the destination vector, SIZEREG gives the length of the vector in singlewords, and RTA specifies how far to shift the bits.

If the source and destination vectors overlap at all, they must coincide completely, or the result is undefined. An ILLEGAL_SHIFT_ROTATE hard trap occurs if the absolute value of RTA is greater than 36.

VDSHF.RT does not alter the first word of the vector, and VDSHF.LF does not alter the last word. This allows the programmer to use a scalar shift or rotate instruction to finish the operation, and thereby obtain a logical shift, arithmetic shift, or rotation. This also permits chaining of VDSHF instructions.

This instruction accomplishes the same task as a loop that applies the scalar DSHF instruction to a series of words, one at a time (see the example under the discussion of DSHF). For all but the shortest series of bits, the vector version will execute more rapidly, but the scalar version gives a choice of precisions.

Restrictions: None

Exceptions: An ILLEGAL_SHIFT_ROTATE hard trap occurs if the absolute value of RTA is negative or greater than 36.

Precision: The elements of both vectors are singlewords in terms of alignment (though the instruction can operate on larger sections of the vector to achieve greater speed). RTA and SIZEREG are singlewords.

This is a simple illustration of VDSHF and SHF combined to perform a logical shift:

```
MOV.S.S SIZEREG,#3        ; Length of vector is 3 singlewords
MOV.S.S RTA,#19.          ; Shift by 19 bit positions
VTRANS.S.S SOURCE,[1,,2 ? 3,,4 ? 5,,6]
                          ; "a,,b" tells FASM to put a in
                          ;   the left halfword, b in the right
VDSHF.LF DEST,SOURCE
SHF.LF.S <DEST-4*1>[SIZEREG]↑2,<source-4*1>[SIZEREG]↑2,RTA
                          ;   Result is [4,,6 ? 8.,,10. ? 12.,,0]
```

ROT — Logical rotate

ROT . {LF,RT} . {Q,H,S,D} **TOP**
ROTV . {LF,RT} . {Q,H,S,D} **TOP**

Purpose: ROT computes DEST:=S1 rotated {left,right} by S2. Rotation introduces bits shifted out of one end into the other end, so that no bits are lost. An ILLEGAL_SHIFT_ROTATE hard trap occurs if S2 is negative or exceeds the width of the anyword being shifted.

ROTV, the reverse form, rotates S2 left or right by S1 bits.

Restrictions: None

Exceptions: None

Precision: For ROT, S2 is a singleword. DEST and S1 have the precision specified by the second modifier.

For ROTV, S1 is a singleword; DEST and S2 have the precision of the second modifier.

The following illustrates a right rotation by a positive amount:

```
ROT.RT.Q RTA,#1,#1        ;RTA:=400 (QW)
```

BITRV — Bit reverse

BITRV . {Q,H,S,D} **TOP**
BITRVV . {Q,H,S,D} **TOP**

Purpose: BITRV reverses the order of the S2 low-order bits of S1, and zero-extends the result into DEST. An ILLEGAL_SHIFT_ROTATE hard trap occurs if S2 is negative or exceeds the word width.

BITRVV reverses the order of the S1 low-order bits of S2 instead.

Restrictions: None

Exceptions: None

Precision: For BITRV, S1 and DEST have the same precision as the modifier. S2 is a singleword.

For BITRVV, S2 and DEST have the precision of the modifier; S1 is a singleword.

The following reverses all nine bits of its operand:

```
BITRV.Q RTA,#[123],#9.  ;RTA:=624 (QW)
```

BITEX — Bit extract

BITEX . {Q,H,S,D} **TOP**
BITEXV . {Q,H,S,D} **TOP**

Purpose: BITEX extracts the bits of S1 selected by the "1" bits of S2. It squeezes these selected bits to the right, zero-extends them, and stores them into DEST.

BITEXV, the reverse form, swaps the roles of S1 and S2.

Restrictions: None

Exceptions: None

Precision: S1, S2, and DEST all have the precision specified by the modifier.

The following extracts alternate bits from the operand:

```
BITEX.Q RTA,#[765],#[525]        ;RTA:=37 (QW)
```

BITCNT — Bit count

BITCNT . {Q,H,S,D} **XOP**
VBITCNT . {H,S,D} **V:=V**
LBITCNT . {H,S,D} **S:=V**

Purpose: BITCNT computes OP1:=*number of "1" bits in* OP2. This instruction is useful for counting the number of elements in a Pascal set.

VBITCNT performs BITCNT on each element of the vector beginning at OP2 and stores the results in the corresponding elements of the vector beginning at OP1.

LBITCNT counts all the "1" bits in all elements of the vector OP1 and stores the resulting total in singleword RTA. If the length of the vector is 0 or negative, the total returned will be zero.

Restrictions: None

Exceptions: None

Precision: For BITCNT, OP1 is a singleword and OP2 has the same precision as the modifier. For VBITCNT, the elements of vector OP1 are singlewords and those of OP2 have the same precision as the modifier. For LBITCNT, RTA is a singleword and the elements of vector OP1 have the precision specified by the modifier.

The following sets RTA to -1 if RTA has odd parity, 0 otherwise:

```
BITCNT.S RTA,RTA
AND.S RTA,#1
NEG.S RTA
```

The parity of an arbitrarily long block of bits can be obtained by using LBITNT. If TABLE is a block of N singlewords, this code sets RTA (flat-style) if TABLE has odd parity:

```
MOV SIZEREG,N
LBITCNT.S RTA, TABLE
AND.S RTA,#1
NEG.S RTA
```

A non-zero integral power of two always has a two's-complement representation with exactly one bit set. Assuming that HUNOZ contains a positive singleword integer, this code jumps to TWOPOWER if HUNOZ is an exact power of two:

```
BITCNT.S RTA,HUNOZ        ;RTA←1 if HUNOZ is a power of two
DJMPZ.EQL RTA,TWOPOWER    ; jump to TWOPOWER if RTA-1 is zero
```

If zero is to be considered a power of two, DJMPZ.EQL can be changed to DJMPZ.LEQ.

BITFST — Bit number of first "1" bit

BITFST . {Q,H,S,D} **XOP**
LBITFST . {H,S,D} **S:V**

Purpose: For BITFST, if OP2=0 then OP1:=-1 else OP1:=*bit number of the leftmost "1" bit in* OP2. This instruction is useful for finding the index of the first element of a Pascal set.

LBITFST finds the first "1" bit in vector OP1 and assigns it to RTA. If there are no "1" bits in the vector or if the length of the vector is zero or negative, then -1 is assigned to RTA.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. For BITFST, OP2 has the same precision as the modifier. For LBITFST, each element of OP1 has the same precision as the modifier.

The following sets RTA to floor(log2(RTA)) with RTA assumed to be a non-zero unsigned singleword integer:

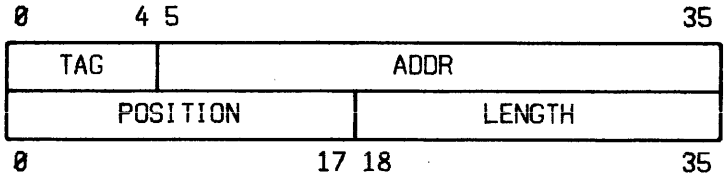
```
BITFST.S RTA,RTA
SUBV.S RTA,#35.
```

This piece of code constructs a byte pointer in (doubleword) RTA to the smallest byte containing all the one-bits in HUNOZ:

```
BITFST.S RTA,HUNOZ       ;number of leading ''0'' bits
BITRV.S RTA1,HUNOZ,#36.  ;reverse HUNOZ into RTA1
BITFST.S RTA1            ;number of trailing ''0'' bits
ADD.S RTA1,RTA           ;number of surrounding ''0'' bits
SUBV.S RTA1,#36.         ;length of smallest containing byte
MOV.H.S RTA1,RTA         ;put position in high halfword of RTA1
MOVP.P.A RTA,HUNOZ       ;make pointer to HUNOZ in RTA
```

2.9 Byte Manipulation

Bytes, byte pointers, and byte selectors: A byte is simply a field of zero or more bits within a singleword or doubleword. The native mode architecture does not tie the concept of a byte to the representation of a character. Instead, it lets the programmer specify the position and width of a byte by constructing a byte pointer:

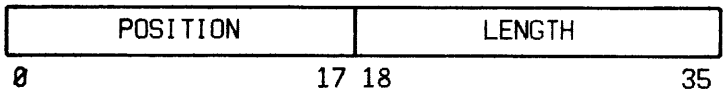


The TAG and ADDR fields compose a pointer (as described in Section 1.8.1), and are subject to the validation checking described in Section 1.8.2. They must point to an aligned singleword in memory--that is, ADDR must be a multiple of 4. The POSITION field gives the bit number within the singleword or doubleword at which the byte begins, and must lie within the range 0 . . 35 for singlewords or 0 . . 71 for doublewords. The LENGTH field gives the number of bits within the byte, and must lie within the range 0 . . 36 for singlewords or 0 . . 72 for doublewords. If the POSITION and LENGTH fields of a byte pointer violate any of those rules, an ILLEGAL_BYTE_PTR hard trap occurs.

One useful consequence of the format for byte pointers is the ability to compare them as if they were ordinary doublewords (provided that one knows the tag fields of the pointers match). The comparison will reveal which byte is higher in memory or, if the two bytes begin at the same position of the same word, which byte is longer.

There exist forms of the byte instructions to manipulate each of the following cases: a singleword byte within an aligned singleword, a doubleword byte within an aligned doubleword, a singleword byte within a pair of singlewords, and a doubleword byte within a triple of singlewords. The latter two are called long byte instructions and may be substantially less efficient than the former for which alignment is insured.

Immediate byte instructions use an operand to specify the singleword or doubleword containing a byte, and thus can access a byte within a constant or register as well as in memory. They use a simplified version of the byte pointer, called a *byte selector*, eliminating the TAG and ADDRESS fields:



Special byte instructions handle the most general form of byte instructions where one has four independent operands: a value (to load or store), the word(s) containing the byte, the position of the byte, and the length of the byte. The BYTDSC/LXBYT and BYTDSC/DXBYT sequences act as these four-operand forms using the register R3 as a temporary holding a byte selector.

The architecture recognizes three types of bytes. A unsigned byte is an non-negative integer. When extracted from memory, it is zero extended on the left to fit the destination. A signed byte is stored in twos complement form. When extracted, it is signed extended on the left. A logical byte is simply a sequence of bits (e.g. a Pascal set) which is indexed by bit position from left to right. When extracted, it is zero filled on the right to fit the destination.

The type of the byte is specified by the modifiers {R,A,L}:

R	Unsigned, right-aligned
A	Signed, right-aligned with sign extension
L	Logical, left-aligned

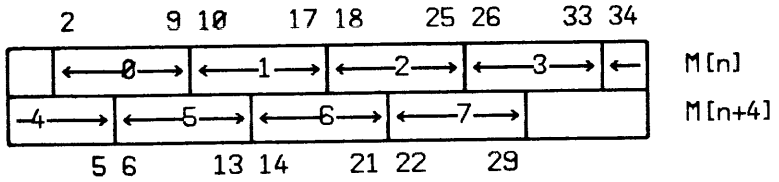
Adjusting byte pointers: It is often useful to "adjust" a byte pointer: that is, to assume that it points to one of a series of packed bytes, and to alter it to point to an earlier or later byte in the series. The following algorithms show how to do so for three different means of packing. In each one, "Addr", "Pos", and "Len" are the address, byte position, and byte length within the byte pointer which we are adjusting; "S2" tells how many bytes forward (or, if it is negative, how many bytes backward) to move the pointer. "DivNI" and "ModNI" are a division and modulus which round toward negative infinity.

```
(* Adjust byte pointer continuously. Positions bytes one after
   another without leaving any unused bits, and thus splitting
   bytes across word boundaries where necessary. Assume Len <= 72
   and (Len + Pos) <= 72. *)
PROCEDURE ADJBPC(VAR Addr, Pos, Len: INTEGER; S2: INTEGER);
   VAR
      BitOffset: INTEGER;
   BEGIN
   BitOffset := Pos + S2 * Len;
   Addr := Addr + DivNI(BitOffset, 36) * 4;
   Pos := ModNI(BitOffset, 36);
   END (* ADJBPC *);

(* Adjust byte pointer, maintaining alignment. Skips bits at
   high-order and/or low-order ends of singleword to avoid
   splitting a byte across a singleword boundary, but
   maintains the same alignment (that is, the same pattern
   of bytes and skipped bits) in each singleword. Assume that
   Addr, Pos, and Len specify a byte which does not cross a
   singleword boundary. *)
PROCEDURE ADJBPA(VAR Addr, Pos, Len: INTEGER; S2: INTEGER);
   VAR
      Alignment, BytesPer, BytesPerWord: INTEGER;
   BEGIN
```

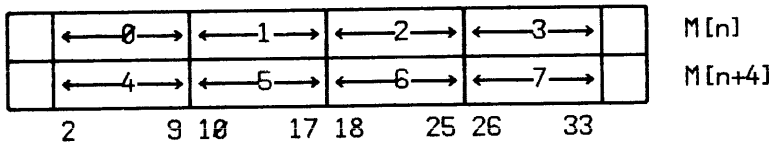
```
    Alignment := ModNI(Pos, Len);
    BytesPer := DivNI(Pos, Len);
    BytesPerWord :=  BytesPer + DivNI((36 - Pos), Len);
    Addr := Addr + DivNI((S2 + BytesPer), BytesPerWord) * 4;
    Pos  := Alignment + ModNI((S2 + BytesPer), BytesPerWord) * Len;
    END (* ADJBPA *);

(* Adjust byte pointer to zero-bit. Position bytes so that the
   high-order byte in each singleword begins at bit zero, and
   no byte crosses a singleword boundary. All skipped bits thus
   appear at the low-order end of the singleword. Assume that
   Addr, Pos, and Len specify a byte which does not cross a
   singleword boundary; if the entry values suggest a byte
   packing scheme that is not zero-bit aligned, impose zero-bit
   alignment if S2 causes the new value to point to a
   different singleword. *)
PROCEDURE ADJBPZ(VAR Addr, Pos, Len: INTEGER; S2: INTEGER);
  BEGIN
  BytesPerWord := DivNI(36, Len);
  BytesBefore := DivNI(Pos, Len);
  BytesAfter := DivNI((36 - Pos), Len) - 1;
  IF S2 >= 0 THEN
      BEGIN
      BS := BytesAfter + 1 - BytesPerWord;
      BytesLeftInWord := BytesAfter;
      END
  ELSE
      BEGIN
      BS := -BytesBefore;
      BytesLeftInWord := BytesBefore;
      END;
  IF ABS(S2) > BytesLeftInWord THEN
      BEGIN
      Addr := Addr + DivNI((S2 - BS), BytesPerWord) * 4;
      Pos  := ModNI((S2 - BS), BytesPerWord) * Len;
      END
  ELSE
      BEGIN
      (* Addr := Addr; *)
      Pos := Pos + S2 * Len;
      END;
  END (* ADJBPZ *);
```

To show the effect of the three different algorithms, assume that RTA is a byte pointer to an 8-bit byte beginning at bit 2 of singleword M[n]. Executing each algorithm eight times with S2 = 1 will cause it to point to eight successive bytes in memory, as shown in the drawings:

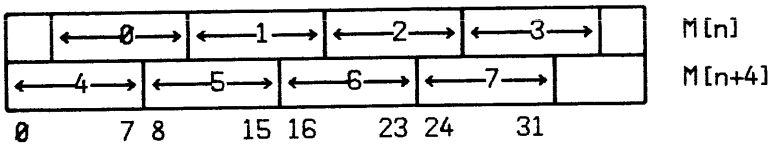
ADJBPC



ADJBPA



ADJBPZ



LBYT — Load byte from byte pointer

LBYT {R,A,L} . {S,D} **XOP**
LBYTL {R,A,L} . {S,D} **XOP**

Purpose: The instruction copies the byte specified by byte pointer OP2 and stores it, justified and extended according to the first modifier, in OP1.

Restrictions: None

Exceptions: In the case of LBYT, an ILLEGAL_BYTE_PTR occurs if the byte is not properly aligned within a datum with the precision specified by the second modified.

Precision: OP1 has the precision specified by the second modifier. OP2 is a byte pointer.

The following sets RTA to the exponent field of the singleword floating point number X (the exponent field is 9 bits wide and starts at bit 1 of the word):

```
LBYT.A.S RTA, [TAG+X ? 1,,9. ]
```

LIBYT — Load byte using byte selector

LIBYT {R,A,L} . {S,D} **TOP**
LIBYTL {R,A,L} . {S,D} **TOP**

Purpose: The instruction copies from S1 the byte specified by byte selector S2 and stores it, justified and extended according to the first modifier, in DEST.

Restrictions: None

Exceptions: In the case of LIBYT, an ILLEGAL_BYTE_PTR occurs if the byte is not properly aligned within a datum with the precision specified by the second modified.

Precision: DEST has the precision specified by the second modifier. S1 has the same precision for LIBYT, and and extra singleword for LIBYTL. S2 is a singleword byte selector.

The following sets RTA to the exponent field of the singleword floating point number X (the exponent field is 9 bits long and starts at bit 1 of the word):

```
LIBYT.A.S RTA,X,#[1,,9.]
```

LXBYT — Load byte using special selector

LXBYT {R,A,L} . {S,D} **XOP**
LXBYTL {R,A,L} . {S,D} **XOP**

Purpose: The instruction copies from OP2 the byte specified by a byte selector in R3 and stores it, justified and extended according to the first modifier, in OP1.

Restrictions: None

Exceptions: In the case of LXBYT, an ILLEGAL_BYTE_PTR occurs if the byte is not properly aligned within a datum with the precision specified by the second modified.

Precision: OP1 has the precision specified by the second modifier. OP2 has the same precision in the short form, or an extra singleword in the long form.

The following loads the Ith element of a TABLE of 7 bit quantities into the high-order bits of RTA.

```
MULT RTA,I,#7              ; get bit offset
MDIVH.FL.S RTA,RTA,[HRECIP36 ? LRECIP36]
                           ; divide by 36 to separate word and
                           ; bit offsets
BYTDSC RTB,#7              ; put byte selector in R3
LXBYTL.L.S RTA,TABLE[RTA]↑2 ; load the byte
```

DBYT — Deposit byte through byte pointer

DBYT {R,L} . {S,D} **XOP**
DBYTL {R,L} . {S,D} **XOP**

Purpose: The instruction copies the appropriate number of bits from OP2 and stores them in the byte specified by byte pointer OP1. The "R" form uses the low-order bits, while the "L" form takes the high-order bits.

Restrictions: None

Exceptions: In the case of DBYT, an ILLEGAL_BYTE_PTR hard trap will occur if the byte is not aligned within the precision specified by the second modifier.

Precision: OP1 is a byte pointer. OP2 has the precision specified by the second modifier.

The following sets the mantissa of the singleword floating point number X to the twenty-six low order bits of RTA (the mantissa is 26 bits long and begins at bit 10:

```
DBYT.R.S [TAG+X ? 10.,,26.],RTA
```

DIBYT — Deposit byte using byte selector

DIBYT {R,L} . {S,D} **TOP**
DIBYTL {R,L} . {S,D} **TOP**

Purpose: The instruction copies the appropriate number of bits from S1 and stores them in the byte within DEST specified by byte selector S2. The "R" form uses the low-order bits, while the "L" form takes the high-order bits.

Restrictions: None

Exceptions: In the case of DIBYT, an ILLEGAL_BYTE_PTR hard trap will occur if the byte is not aligned within the precision specified by the second modifier.

Precision: S1 has the precision specified by the second modifier. DEST has the same precision in the short form and an extra singleword in the long form. S2 is a singleword byte selector.

The following sets the exponent field of the singleword floating point number in RTA to zero. (The exponent field is 9 bits long and begins at bit 1):

```
DIBYT.R.S RTA,#0,#[1,,9.]
```

DXBYT — Deposit byte using special byte selector

DXBYT {R,L} . {S,D} **XOP**
DXBYTL {R,L} . {S,D} **XOP**

Purpose: The instruction copies the appropriate number of bits from OP2 and stores them in the byte within OP1 specified by a byte selector in R3. The "R" form uses the low-order bits, while the "L" form takes the high-order bits.

Restrictions: None

Exceptions: In the case of DXBYT, an ILLEGAL_BYTE_PTR hard trap will occur if the byte is not aligned within the precision specified by the second modifier.

Precision: OP2 has the precision specified by the second modifier. OP1 has the same precision in the short form and an extra singleword in the long form.

The following stores the high-order bits of RTA into the Ith element of a TABLE of 7 bit quantities:

```
MULT RTA,I,#7            ; get bit offset
MDIVH.FL.S RTA,RTA,[HRECIP36 ? LRECIP36]
                         ; divide by 36 to separate word and
                         ; bit offsets
BYTDSC RTB,#7            ; put byte selector in R3
DXBYTL.L.S TABLE[RTA]↑2,RTA ; store into the byte
```

BYTDSC — Build special byte selector

BYTDSC — XOP

Purpose: The instruction creates a byte selector in R3. OP1 is copied into the POSITION field and OP2 is copied into the LENGTH field. This is normally used in conjunction with the LXBYT and DXBYT instructions to obtain a full four-operand form byte instruction.

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

2.10 Stack Manipulation

A stack is specified by a pointer to its end. The architecture supports both stacks which grow upward in memory toward higher addresses and stacks which grow downward in memory toward lower addresses. Instructions which manipulate stacks generally specify either "UP" or "DN" (meaning down) as a modifier, indicating the direction in which they consider the stack to grow.

For upward-growing stacks, the stack pointer specifies the next *free* singleword on the stack, so that a push operation first stores the item and then increments the pointer. For downward-growing stacks, the pointer specifies the top item of the stack, so that a push operation first decrements the pointer and then stores the new item.

It is possible to check that a push or pop operation does not overflow the area allocated to the stack by using the segment bounds checking mechanism, describe in section 1.8.2. By using an entire segment for a stack, the bounds of the segment define the limits on the stack. Since push and pop operations perform regular address arithmetic to increment or decrement the stack pointer, an OUT_OF_BOUNDS hard trap will occur on stack overflow. Note that in an upward-growing stack, a trap occurs when a push stores into the last word of the segment. As a result, there is always an unused word at the end of the segment. Conversely, in a downward-growing stack, a trap occurs when popping the first word of the segment, and there must be a free word at the beginning of the segment.

Register R31 (called SP) specifies a *particular* upward-growing stack for implicit use by interrupts, traps, and linkage instructions such as CALL, JSR and ALLOC. The instructions in this section can operate on that stack, but can operate equally well on additional stacks specified by arbitrary stack pointers.

PUSH — Push onto designated stack

PUSH . {UP,DN} . {Q,H,S,D} — XOP

Purpose: Push OP2 onto the upward-growing or downward-growing stack designated by the stack pointer OP1.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: OP1 is a singleword; OP2 has the precision of the modifier.

The following pushes RTA on the stack designated by stack pointer SPL:

```
PUSH.UP.S SP,RTA
```

POP — Pop from designated stack

POP . {UP,DN} . {Q,H,S,D} **XOP**

Purpose: From the upward-growing or downward-growing stack designated by the stack pointer OP1, pop the top value (whose precision is specified by the second modifier) and store that value in OP2.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: OP2 has the precision of the modifier; OP1 is a singleword.

The following pops the top halfword on an upward-growing stack into RTA. SP is the standard stack pointer.

```
POP.UP.H SP,RTA
```

PUSHADR — Push address onto designated stack

PUSHADR . {UP,DN} **XOP**

Purpose: Compute a tagged pointer to OP2 and push that pointer onto an upward-growing or downward-growing stack specified by stack pointer OP1.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: OP1 is a singleword.

The following pushes a pointer to WHIRR onto the stack specified by a pointer at R25:

```
PUSHADR.UP R25,WHIRR
```

2.11 Routine Linkage and Traps

These instructions provide call and return mechanisms for subroutines, coroutines and trap handlers. (Additional instructions WTDBP and RTDBP, used to specify the locations for trap vectors, appear in Section 2.14.)

The architecture provides several complete sets of call and return instructions with varying degrees of sophistication. They include:

JSP, JMPRET Jump to and return from a simple subroutine without using the stack. JSP calls the subroutine, saving the return address in a specified location. JMPRET is used to indirect through that location to return.

JSR, ALLOC, RETSR, RET
Jump to and return from simple subroutines. JSR calls the subroutine, pushing a single parameter on the stack; ALLOC may be used to save registers and allocate space upon the stack; and RETSR returns from the subroutine, restoring the parameter. Alternatively, RET returns but discards the parameter pushed by JSR and, if desired, a number of words preceding it on the stack.

CALL, ENTRY, UNCALL
Call and return from an internal procedure, using a stack frame. CALL calls the procedure, ENTRY builds the stack frame, and UNCALL returns from the procedure, dropping back to the preceding stack frame.

CALLX, ENTRY, RETS, UNCALL
Call and return from an external procedure, using a stack frame. CALLX calls the procedure and ENTRY builds the stack frame. If the call crossed a ring boundary, the procedure returns with RETS.A rather than with UNCALL. See Section 1.9.5 for details on cross ring calls.

TRPSLF, RETS Cause a trap to one of the vectors for the current address space, and return from the corresponding trap handler. See Section 1.9.4 for details.

TRPEXE, RETS Cause a trap to the executive and return from the corresponding trap handler. See Section 1.9.4 for details.

JCR Jump between coroutines without using the stack.

JMPCALL, JMPRET
These are simple jump instructions which are considered to be call and return instructions for purposes of call tracing. JMPCALL may be used when a compiler converts a tail-recursive call to a jump. JMPRET may be used when a non-local goto is performed.

LCALL, LCALL0, LCALL1, LRETURN
Call and return from a LISP function.

Note call and return instructions are subject to the pointer validation described in section 1.8.4 whenever they modify the PC. In addition, the processor uses address arithmetic when it manipulates the stack pointer. In other words, a JSR instruction simulates the following instruction:

```
MOVP.P.A SP,(SP)4
```

rather than:

```
ADD.S.S SP,#4
```

On instructions like JSR which implicitly alter the stack pointer, the text will note that this can cause traps due to segment bounds violation or pointer tag validation faults; on instructions like ALLOC which explicitly evaluate an operand and move it to the stack pointer, the text will not belabor the obvious.

The following instructions will invoke the CALL_TRACE_TRAP hard trap when the call tracing mechanism in PROCESSOR_STATUS is enabled:

CALL
CALLX
JCR
JMPCALL
JMPRET
JSP
JSR
RET
RETS
RETSR
UNCALL

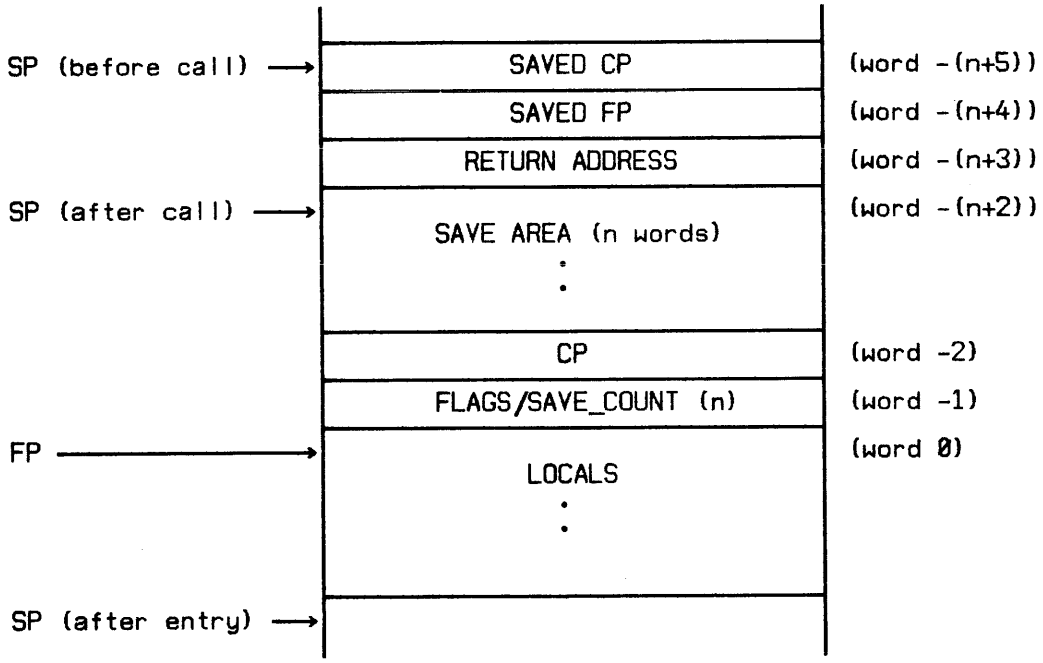
2.11.1 The Generalized Stack Frame Convention

All of the linkage instructions use the singleword register, R31, as stack pointer (SP). The CALL/ENTRY/UNCALL family of instructions establish a stack frame convention which further defines R29 to be a *closure pointer* (CP), defines R30 to be a *frame pointer* (FP), and defines a specfic stack frame format.

CP The closure pointer points to the stack frame for the procedure which is immediately global to the one which is currently executing. In Pascal, this is the procedure (or main program) inside which the currently executing procedure was declared. This pointer establishes the static scope of a language.

FP The frame pointer points to the stack frame for the currently executing procedure.

The format of the stack frame of an executing procedure is shown below. Notice that the FP points past a variable size header to the first word that may be occupied by program variables.



The copy of CP located in word -2 points acts as a link in the *display chain* of the enclosing static scopes. This copy is always at a fixed offset relative to the FP so that it can be found by procedures at a deeper level of scope. When a procedure calls another, the value of the CP is saved on top of the stack where it can be restored when the callee returns.

The FLAGS/SAVE_COUNT word contains the number of words in the save area in the least significant bits, bits 30 . . 35; the high order bits are zeroed on entry and may be used by the program to contain special flags.

To illustrate the stack frame convention, consider the following fragment of a Pascal program:

```
PROCEDURE A;
   VAR A1, A2, A3;
   PROCEDURE C;
      VAR C1, C2, C3;
      BEGIN
      ...
      END (* C *);
   PROCEDURE B;
      VAR B1, B2, B3;
      BEGIN
      C;
      ...
      END (* B *);
   BEGIN
   B;
   ...
   END (* A *);
```

Suppose that someone calls procedure A, which calls procedure B, which in turn calls procedure C. We stop the processor some time after C begins to execute, but before it has called any further procedure. Following the stack frame convention, Figure 2-2 shows the appearance of the stack.

The CALL and CALLX instructions push the caller's CP, FP and return address onto the top of the stack, and load the CP for the called routine. The ENTRY instruction allocates the stack frame, saves a specified number of registers, and initializes the stack frame. UNCALL pops the stack, returns from the called procedure and restores the caller's CP and FP. Figure 2-2 gives an example of the the instructions required for procedure B to call procedure C:

Within procedure B:

```
CALL CP,C                ; Call C, giving it the same
NI: ...                  ; CP as B because both are
                         ; nested in A. The old CP, FP
                         ; and address of NI are pushed
                         ; onto the stack
```

At the beginning of procedure C:

```
C: ENTRY.29 %0,(SP)4*<29.+2+FrameSize>  ; Save the registers, the new CP
                                        ; and the count; set FP to point
                                        ; to the new frame at SP+4*(29.+2),
                                        ; and SP to the address of OP2
                                        ; allocating the stack frame
<Code of procedure C>
MOVMS.29 %0,(FP)-4*<29.+2>              ; restore the registers
UNCALL (FP)-4*<29.+2+1>,(FP)-4*<29.+3+2>; Restore FP and CP, pop the stack
                                        ; and return
```

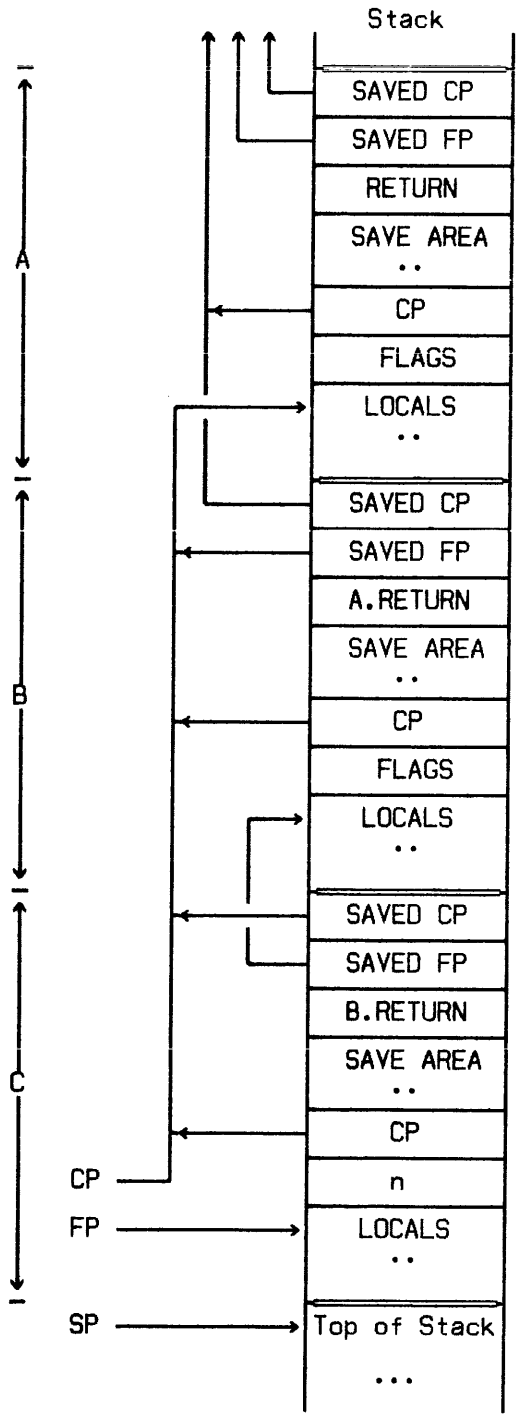


Figure 2-2
Stack Frame Illustration

2.11.2 The LISP Stack Frame Convention

The stack frame format defined by the LISP environment is as follows:

LF.RETURN_VALUE
This is a singleword slot that is assigned the function return value upon return from the function. It is at the top of the frame so that appears to be a value left at the top of the stack.

LF.SPARE_PC Used in esoteric cases of function returns.

LF.OLD_CP This contains the CP of the caller of the current function.

LF.OLD_FP This contains the FP of the caller of the current function

LF.RETURN_PC
This contains the return address within the calling function.

LF.FUNCTION This contains the CP of the current function.

LF.ARGS This is the base of the argument list for the current function.

In a LISP function, the FP points 32 singlewords past the start of the stack frame header. This permits the maximum addressibility of data within the frame with short operand format addressing.

2.11.3 Routine Linkage Instructions

CALL — Call an internal procedure

CALL — JOP

Purpose: Call an internal procedure, assuming the use of the standard stack frame. First, push CP, FP and PC_NEXT_INSTR (the return address) onto the stack pointed to by SP. Then set CP to OP1. Last, transfer control to JUMPDEST.

Restrictions: None

Exceptions: An OUT_OF_BOUNDS hard trap occurs if the instruction causes the SP to cross a segment boundary.

Precision: OP1 is a singleword; OP2 is a jump destination.

Suppose a procedure named C is declared within a procedure named B. The following sequence would call C from B:

```
MOVP.P.A R2,Parmlist        ; Pointer to parameters
CALL FP,FirstC              ; Call C. Use B's FP as C's
                            ; CP because C is nested
                            ; within B
```

CALLX **Call an external procedure**

CALLX **XOP**

Purpose: Call an external procedure, assuming the use of the standard stack frame. If the pointer, OP2, has a gate tag, perform a cross-ring call through a gate (see Section 1.9.5); otherwise, perform a normal call. First, push CP, FP and PC_NEXT_INSTR (the return address) onto the stack pointed to by SP. Then set CP to OP1. Last, transfer to the location pointed to by OP2.

Restrictions: None

Exceptions: An OUT_OF_BOUNDS hard trap occurs if the instruction causes the SP to cross a segment boundary.

Precision: OP1 is a singleword; OP2 is a pointer

Assume that a procedure has been passed as a parameter to the current routine, and that the two singlewords at (AP)0 are a pointer to the code for that procedure, followed by its closure pointer. To invoke the procedure, the current routine would execute:

```
CALLX (AP)1*4,(AP)0*4
```

ENTRY — Initialize a stack frame

ENTRY . {0 .. 32} **XOP**

Purpose: Initialize a standard stack frame. This pushes a specified number of words on the SP stack, initializes the stack frame header, and adjusts the stack pointer to allocate space on the top of the stack.

More specifically, the instruction first moves a block of 0 .. 32 singlewords (which may lie in registers) starting with OP1 to the vector pointed to by SP (if the source and destination for this move-operation overlap, the result is undefined). Following the block it stores, in order, the CP and the count of the number of words in the block (equal to the modifier). Then, it sets FP to the address of the word following the count word. This establishes the location of the new stack frame. Last, it sets SP to the address of OP2. One typically chooses OP2 to be a location beyond the last of the words moved, though this is not required.

Restrictions: None

Exceptions: An OUT_OF_BOUNDS hard trap occurs if the instruction would cause the SP to cross a segment boundary.

Precision: OP1 is a vector of singlewords, and OP2 must have singleword alignment.

The following sequence saves 8 registers starting at RTA, and initializes the stack frame.

```
ENTRY.8 RTA, (SP)4*<8+2+FrameSize>
```

UNCALL **Return from a call**

UNCALL **XOP**

Purpose: Return from a procedure called by the CALL or CALLX instruction. CP:=OP1[-2*4]; FP:=OP1[-1*4]; SP:=ADDRESS(OP2); goto the location pointed to by OP1[0]. Under normal usage, OP1 is the last word pushed by a CALL or CALLX instruction. When the stack is to be popped, OP2 is the first word pushed, i.e. the old top of stack.

Use RETS.A, not UNCALL, to return from cross-ring calls.

Restrictions: None

Exceptions: If the instruction causes SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Precision: OP1 and OP2 are singlewords.

The following sequence returns from a subroutine that saved eight registers on entry.

```
MOVMS.8 RTA, (FP)-4*<8+2>
UNCALL  (FP)-4*<8+2+1>, (FP)-4*<8+2+3>
```

JSR **Jump to subroutine**

JSR **JOP**

Purpose: Push first OP1 and then the return address onto the stack whose pointer is SP. Then transfer to JUMPDEST.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: All operands are singlewords.

The following pushes RTA and ADDRESS(F01) on the stack before jumping to BAZ:

```
        JSR RTA,BAZ
F01:    ...             ;return address
```

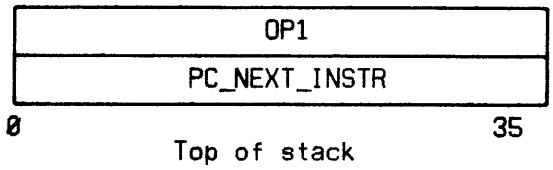


Figure 2-3
JSR Save Area Format

ALLOC — Allocate space atop stack

ALLOC . {1 .. 32} **XOP**

Purpose: This instruction pushes a specified group of singlewords onto the SP stack (the one used by the subroutine calling mechanism) and then adjusts the stack pointer, possibly allocating additional space atop the stack. Typically it is used to save registers and make room for a stack frame.

More specifically, the instruction first moves a block of 1 . . 32 singlewords (which may lie in registers) starting with OP1 to the vector pointed to by SP (if the source and destination for this move-operation overlap, the result is undefined). Then it effectively does a MOVP.P.A SP,OP2. Thus, one typically chooses for OP2 to be a location beyond the last of the words moved, though this is not required. If OP1 is a constant, ALLOC will spread its value through the specified locations.

Restrictions: None

Exceptions: If ALLOC would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Precision: OP1 and OP2 must be singlewords. OP1 is a block and can lie in the registers.

The following saves all the registers and reserves an additional DW on the stack as well:

```
ALLOC.32 R0,(SP)<4*(40+2)>
```

Note that the modifier is a *decimal* number, but the numbers in the operands are *octal*. The same instruction could be written:

```
ALLOC.32 R0,(SP)<4*(32.+2)>
```

RETSR — Return from subroutine

RETSR — XOP

Purpose: Return from a subroutine that was invoked by the JSR instruction. First the instruction copies ADDRESS(OP2) into SP. Then it pops the first singleword (return address) from the stack pointed to by SP and stores it in the PC. Then it pops the second singleword (typically the value of OP1 placed there by the JSR instruction) and stores it in OP1.

To be sure that RETSR is the exact reverse of JSR, the programmer must use the same OP1 in both JSR and RETSR, and assure that OP2 in the RETSR instruction is the same memory location that SP pointed to immediately after the JSR. If the subroutine does not alter SP, then OP2 should be "(SP)"; otherwise, the subroutine should save a stack marker and use it as OP2.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: All operands involved are singlewords.

The following code calls BAZ, which returns to F01, saving and restoring RTA on the stack. Assume SP is the stack pointer:

```
        JSR RTA,BAZ
F01:    ...             ;return here

BAZ:    ...             ;called routine
        RETSR RTA,(SP)
```

Suppose that BAZ needs N words of temporary stack space while it is running. These words can be allocated using the MOVP or ALLOC instructions, and the RETSR instruction can automatically discard these words and pop the JSR save area as well:

```
BAZ:    ALLOC.2 R8,(SP)<N+2>*4  ;save R8 and R9, and allocate N words
        ...                     ;called routine
        MOVMS.2 R8,(SP)-<N+2>*4 ;restore registers R8 and R9
        RETSR RTA,(SP)-<N+2>*4  ;pop stack and return from subroutine
```

RET — Return and pop parameters

RET — XOP

Purpose: Return without restoring parameters. First the instruction makes SP point to OP2. Then it copies one singleword (the return address) to the PC from the top of the stack pointed to by SP. Then it makes SP point to OP1, thereby optionally popping and discarding the return address and parameters (such as the one pushed onto the stack by the JSR instruction) as well.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: All operands involved are singlewords.

The following returns from a previous JSR call, throwing away the operand previously pushed on the stack by the JSR:

```
RET (SP)-4,SP
```

TRPSLF — Trap to self

TRPSLF . { 0 .. 63 } **XOP**

Purpose: Trap to a routine in the current address space. The operation of TRPSLF is explained in detail in Section 1.9.4; briefly, the modifier selects one of 64 trap vectors each of which specifies a handler address. Each operand is evaluated as a normal XOP and passed to the handler in the stack entry's parameter area.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: Unspecified

The following causes a trap to the "number 0" trap routine in the current address space, passing to it the operands X and Y:

```
TRPSLF.0 X,Y
```

TRPEXE — Trap to executive

TRPEXE . { 0 .. 63 } **XOP**

Purpose: Trap to an executive routine. The operation of TRPEXE is explained in detail in Section 1.9.4; briefly, the modifier selects one of 64 trap vectors each of which specifies a handler address. Each operand is evaluated as a normal XOP and passed to the handler in the stack entry's parameter area.

Restrictions: None

Exceptions: Performs bounds checking and tag validation on the stack pointer.

Precision: Unspecified

The following causes a trap to the "number 0" trap routine in the executive's address space with operands X and Y:

```
TRPEXE.0 X,Y
```

RETS — Return from trap

RETS . {R,A} **XOP**

Purpose: This instruction is used to return from all traps, interrupts and gates. It is more fully described in section 1.9.2. The entire trap mechanism is explained in section 1.9.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a vector of singlewords.

The following shows the return from a trap handler. The pseudoregister ((SP)-4)0 specifies the last word of the stack entry, which contains a pointer to the first word of the entry:

```
(code to handle the trap without altering SP)
RETS.R ((SP)-4)0
```

JCR **Jump to coroutine**

JCR **XOP**

Purpose: The instruction first exchanges OP1 (usually register SP) with OP2 (usually a memory location holding a saved copy of the value of SP used by the other coroutine). Then it copies the saved "return address" from NEXT(OP2), stores PC_NEXT_INSTR in NEXT(OP2), and branches to the return address.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords.

When each of two coroutines has its own distinct stack, the JCR instruction transfers between them without using either stack. Instead, it stores the stack pointer and program counter for the currently inactive coroutine in two consecutive singlewords pointed to by OP2. In the following example, let SAVE.AREA be the first of those two singlewords. Then the following instruction saves the stack pointer and PC for the current routine, sets up the stack pointer and PC for the other routine, and branches to it.

```
JCR SP,SAVE.AREA        ;call other coroutine
```

JSP **Jump and save PC**

JSP **JOP**

Purpose: First OP1:=PC_NEXT_INSTR, then go to JUMPDEST.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following saves the return address in R0 and calls PRSTR:

```
JSP R0,PRSTR
```

JMPCALL, JMPRET — Jump to call/return

JMPCALL	JOP
JMPRET	JOP

Purpose: These instructions are identical with the JMPA instruction, except that JMPCALL is considered to be a call instruction and JMPRET is considered to be a return instruction when call tracing is enabled.

LCALL — LISP function call

LCALL — XOP

Purpose: Call a LISP function. This instruction saves the caller's state, creates a new stack LISP stack frame for the callee, and transfers to it.

First, RTA := OP1; this sets the call type and argument count for the call. Next initialize the stack frame header; OP2 is the location 32 singlewords past the actual header. Store CP, FP and PC_NEXT_INSTR in LF.OLD_CP, LF.OLD_FP and LF.RETURN_PC, respectively; then set FP to ADDRESS(OP2) to address the new frame. Finally, set CP to the contents of LF.FUNCTION, and fetch the new PC from the location addressed by this new CP.

The LCALL0 and LCALL1 may be used for functions with zero or one argument, respectively.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords.

The following sequence calls a LISP function with two arguments:

```
ALLOC.5 #0,(SP)4*5       ; zero the header of the frame
PUSH SP,FUNCTION         ; push the CP of the function to call
PUSH SP,ARG1             ; push the arguments
PUSH SP,ARG2
LCALL #2,(SP)4*<32.-8>   ; make the call
```

LCALL0 **LISP function call with no arguments**

LCALL0 **XOP**

Purpose: Call a LISP function with no arguments.

First, RTA := 0; this sets the call type and argument count for the call. Next initialize the stack frame header; SP points to the start of the actual header. Zero LF.RETURN_VALUE and LF.SPARE_PC. Store CP, FP, PC_NEXT_INSTR and OP1 in LF.OLD_CP, LF.OLD_FP, LF.RETURN_PC and LF.FUNCTION, respectively. Allocate the new frame with FP := SP + 4*32 and SP := SP + 4*6 (using pointer arithmetic). Finally, set CP to OP1 and fetch the new PC from the location addressed by this new CP.

Restrictions: None

Exceptions: An OUT_OF_BOUNDS hard trap occurs if SP is within 32 singlewords of a segment boundary

Precision: OP1 is a singleword.

The following sequence calls a LISP function with no arguments:

```
LCALL0 FUNCTION
```

LCALL1 — LISP function call with one argument

LCALL1 — XOP

Purpose: Call a LISP function with one argument.

First, RTA := 1; this sets the call type and argument count for the call. Next initialize the stack frame header; SP points to the start of the actual header. Zero LF.RETURN_VALUE and LF.SPARE_PC. Store CP, FP, PC_NEXT_INSTR, OP1 and OP2 in LF.OLD_CP, LF.OLD_FP, LF.RETURN_PC, LF.FUNCTION and LF.ARGS[0], respectively. Allocate the new frame with FP := SP + 4*32 and SP := SP + 4*7 (using pointer arithmetic). Finally, set CP to OP1 and fetch the new PC from the location addressed by this new CP.

Restrictions: None

Exceptions: An OUT_OF_BOUNDS hard trap occurs if SP is within 32 singlewords of a segment boundary

Precision: OP1 is a singleword.

The following sequence calls a LISP function with one argument:

```
LCALL1 FUNCTION,ARG
```

LRETURN **LISP function return**

LRETURN **XOP**

Purpose: Return from a LISP function. This instruction restores the caller's state.

OP1 is LF.SPARE_PC. Load R28 from LF.SPARE_PC; CP from LF.OLD_CP; FP from LF.OLD_FP. Set SP to ADDRESS(OP1). Transfer to the location pointed to by LF.RETURN_PC.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following returns from a LISP function called with LCALL, LCALL0, or LCALL1:

```
LRETURN (FP)-4*31
```

2.12 Interrupts and I/O

See Sections 1.9 and 1.10 for explanations of the interrupt and input/output mechanisms.

The {B,Q,H,S} modifiers that appear on certain instructions refer to bitwise, quarterword, halfword, and singleword translations, which are likewise explained in Section 1.10.

IOR — Read with I/O translation

IOR . {Q,H,S,D,LSB16,MSB16,LSB32,MSB32}	**XOP**
MIOR . {LSB16,MSB16,LSB32,MSB32}	**XOP**
VIOR . {Q,LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VPIOR . {Q,LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VMIOR . {LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VMPIOR . {LSB16,MSB16,LSB32,MSB32,B}	**V:=V**

Purpose: These instructions read data, performing the I/O translations described in section 1.10.1.

IOR copies a scalar from OP2, which must lie in an I/O memory, to OP1, which must lie in main memory, using the specified translation. MIOR is similar, but both its operands must lie in main memory.

VIOR and VMIOR are analogous to IOR and MIOR, but operate on vectors instead of scalars.

VPIOR and VMPIOR are analogous to VIOR and VMIOR, but interpret OP1 as a physical address rather than a virtual address.

Restrictions: For VIOR.B, SIZEREG must be a multiple of 8.

Exceptions: None

Precision: For the scalar instructions, modifiers {Q,H,S,D} indicate the precision of both operands, modifiers {LSB16,MSB16} indicate that both operands are halfwords, and modifiers {LSB32,MSB32} indicate that both operands are singlewords.

For the vector versions of these instructions, both operands are vectors of singlewords regardless of the modifier, and SIZEREG indicates the number of singlewords in the destination (main memory) vector (the "B" modifier can cause the destination vector to be shorter than the source vector).

Assume BUFFER is a legitimate address on an I/O page. To read eighty characters from the I/O memory (starting at BUFFER) to a block in memory starting at IMAGE, the following instruction sequence could be used:

```
MOV.S.S SIZEREG,#<80./4>        ;set SIZEREG to eighty QWs
VIOR.Q IMAGE,BUFFER             ;do read
```

IOW — Write with I/O translation

IOW . {Q,H,S,D,LSB16,MSB16,LSB32,MSB32}	**XOP**
MIOW . {LSB16,MSB16,LSB32,MSB32}	**XOP**
VIOW . {Q,LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VPIOW . {Q,LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VMIOW . {LSB16,MSB16,LSB32,MSB32,B}	**V:=V**
VMPIOW . {LSB16,MSB16,LSB32,MSB32,B}	**V:=V**

Purpose: These instructions write data, performing the I/O translations described in section 1.10.1.

IOW copies a scalar from OP2, which must lie in main memory, to OP1, which must lie in an I/O memory, using the specified translation. MIOW is similar, but both its operands must lie in main memory.

VIOW and VMIOW are analogous to IOW and MIOW, but operate on vectors instead of scalars.

VPIOW and VMPIOW are analogous to VIOW and VMIOW, but interpret OP2 as a physical address rather than a virtual address.

Restrictions: For VIOW.B, SIZEREG must be a multiple of 8.

Exceptions: None

Precision: For the scalar versions of these instructions, the modifiers {Q,H,S,D} indicate the precision of both operands, the modifiers {LSB16,MSB16} indicate that both operands are halfwords, and the modifiers {LSB32,MSB32} indicate that both operands are singlewords.

For the vector versions of these instructions, both operands are vectors of singlewords regardless of the modifier, and SIZEREG indicates the number of singlewords in the source vector (the "B" modifier can cause the destination vector to be longer than the source vector).

Assume BUFFER lies within an I/O page. To transfer the four characters "S-1!" into the IOBUF starting at BUFFER the following instructions could be used:

```
MOV.S.S SIZEREG,#<4/4>          ;make vector 4 characters long
VIOW.Q BUFFER,#["S-1!"]         ;do write
```

Because no translation is required the following instruction would work just as well:

```
IOW.S BUFFER, ["S-1!"]          ;copy a singleword
```

IORMW — I/O read/modify/write

IORMW — TOP

Purpose: In one memory cycle (and hence indivisibly with respect to other processors in a multiprocessor system) DEST:=S1 and then S1:=S2. (More precisely, because the processor prefetches operands and because TOP instructions store DEST last, this instruction makes a temporary copy of S1, stores S2 in S1, and then stores the copy into DEST.)

DEST and S2 must lie in main memory. S1 must lie on an I/O page.

Restrictions: None

Exceptions: None

Precision: S1, S2, and DEST are all singlewords.

The following illustrates the use of IORMW:

```
IORMW RTA,#-1,LOCK
```

INTIOP **Interrupt I/O processor**

INTIOP **XOP**

Purpose: Interrupt the I/O processor connected to the I/O memory containing OP1, and pass OP2 to the I/O processor as a parameter whose purpose is not specified by the architecture.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords. OP1 must lie within an I/O page having WRITE_PERMIT access.

Assume BUFFER lies within an I/O page. The following instruction will interrupt the I/O processor connected to the I/O memory containing BUFFER:

```
INTIOP BUFFER,#0
```

WAIT — Wait for interrupt

WAIT **XOP**

Purpose: Cause the processor to wait for an interrupt at the processor priority specified by OP1. Changing the priority indivisibly with the WAIT instruction eliminates the problem that would occur if you intended to WAIT for an interrupt but in fact the interrupt occured after changing the priority but before executing the WAIT instruction.

Restrictions: Illegal in user mode.

Exceptions: If OP1 is not a valid processor priority an ILLEGAL_PRIORITY hard trap occurs.

Precision: OP2 is unused.

The following instruction waits for an interrupt at priority 5:

```
WAIT #5
```

RIEN — Read interrupt enable

RIEN **XOP**

Purpose: If interrupts are enabled for the I/O memory containing singleword OP2, then OP1 := -1 else OP1 := 0.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords; OP2 must lie on an I/O page.

The following jumps to DISABLED if interrupts are not enabled for the I/O memory which contains TTYMUX:

```
RIEN RTA,TTYMUX
JMPZ.EQL.S RTA,DISABLED
```

WIEN — Write interrupt enable

WIEN — XOP

Purpose: If the low order bit of OP2 is "1", enable interrupts for the I/O memory containing OP1; otherwise, disable interrupts for that I/O memory.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords. OP1 must lie on an I/O page.

The following enables all interrupts for the I/O memory containing TTYMUX:

```
WIEN TTYMUX,#1
```

RIPND — Read interrupt pending

RIPND **XOP**

Purpose: OP1 gets the priority level of the pending interrupt for the I/O memory containing OP2. (OP1=0 indicates no interrupt is pending.)

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords. OP2 must lie on an I/O page.

The following sets RTA to the level of pending interrupt for the I/O memory containing TTYMUX:

```
RIPND RTA,TTYMUX
```

WIPND **Write interrupts pending**

WIPND **XOP**

Purpose: If an interrupt is pending for the I/O memory containing OP1, change the priority of the interrupt to the level specified by OP2. If not, cause an interrupt with priority specified by OP2 on behalf of the I/O memory containing OP1 (whether the interrupt occurs immediately or remains pending depends, as always, on the relative priority of the uniprocessor). If OP2=0, the instruction effectively clears any pending interrupt for the I/O memory in question.

Restrictions: Illegal in user mode.

Exceptions: If OP2 is not a valid level, an ILLEGAL_PRIORITY hard trap occurs.

Precision: OP1 and OP2 are both singlewords. OP1 must lie on an I/O page.

The following clears any pending interrupt for the I/O memory which contains TTYMUX:

```
WIPND TTYMUX,#0
```

2.13 Cache Handling

The S-1 uniprocessor has four caches: an instruction cache, a data cache, an instruction map cache, and a data map cache. The first two hold recently used words from address spaces, and the latter two hold recently used entries from the virtual-to-physical address mapping tables (described in Section 1.7).

If the uniprocessor accesses memory to fetch an instruction, then that access involves the instruction cache and the instruction map cache. If the access reads or writes a piece of data, then it involves the data cache and the data map cache. If the ACCESS bits for a particular page specify EXECUTE_PERMIT as well as READ_PERMIT or WRITE_PERMIT, then conceivably one could, by alternately reading (or writing) a location and executing it, cause that location to appear in both the instruction cache and the data cache; no problems need result. (In the more likely situation where the ACCESS bits are used to enforce separation of instructions and data, such a situation would not occur.)

In general, the caches employ a least recently used (LRU) algorithm to decide which cache residents to evict to make room for new residents. Not every instruction causes its operands to be regarded as used, however. I/O instructions do not update the LRU status bits for their operands, for example, since the data involved in an I/O operation is unlikely to be accessed repeatedly.

While the caches are usually invisible to software, instructions are provided to sweep them--that is, deliberately update main memory to reflect any changes in cache contents--if this is felt to improve performance. The cache sweeping instructions take ordinary operands which specify memory location on the pages to be swept; the instructions implicitly examine the *addresses* of those operands rather than the operands themselves to determine which pages to sweep.

SWPIC — Sweep instruction cache

SWPIC . {V,P} **XOP**

Purpose: Sweep the instruction cache by removing a vector of consecutive singleword residents without writing them back to main memory. (Since access to an instruction page prevents writing, the contents of the cache cannot differ from the corresponding portions of main memory.) OP1 is the vector.

The {V,P} modifier tells the processor how to determine which locations are "consecutive". In either case, it first evaluates OP1 as it would for any ordinary memory reference. If the modifier is V, it then sweeps the vector of words whose virtual addresses follow that of OP1. If the modifier is P, it sweeps the vector of words whose physical addresses follow that of OP1.

Restrictions: Physical sweeps are legal only in privileged mode.

Exceptions: None

Precision: OP1 is a vector of singlewords. OP2 is unused.

The following sweeps all instructions from START up to but not including the following instructions:

```
MOV.S.S SIZEREG,<.-START+3>/4   ;specify the length of the vector
SWPIC.V START                   ;sweep cache
```

SWPDC — Sweep data cache

SWPDC . {V,P} . {U,UK} **XOP**

Purpose: Sweep the data cache by writing a vector of consecutive singleword residents back to main memory. If the second modifier is U, merely update main memory; if it is UK, update main memory and then remove the specified residents from the cache ("kill" them). OP1 is the vector.

The {V,P} modifier tells the processor how to determine which locations are "consecutive". In either case, it first evaluates OP1 as it would for any ordinary memory reference. If the modifier is V, it then sweeps the vector of words whose virtual addresses follow that of OP1. If the modifier is P, it sweeps the vector of words whose physical addresses follow that of OP1.

Restrictions: Physical sweeps are legal only in privileged mode.

Exceptions: None

Precision: OP1 is a vector of singlewords. OP2 is unused.

The following updates the first 128 quarterwords in the address space, without removing them from the data cache (i.e., not killing them):

```
MOV.S.S SIZEREG,#128.    ;specify the vector length
SWPDC.V.U 0              ;sweep cache
```

SWPIM, SWPDM, FLSHIM, FLSHDM — Sweep/flush instruction/data map cache

SWPIM	**XOP**
SWPDM	**XOP**
FLSHIM	**XOP**
FLSHDM	**XOP**

Purpose: Sweep a map cache, removing one resident, or flush a map cache, removing all residents.

SWPIM removes from the instruction map cache the entry for the page containing OP1. SWPDM removes from the data map cache the entry for the page containing OP1.

FLSHIM removes all entries from the instruction map cache. FLSHDM removes all entries from the data map cache.

None of these instructions update main memory.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: For SWPIM and SWPDM, OP1 is a singleword and OP2 is unused. For FLSHIM and FLSHDM, OP1 and OP2 are unused.

The following kills the instruction map entry for the first page in the user's address space:

```
SWPIM 0
```

The following kills the data map entry for the page containing the memory location pointed to by RTA:

```
SWPDM (RTA)
```

2.14 Context (Map, Register Files, and Status Registers)

This section describes a number of instructions which an operating system can use to set up the proper environment for a task. They manipulate the user and processor status registers, the multiple sets of user registers, the mapping system, and the origin of trap, interrupt, and gate vectors. Sections 1.2.3, 1.4, 1.7, and 1.9 explain details of these features of the architecture.

Address Space IDs: In a multiprogramming environment, it is likely that various tasks will alternately use the same virtual addresses but different mappings to the physical address space--in other words, that the operating system could keep multiple tasks in various regions of physical memory and switch between them by changing the virtual-to-physical address mapping tables. The operating system would have to sweep the map caches before switching from one task to the next to prevent the new task from being affected by mapping information left in the caches by the old one. To obviate this time-consuming process, the operating system can specify via the SWITCH instruction a different code, called an address space ID, for each task. The caching mechanism combines this code with virtual address references made by that task, rendering them unique from virtual address references made by other tasks. Thus, for example, a reference to virtual address 1000 in ring 3 with address space ID 5 is distinct from a reference to virtual address 1000 in ring 3 with address space ID 20; the mapping information for both of these may reside in cache simultaneously and can provide two different address transformations. It is the responsibility of the operating system never to specify the same ID for two different tasks which use the same address space unless it sweeps the map caches between instances of the two tasks.

An address space ID must not be set to the value 0, which is reserved for use by the hardware.

SWITCH **Switch context**

SWITCH XOP

Purpose: OP1 is a vector describing the state of a task to be run. The instruction loads the appropriate internal registers with the information from this vector and resumes execution (restarting an interrupted instruction if INSTRUCTION_STATE so demands.)

The vector contains the following information:

Singleword	Information
0	DSEGP
1	Address space ID for ring 0
2	Address space ID for ring 1
3	Address space ID for ring 2
4	Address space ID for ring 3
5	PROCESSOR_STATUS
6	USER_STATUS
7	PC
8	SIZE of INSTRUCTION_STATE
9 . . .	INSTRUCTION_STATE

Address space IDs are explained in Section 2.14. The DSEGP is explained in Section 1.7.

Restrictions: Illegal in user mode. OP1 cannot be a register or constant.

Exceptions: None

Precision: OP1 is the first element of a vector of singlewords. OP2 is unused.

Start executing the task described in the vector beginning at NextTask:

```
SWITCH NextTask
```

WASJMP — Write address space and jump

WASJMP — JOP

Purpose: OP1 is a vector describing a particular mapping of four virtual address spaces onto the physical address space. The instruction loads the DSEGP and address space IDs from this vector, thereby causing the address translation mechanism to adopt this mapping, and resumes execution at JUMPDEST (where JUMPDEST is translated according to the newly established mapping).

The vector contains the following information:

Singleword	Information
0	DSEGP
1	Address space ID for ring 0
2	Address space ID for ring 1
3	Address space ID for ring 2
4	Address space ID for ring 3

Address space IDs are explained in Section 2.14. Note that you must not set an address space ID to 0 since that value is reserved for use by the hardware.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is the first element of a vector of singlewords.

Tell the address translation mechanism to use the mapping specified by NewMap, and resume execution at NewProcess:

```
WASJMP NewMap,NewProcess
```

RRFILE **Read register file identity**

RRFILE **XOP**

Purpose: OP1:=PROCESSOR_STATUS.REGISTER_FILE, right justified and padded with zeros. This instruction tells *which* register file is in use.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

Set RTA to the number (in the range 0 .. 15) of the current register file:

```
RRFILE  RTA
```

WRFILE — Write register file identity

WRFILE **XOP**

Purpose: PROCESSOR_STATUS.REGISTER_FILE:=OP1. This instruction selects *which* register file to use. If OP1 is not within the range 0 . . 15 an ILLEGAL_REGISTER hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

Select register file number 2:

```
WRFILE  #2
```

WREGFILE — Write register file

WREGFILE — XOP

Purpose: OP1 is a singleword specifying a register file. The instruction copies vector OP2, which is 32 singlewords long, into that register file.

Restrictions: Illegal in user mode. OP2 cannot be a register.

Exceptions: If OP1 is outside the range 0 . . 15, an ILLEGAL_REGISTER hard trap occurs.

Precision: OP2 is a vector of 32 singlewords. OP1 is a singleword.

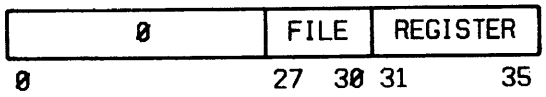
Initialize register file 7 using 32 singlewords popped from the stack pointed to by ANSP:

```
WREGFILE #7,(ANSP)<-32.*4>
MOVP.P.A ANSP,(ANSP)<-32.*4>
```

RREG — Read register

RREG **XOP**

Purpose: OP2 is a singleword specifying a register within a particular register file. The instruction copies that register into OP1. The format of OP2 is:



where FILE is in the range 0 . . 15 and REGISTER is in the range 0 . . 31.

Restrictions: Illegal in user mode.

Exceptions: If OP2 is invalid, an ILLEGAL_REGISTER hard trap occurs.

Precision: Both operands are singlewords.

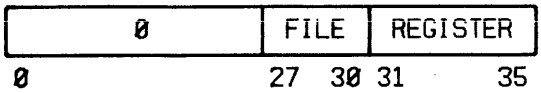
Copy the version of R4 in register file 7 into the current RTA:

```
RREG RTA,#<32.*7+4>
```

WREG — Write register

WREG — XOP

Purpose: OP1 is a singleword specifying a register within a particular register file. The instruction copies OP2 into that register. OP1 has the following format:



where FILE is in the range 0 . . 15 and REGISTER is in the range 0 . . 31.

Restrictions: Illegal in user mode.

Exceptions: If OP1 is invalid, an ILLEGAL_REGISTER hard trap occurs.

Precision: Both operands are singlewords.

Copy the current register R3 into the version of register R3 in register file 7 (note that this involves register 3, not the PC):

```
WREG #<32.*7+3>,R3
```

RPS

Read processor status

XOP

RPS

Purpose: OP1:=PROCESSOR_STATUS

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following copies PROCESSOR_STATUS into RTA:

```
RPS RTA
```

WFSJMP — Write full status and jump

WFSJMP — JOP

Purpose: PROCESSOR_STATUS:=FIRST(OP1); USER_STATUS:=SECOND(OP1).

Restrictions: Illegal in user mode.

Exceptions: An ILLEGAL_PROCESSOR_STATUS or ILLEGAL_USER_STATUS hard trap will occur if an illegal value of PROCESSSOR_STATUS or USER_STATUS is specified, respectively.

Precision: FIRST(OP1) and SECOND(OP1) are singlewords.

The following sets PROCESSOR_STATUS to FIRST(NEWPST), sets USER_STATUS to SECOND(NEWPST) and jumps to BRAZIL:

```
WFSJMP NEWPST,BRAZIL
```

RUS — Read user status

RUS **XOP**

Purpose: OP1:=USER_STATUS. OP2 is unused.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following loads RTA from USER_STATUS:

```
RUS RTA
```

JUS — Jump on selected user status bits

JUS . {NON,ALL,ANY,NAL} JOP

Precision: If USER_STATUS LCOND OP1 (where LCOND∈{NON,ALL,ANY,NAL}) is true, control is transferred to the location specified by JUMPDEST.

Restrictions: None

Exceptions: None

Precision: All operands concerned are singlewords.

Let ERRORS be a mask for several bits in USER_STATUS. The following jumps to ZIP if any of these bits are set:

```
JUS.ANY ERRORS,ZIP
```

JUSCLR — Jump on selected user status bits and clear

JUSCLR . {NON,ALL,ANY,NAL} **JOP**

Purpose: OP1 is a mask for selecting bits from USER_STATUS. The instruction first tests those bits using the condition specified by the modifier. Then it clears those bits. Finally, if the test yielded true, the processor jumps to JUMPDEST.

Formally:

```
TEMP:=USER_STATUS;
(* ¬ represents one's complement *)
USER_STATUS:=USER_STATUS∧(¬OP1);
If TEMP {NON,ALL,ANY,NAL} OP1 THEN GOTO JUMPDEST;
```

Restrictions: None

Exceptions: An ILLEGAL_USER_STATUS hard trap will occur if clearing the specified bits would produce an illegal value for USER_STATUS.

Precision: All operands are singlewords.

Let ZDIV be the mask for the INT_Z_DIV bit in USER_STATUS. The following jumps to YOW and clears this bit if it is set:

```
JUSCLR.ALL ZDIV,YOW
```

SETUS, SETPS — Set specified user or processor status bits

SETUS **XOP**
SETPS **XOP**

Purpose: USER_STATUS:=(USER_STATUS AND NOT(OP2)) OR (OP1 AND OP2) (where AND, NOT, and OR are bitwise operations). Note that an ILLEGAL_USER_STATUS, ILLEGAL_PROCESSOR_STATUS hard trap will occur if an illegal value is specified. SETPS is identical to SETUS but effects PROCESSOR_STATUS

Restrictions: SETPS illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are singlewords.

The following sets the low order bit in USER_STATUS:

```
SETUS #1,#1
```

RRNDMD, WRNDMD — Read/write rounding mode

RRNDMD . {INT,FLT} **XOP**
WRNDMD . {INT,FLT} **XOP**

Purpose: RRNDMD reads the rounding mode specified by the modifier into OP1. WRNDMD sets the rounding mode specified by the modifier from the rightmost 5 bits of OP1; if OP1 contains bits outside that field, the result is undefined.

The modifier has the following meaning:

INT	USER_STATUS.INT_RND_MODE
FLT	USER_STATUS.FLT_RND_MODE

See Section 2.2.5 for a description of rounding modes.

Note that when the integer rounding mode is set, it stays in effect until the first instruction that uses integer rounding (e.g. MDIV or FIX). All such instructions reset the integer rounding mode to diminished-magnitude.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following jumps to ISFLOOR if floor rounding is specified by USER_STATUS.FLT_RND_MODE. Otherwise, it selects ceiling rounding:

```
FLOOR=0
CEILING=4
RRNDMD.FLT RTA
SKP.EQL.S RTA,#FLOOR,ISFLOOR
WRNDMD #CEILING
```

SETPRI — Set processor priority

SETPRI — XOP

Purpose: OP1 is set to the processor priority; then the processor priority is set to OP2.

Restrictions: None

Exceptions: Illegal in user mode.

Precision: OP1 and OP2 are singlewords.

The following sets the processor priority to 1 saving the old priority in RTA:

```
SETPRI RTA,#1
```

WUSJMP **Write user status and jump**

WUSJMP **JOP**

Purpose: USER_STATUS:=OP1. Control is then transferred to the location specified by JUMPDEST.

Restrictions: None

Exceptions: An ILLEGAL_USER_STATUS hard trap will occur if an illegal value of USER_STATUS is specified.

Precision: All operands concerned are singlewords.

The following sets the USER_STATUS to NEWUS and jumps to AWAY:

```
WUSJMP NEWUS,AWAY
```

RTDBP, WTDBP — Read and write TDBP

RTDBP **XOP**
WTDBP **XOP**

Purpose: These instructions read and write the trap descriptor block pointer, the register which specifies the origin of a table which in turn specifies the origins of each set of trap, interrupt, and gate vectors.

RTDBP loads into OP1 the 34-bit physical address stored in TDBP. WTDBP loads into TDBP the rightmost 34 bits of OP1.

The effect of altering the trap descriptor block without executing a WTDBP instruction is undefined.

Restrictions: Illegal in user mode.

OP1 may not be a register or a constant (i.e. consider it to be a one word long vector).

Exceptions: None

Precision: OP1 is a singleword. WTDBP may not be a register or constant. OP2 is unused.

The following specifies that the trap descriptor block begins at the first singleword of physical memory:

```
WTDBP [0]
```

2.15 Performance Evaluation

For details on the performance counters, see section 1.9.9.

RCTR **Read counter**

RCTR **XOP**

Purpose: OP2 is a counter number. OP1 gets the contents of the counter specified by OP2.

Restrictions: None

Exceptions: None

Precision: OP1 is a doubleword. OP2 is a singleword.

The following sets RTA (DW) to the current real-time cycle count:

```
RCTR RTA,#0
```

WCTR — Write counter

WCTR — XOP

Purpose: OP1 is a counter number. Write OP2 into the counter specified by OP1.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is a doubleword.

The following zeros the real-time cycle counter:

```
WCTR #0,#0
```

2.16 Program Debugging Tools

The processor has separate instruction and data *breakpoint tables* that provide a capability to trap on references to 16 instruction addresses and 16 data addresses. It also has bit maps that can be used to selectively enable these address breakpoints.

The processor also maintains four separate *flow tables* that record the addresses of the instructions being executed. This provides a partial execution history of the program. The choice of flow table is determined by PROC_STATUS.FLOW_TABLE. The tables hold up to 256 entries each.

WIBPTM, WDBPTM — Load break point masks

WIBPTM **XOP**
WDBPTM **XOP**

Purpose: These use the least significant 16 bits of OP1 as a mask to specify which entries in the specified break point table are to be enabled. If bit N+20 of the mask is set, then a match against table entry N causes a DATA_BREAK_POINT or INSTRUCTION_BREAK_POINT hard trap. The LIBPTM sets the mask for the instruction break point table; LDBPTM, for the data break point table.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword.

The following instructions disable the all address break point checking:

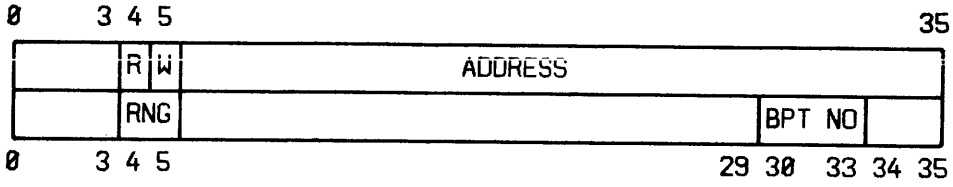
```
WIBPTM #0
WDBPTM #0
```

WIBPT, WDBPT — Load breakpoint table

WIBPT **XOP**
WDBPT **XOP**

Purpose: WIBPT and WDBPT edit the instruction and data breakpoint tables, respectively. OP1 is a vector of doublewords describing the breakpoints to be defined. The number of entries in the vector is given by SIZEREG (R3).

The format of the entries is as follows:



The number of the breakpoint is given by BPTNO and may be in the range 0 . . 15.

The fields RNG and ADDRESS give the ring number and virtual address to be matched. The address is significant only to singlewords, i.e. the low order two bits of the address are ignored. Thus, a breakpoint set for a singleword matches a quarterword or halfword reference anywhere within the singleword.

The R and W fields specify whether or not a trap is to occur on a read or write data reference; a trap occurs if the bit is set. Note that if both bits are set, a trap only occurs on a reference that both reads and writes the location--for example, in the RMW instruction or a TOP where DEST is S1. In order to catch *either* a read reference or a write reference, it is necessary to have two separate entries. The meaning of the R and W bits are undefined for WIBPT; they should be left zero.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a vector of doublewords; SIZEREG is a singleword.

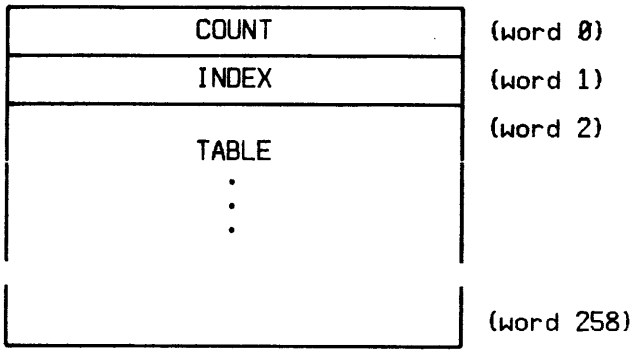
The following sequence sets three breakpoints. One is an instruction breakpoint; the other two are data breakpoints which catch any reference to a location.

```
        WIBPTM  #0              ; turn off existing entries
        WDBPTM  #0
        MOV.S.S SIZEREG,#1      ; load 1 instruction entry
        WIBPT   ITABLE
        MOV.S.S SIZEREG,#2      ; load 2 data entries
        WDBPT   DTABLE
        WIBPTM  #000000100000   ; turn on matching
        WDBPTM  #000000140000

ITABLE: addr1+100000000000      ; addr1, ring3, instruction
        060000000000
DTABLE: addr2+040000000000      ; addr2, ring2, read or write
        040000000000
        addr2+020000000000
        040000000004
```

RFLTAB — Read flow table

RFLTAB — XOP

Purpose: This reads the contents of one of the four internal flow tables. OP1 is the start of the table; OP2 gives the number of the flow table and must be in the range 0 . . 3. The format of the table is as follows:



COUNT gives the number of valid entries in TABLE. INDEX gives the index of the most recent entry in the table; INDEX-1 points to the second most recent; and so on, wrapping around at the 0th entry. Each entry in the table is a normal pointer with a ring tag.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a vector of singlewords; OP2 is a singleword.

The following reads the flow table number 2.

```
RDFLTAB TABLE,#2
```

ZFLCNT — Zero flow table count

ZFLCNT — XOP

Purpose: This empties the contents of one of the four internal flow tables. OP1 gives the number of the flow table and must be in the range 0 . . 3.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword.

The following empties flow table number 3.

```
ZFLCNT  #3
```

2.17 Miscellaneous

NOP — No operation

NOP — XOP

Purpose: NOP may have operands, but it performs no operation and stores no result. It always transfers control to the next instruction.

Restrictions: None

Exceptions: The operand addressing calculations are carried through; while the operands themselves are not referenced, an invalid addressing mode will cause a RESERVED_ADDRESS_MODE hard trap.

Precision: OP1 and OP2 may be any precision since they are not fetched.

The following three instructions are, respectively, one, two and three word NOPs:

```
NOP #0,#0
NOP #0,#[0]
NOP #[0],#[0]
```

HALT — Halt this processor

HALT — JOP

Purpose: Halt the processor. Execution continues at JUMPDEST when the halted processor continues. HALT affects only the processor that executes it. OP1 is unused.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is unused

The first instruction continues at CONT; the second halts immediately upon continuation:

```
HALT CONT
HALT .
```

RPID — Read processor identification number

RPID **XOP**

Purpose: OP1:=PROC_ID

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused

The following sets RTA to the processor ID number.

```
RPID RTA
```

3 The FASM Assembler

3.1 Commands to invoke FASM

FASM is a cross-assembler which executes on the PDP-10 and emits code for the S-1 native mode instruction set. To use it with the WAITS operating system at Stanford University, type:

```
R FASM2;<output>,<listing><-<input>
```

<input> is the name of the file containing assembly source language. The file extension defaults to "•S1" if omitted.

<output> is the file FASM puts relocatable code into. The file extension defaults to "•LDI" if omitted.

<listing> is the file FASM puts its listing into. If you omit the file extension, FASM assumes "•LST".

Alternatively, type the following and FASM will suppress the listing:

```
R FASM2;<output><-<input>
```

Or type the following and FASM will suppress the listing, putting relocatable code in a file whose name matches that of <input> but whose extension is "•LDI":

```
R FASM2;<input>
```

Or type the following and the program will prompt with "*" and wait for the rest of the command line.

```
R FASM2
```

It is possible to segment the input into several files. To assemble files IN1, IN2 and IN3, for example, type:

```
R FASM2;OUT←IN1+IN2+IN3
```

or:

```
R FASM2;OUT←IN1,IN2,IN3
```

or create a file called IN containing the line "IN1+IN2+IN3" and then type:

```
R FASM2;OUT←@IN
```

A file which, like IN, contains part of the command line is an *indirect file*. Within an indirect file a semicolon tells the program to ignore the rest of the line, including the carriage return and line feed. This allows the command to extend over more than one physical line, as the following example shows:

```
OUT←IN1+;
IN2+;
IN3
```

The first linefeed that is not ignored will cause the indirect file to be closed and command line processing to continue from where the indirect file was called. An indirect file may also call another indirect file (up to 10 levels).

Use the SNAIL commands LOAD and COMPILE to automatically run FASM and then optionally call FSIM2. The /L switch may be used with SNAIL to force FASM to make a listing.

3.2 Preliminaries

FASM makes three passes over the input file to do a good (but not perfect) job of substituting relative-JOP instructions for generally bulkier absolute-JOP instructions. During the first pass, FASM uses only absolute jumps, setting each label to the maximum possible value it will attain. During the second pass, FASM replaces absolute jumps with relative ones where possible, provided the jump destination is in instruction space only and not external. During the third pass, FASM generates the code.

FASM accepts the superset of the ASCII character set used at the Stanford Artificial Intelligence Lab (SAIL), but wherever its syntax uses special characters from the SAIL set, it also accepts substitutes from the standard ASCII set. This section will present both choices.

Because each page of S-1 memory can be marked EXECUTE_PERMIT, READ_PERMIT, and/or WRITE_PERMIT, FASM maintains separate location counters controlled by the ISPACE, DSPACE, XSPACE, IPAGE, and DPAGE pseudo-ops explained later.

Like any assembler, FASM processes statements, each of which may define a symbol, emit an S-1 instruction, or emit a dataword.

But unlike many assemblers, which simplemindedly parse lines looking for label, opcode, and operand fields, FASM starts by scanning the text character by character, expanding macros. The resulting strings go to the portion of the assembler that recognizes assembly language constructs. Many of those constructs themselves (symbol definitions, literals, pseudo-ops, and so on) return values just as functions in a high-level language do, so the programmer may embed them in expressions with considerable flexibility.

3.3 Expressions

The primary building block of a FASM statement is the expression. An expression is made up of terms separated by operators with no embedded blanks. The simplest legal expression is a single term with no operators.

Attributes: An expression may have one or more attributes. The possible attributes are: *register, instruction value (IVAL), data value (DVAL),* and *external value (XVAL).* These attributes are derived from the terms and operators that make up the expression.

A term in an expression may be a number, a symbol, a literal, a text constant or a value-returning pseudo-op.

When it encounters an expression, FASM attempts to perform the indicated operations on the specified terms. Sometimes, the value of a term is not available (for example, is undefined or is external) at the time the expression is evaluated. Sometimes this is permissible and sometimes it will cause an error. In the descriptions that follow it will sometimes be said that an expression must be defined at the time it is evaluated.

3.3.1 Operators

The following are the valid operators along with their precedences. Each is binary unless marked "(unary)".

Purpose	ASCII symbol	SAIL symbol	Precedence
Addition	+	+	1
Subtraction	-	-	1
Multiplication	*	*	2
Division	/	/	2
Bitwise OR	!	∨	3
Bitwise AND	&	∧	3
Bitwise XOR	#	≠	3
Power of 2	^	↑	4
Bitwise NOT		¬	5 (unary)
Plus	+	+	5 (unary)
Minus	-	-,∂	5 (unary)
Register attribute	%	%	5 (unary)

(Though FASM recognizes no ASCII equivalent for "¬", the programmer can achieve the effect of "¬X" by writing "<-1*X>".)

A↑B has the value of A shifted left (if B is positive) or right (if B is negative) by B bits.

The "%" symbol gives the term following it the *register* attribute (though context may override that attribute; for example, a "%5" in an expression inside a constant operand merely contributes an integer "5" to the expression which then becomes a constant.)

Each operator has a precedence which is used to determine order of association. For operations with the same precedence, association is to the left. Angle brackets <> (also known as brokets and pointy brackets) may be used to parenthesize arithmetic and logical expressions. (Parentheses "()" themselves may not be used for this purpose because they are significant for expressing various addressing modes.) A parenthesized (or rather, *broketed*) expression may take more than one line, in which case the value of the last line is used as the value of the expression. However, *all* the lines are evaluated and then all the values are thrown out except for the last one. These evaluations may have side effects like defining symbols, or executing macros, etc.

3.3.2 Numbers

A string of digits is interpreted as a number. If a "." appears at the end of the number, FASM assumes it is a decimal integer. If a "." appears inside the number, FASM assumes it is a decimal floating point number in singleword format. Otherwise, FASM assumes it is an integer in the current radix, which defaults to base 8 (octal) but may be changed with the RADIX pseudo-op. A singleword floating point number has digits on both sides of a decimal point and may be followed by an E, an optional + or -, and a one or two digit exponent, which is assumed to be a decimal number and should not have an explicit decimal point.

3.3.3 Symbols

A symbol is a one- to sixteen-character name made up from letters, numbers, and the characters ".", "_", and "$". (A symbol may actually contain more than sixteen characters, but all characters after the sixteenth are ignored.) Lower-case letters are permitted, but are considered to be the same as the equivalent upper-case characters. A symbol must not look like a number; for example, 43. is an integer and 0.1 is a floating point number, whereas 0..1, 1.E5, and 2.3E.5 are symbols (because they do not quite qualify as floating point numbers).

Following the initial character of a symbol, one may enclose in quotation marks any characters which would otherwise be forbidden. The quotation marks and the otherwise forbidden characters

all become part of the symbol. For example, the first of the following two lines is an arithmetic expression involving symbols "CAT", "A", and "DOG", whereas the second is a single symbol "CAT"*A-"DOG":

```
CAT*A-DOG
CAT"*A-"DOG
```

Symbols have values and attributes. The values are 36-bit numbers which are used in place of the symbol when it appears in an expression. The attributes are: *register, instruction value (IVAL), data value (DVAL), half-killed, external value,* and *macro name.*

If a symbol is a macro name, then instead of having a value, the symbol has a macro definition associated with it. This macro definition is expanded when the symbol is seen under certain circumstances and the expansion is used in place of the symbol in the expression. (See the section on macros for more details on macro definition and expansion.)

Predefined symbols: FASM recognizes certain symbols without requiring the programmer to define them.

"."	A lone dot represents the current location counter. It is either an IVAL or a DVAL, depending upon whether ISPACE, DSPACE, IPAGE, or DPAGE is in force. Its value is the quarterword address at which the next instruction or data will be assembled. Its default attribute is IVAL and its initial value is 0 for a relocatable assembly or 10000 octal for an absolute assembly.
RTA, RTB	RTA and RTB represent %20 and %30 respectively, so their attribute is *register.*

3.3.4 Literals

A literal is any set of assembler statements enclosed in [ ] (called square brackets) and separated by "↔", "?", or linefeeds. A literal directs the assembler to assemble the statements appearing inside the square brackets and store them at some location other than the current location counter. If embedded in an expression, the entire literal returns a value: the address at which the first singleword of the literal is assembled. There are certain restrictions on just what may appear inside a literal. Certain pseudo-ops are illegal inside of literals (see the section on pseudo-ops). Currently, labels are not permitted inside a literal, although this may change in the future. The symbol "." is not affected by the fact that it is referenced from inside a literal. It will have the value it had at the point where the literal was begun even though the literal may already have assembled some statements.

Just where the literal is assembled is determined by several factors. First it is determined whether the literal is an instruction-space or a data-space literal. This is determined in the following manner. If the next characters immediately after the [that begins the literal are !I or !D, then the literal is an instruction-space or data-space literal, respectively. If not, then the literal will be an instruction-space literal if it contains any opcodes. Otherwise it will be a data-space literal. All instruction-space literals will be assembled starting at the current location counter when a LIT pseudo-op is encountered while in instruction-space. A similar statement is true of the data-space literals. Certain other pseudo-ops cause an implicit LIT to be done first.

One typical use of a literal is to move a doubleword from data memory into register space. The following initializes %40 to the largest doubleword integer:

```
        MOV.D.D %40,BIGNUM
        DSPACE
BIGNUM: 377777,,-1
        -1
        ISPACE
        ...
```

but a more elegant way, using a literal, would be:

```
        MOV.D.D %40,[377777,,-1 ? -1]
```

Similarly, the following example uses %40 to index into a table of indirect pointers, perhaps to implement a CASE statement in Pascal:

```
                JMPA CTABL[%40]↑2@
DSPACE
        CTABL:  CASE0+TAG
                CASE1+TAG
                CASE2+TAG
ISPACE
```

but a literal expresses the same structure more compactly:

```
JMPA <[CASE0+TAG ? CASE1+TAG ? CASE2+TAG]>[%40]↑2@
```

3.3.5 Text Constants

An ASCII text constant is enclosed in double-quotes and has the value of the right-adjusted ASCII characters packed one to a quarterword. For example:

```
"ab"
```

is the same as the number 141142_8. If more than four characters are specified, then only the value of the last four will be used. If the trailing double-quote is missing, the assembler will stop accumulating characters when it sees the end of line. The last four characters will be used in the constant and no error message will be given.

A delimiter such as a space must precede a text constant so FASM does not consider it to be a quoted portion within a symbol.

3.3.6 Value-returning Pseudo-ops

Some pseudo-ops generate values and may be used as terms in an expression. See the descriptions of the individual pseudo-ops to learn what values they return.

3.3.7 Combining terms to make expressions

FASM determines the value of an expression simply by combining the values of the individual terms according to the operators between them.

Determining the attribute of the expression is a bit more complicated, however.

When a symbol with the register attribute appears in an expression, then the entire expression has the register attribute. At most one external symbol may appear in an expression. It does not matter how it appears in the expression; it is assumed to be added in. This causes the expression to be an XVAL. If an IVAL (DVAL) ever appears in an expression then the whole expression is an IVAL (DVAL) with one exception. An IVAL (DVAL) minus an IVAL (DVAL) is no longer an IVAL (DVAL). Note: in a relocatable assembly all relocation is done by *addition* of the I space or D space

relocation or of an external symbol's value. Therefore using the negative of an IVAL, DVAL or external value will not have the right effect.

3.4 Statements

A statement can accomplish three things: define a symbol, emit an S-1 instruction, or emit a data word.

How a statement is terminated will depend upon the exact type of statement. In general, a statement is terminated with a linefeed, a ↔, a ?, or a semicolon that begins a comment. (The comment itself terminates at the next linefeed. Some statements, like symbol definitions, can also be terminated with a space or a tab.

3.4.1 Symbol Definition

A symbol may be defined to have a specific value either with the assignment statement or by declaring the symbol to be a label. The assignment statement has two forms:

```
SYMBOL←expression or SYMBOL←←expression
```

An = may be used in place of a ←. These statements define or redefine the symbol to have the value of the expression. The expression must be defined at the time the assignment statement is processed. Any attributes of the expression are passed on to the symbol (except for the *half-killed* attribute). For example, if the expression has a register value, then the symbol is given the register attribute. In addition if the second form is used (with two left-arrows) then the symbol will additionally be given the half-killed attribute. This attribute is not used by the assembler but is passed on to the debugger, where it means that the symbol should not be used in symbolic type-out. It does not affect the ability to use the symbol for type-in.

A symbol may be declared to be a label by saying either of:

```
SYMBOL: or SYMBOL::
```

These both define the symbol to be equal to the location counter. The attributes of the location counter are passed on to the symbol. The double colon (::) causes the symbol to be half-killed.

It is legal to redefine a symbol's value with an assignment statement but it is not possible to redefine a label's value or to define as a label any symbol that has previously had a value assigned.

An assignment statement can itself be an expression and has the value of the expression to the right of the arrows. Therefore it is possible to assign the same value to multiple symbols as follows:

```
A←B←←C←%1
```

which will define all of A, B and C to have the register value 1. An assignment statement is terminated by almost any separator, including space and tab. Therefore it is possible to put more than one assignment statement on one line, or to put an assignment statement on the same line with other statements.

Note that while the text of this manual refers to registers with symbols that suggest their decimal singleword numbers, it is necessary to define them with octal quarterword address:

```
R8←%40
```

3.4.2 S-1 Instructions

An instruction is a statement that can cause the assembly of one, two or three singlewords. It is made up of an opcode with modifiers followed by a list of operands.

3.4.2.1 Operands

(Throughout the following discussion, either "*" or "?" indicates a constant, and "| ..... |", "⊂ ..... ⊃", and "[.....]" are all equivalent pairs of brackets.)

In general, an operand may be any of the following:

Register or memory reference:

`expression` If the attribute of the expression is "register", FASM interprets it as a quarterword address in the registers; otherwise, FASM interprets it as a quarterword memory reference. If an instruction requires a singleword address, FASM derives it by dividing the value of the specified label or expression by four. If an instruction requires a relative address, FASM derives it by subtracting the current location counter from whatever label or expression the programmer provides.

General constant:

`#expression` If the expression is in the range -32 . . 31 (decimal) the assembler will generate a short constant. If not, it will generate a long, sign-extended constant. (It is dangerous to use an as yet undefined symbol in this expression, because the assembler might decide to switch from one length to the other, confusing the rest of the assembly.)

Pseudoregister:

```
(register expression)expression
```

Long constants:

```
#⊂expression⊃
#[expression]
#⊂!S ↔ expression⊃
#[!S ? expression]
```

Any of these produces an LO constant (even if the number is small enough to fit inside an SO) right justified with sign extended or compressed as necessary.

```
#⊂expression ↔ !0⊃
#[expression ? !0]
```

Either of these produces an LO constant which, if the instruction using it calls for a doubleword, is left justified and extended with zeroes. The spaces around the "↔" or "?" are optional.

```
#[!0 ? expression]
#⊂!0 ↔ expression⊃
```

Either of these operands produces an LO constant which, if the instruction using it calls for a doubleword, is right justified and extended with zeroes. The spaces around the "↔" or "?" are optional.

Indexed constant:

```
#⊂expression⊃(register expression)
#⊂expression⊃[register expression]
#[expression](register expression)
#[expression][register expression]
```

An indexed constant adds a constant to the contents of a singleword register. The register expression must lie in the range 0 . . 124 and be divisible by 4.

Operand descriptor:

`!expression` Intended primarily for patching, this generates an operand descriptor (OD) that matches the low 12 bits of the result of the expression. FASM does not check to be sure such an OD is legal, and does not generate an extended word even if the OD calls for one.

Long operand variable:

```
(base)offset[index]↑shift
⊂base⊃offset(index)↑shift
base[index]↑shift
base(index)↑shift
```

This is the general syntax for a long operand (LO) variable. The processor computes the address as if by scanning the expression from left to right. It starts with the contents of the memory location or register specified by "base". Then it adds "offset", if any. Finally it takes the contents of the memory location or register specified by "index", shifts it left by the number of bits specified by "shift", and adds it to the base-offset combination to obtain the address of the operand.

If "@" appears after the entire phrase, indicating indirect addressing, the processor interprets the operand as a pointer and uses it to fetch the ultimate operand. If, on the other hand, the "@" appears after the offset, the processor uses the base+offset address to fetch a pointer from memory, and indexes from it.

The LO variable addressing modes have space use the OD for a sort of "nested" short operand (SO) variable, and they fall into three categories based on how they use this SO variable: as the base, as the index, or not at all.

DEFINITION OF TERMS:

SW_REG	<R0 .. R31>
LONG_DISP	31-bit signed displacement
LONG_ADDR	31-bit unsigned address
SHORT_DISP	26-bit signed displacement
SHIFT	0 .. 3 bit left shift

```
SHORT_SHIFT     Ø or 2 bit left shift
INDEX_REG       <R3 .. R31>
SF              -32 .. 31

USING THE SO AS THE BASE:

        (SW_REG)LONG_DISP
        (SW_REG)LONG_DISP@
        (SW_REG)SHORT_DISP[SW_REG]↑SHIFT
        (SW_REG)SHORT_DISP@[SW_REG]↑SHIFT
        (SW_REG)SHORT_DISP[SW_REG]↑SHORT_SHIFT@

        ((INDEX_REG)SF)LONG_DISP
        ((INDEX_REG)SF)LONG_DISP@
        ((INDEX_REG)SF)SHORT_DISP[SW_REG]↑SHIFT
        ((INDEX_REG)SF)SHORT_DISP@[SW_REG]↑SHIFT
        ((INDEX_REG)SF)SHORT_DISP[SW_REG]↑SHORT_SHIFT@

USING THE SO AS THE INDEX:

        LONG_ADDR[SW_REG]↑SHIFT
        LONG_ADDR@[SW_REG]↑SHIFT
        LONG_ADDR[SW_REG]↑SHORT_SHIFT@
        (SW_REG)SHORT_DISP[SW_REG]↑SHIFT
        (SW_REG)SHORT_DISP@[SW_REG]↑SHIFT
        (SW_REG)SHORT_DISP[SW_REG]↑SHORT_SHIFT@

        LONG_ADDR[(INDEX_REG)SF]↑SHIFT
        LONG_ADDR@[(INDEX_REG)SF]↑SHIFT
        LONG_ADDR[(INDEX_REG)SF]↑SHORT_SHIFT@
        (SW_REG)SHORT_DISP[(INDEX_REG)SF]↑SHIFT
        (SW_REG)SHORT_DISP@[(INDEX_REG)SF]↑SHIFT
        (SW_REG)SHORT_DISP[(INDEX_REG)SF]↑SHORT_SHIFT@

NOT USING THE SO:

        LONG_ADDR
        LONG_ADDR@
        (SW_REG)SHORT_DISP
        (SW_REG)SHORT_DISP@
```

3.4.2.2 Opcodes and Modifiers

An opcode is built out of a base opcode name followed optionally by a "." and an opcode modifier and another "." and another modifier, etc. The modifiers are standard as defined in the opcode files. Numeric modifiers are in decimal *without* a decimal point.

It is also possible to use an already defined symbol as a modifier. For example, if A has been defined by A←%4 then SLR.A assembles the same way as SLR.4 does. Note that an expression may *not* be used in place of a modifier. For example, SLR.4+4 is not permitted in place of SLR.8 . Also note that if there is a conflict between a legal modifier name and a symbolic value, the legal modifier name will win. For example:

```
M1←←1
BNDTRP.M1.S XXX,YYY
```

will NOT be the same as:

```
BNDTRP.1.S XXX,YYY
```

because M1 is a legal modifier for BNDTRP and takes precedence over the lookup of the symbol M1.

Modifiers should not be omitted from instruction opcodes, with one exception: a precision modifier {Q, H, S, D} which is omitted will be assumed to be S. Modifiers should be written in the order defined by the instruction descriptions.

The opcode must be separated from the operand list by spaces or tabs.

3.4.2.3 Instruction Types

There are several basic instruction types: XOPs, TOPs, SOPs JOPs, and HOPs. For the assembler, they differ as to the number and interpretation of operands.

An XOP is (in general) a two-operand instruction. If no operands are given, then the instruction must be one (e.g. WAIT) which requires no operands, and the operand descriptors are set to zero. If exactly one operand is given then, depending upon the specific instruction, either it is used for both operands or the second operand is defaulted to be register zero (R0). For example,

```
INC COUNT
```

is equivalent to

```
INC COUNT,COUNT.
```

A TOP is a three-operand instruction, where one of the operands is restricted. Operands may be written only in certain combinations indicated by a two-bit field called *T* within the instruction. FASM automatically sets this field based on the operands specified by the programmer. If X and Y represent two operands which are distinct from each other and from RTA and RTB, then there are four possible combinations for the operands, as the following shows:

```
SUB X,X,Y
SUB RTA,X,Y
SUB X,RTA,Y
SUB RTB,X,Y
```

Other combinations, such as the following, are illegal:

```
ADD X,Y,RTA
```

If the programmer writes only two operands for a TOP, FASM repeats the first.

An SOP is a two-operand instruction with a skip destination. Both of the operands must be present. The skip destination is written as if it were a third operand, and should be an expression which evaluates to the quarterword address of the instruction that is to be skipped to. If the skip destination is missing, then the instruction is assembled so as to skip over the next instruction, however long it is. For example,

```
ISKP.GTR %1,#100,EXIT
```

assembles a conditional skip to the label EXIT. During the last pass of the assembly, the assembler checks to see that the skip is within range. This means that the value of the skip destination operand must be within -8 . . 7 singlewords of the location of the SOP. The difference in this range is assembled into the SKP field of the instruction.

A JOP is a two-operand instruction, the second of which is the jump destination. If only one operand is specified, then which operand it is assumed to be depends upon the exact opcode. Some opcodes expect only one argument, in which case that argument is the jump destination (JMPA, for example). The opcodes JSR and JCR expect one or two operands. If only one is supplied it is assumed to be the jump destination. For other JOPs, if there is only one argument, it is assumed to be OP1 and the jump is assembled to skip over the next instruction (just as for an SOP with an omitted skip destination). The assembler will try its best to assemble the jump with the PR-bit on (using relative addressing). It even takes a whole extra pass through the source file just for this.

For example,

```
IJMPZ.NEQ %20,LOOP
```

assembles a jump to location LOOP.

The only *HOP* instruction is SJMP, which expects a single operand, which should be a simple label or expression that evaluates to the quarterword address of the jump destination. FASM subtracts the current location counter from the operand value and divides by 4 to obtain the necessary singleword relative address. While compact and useful for patching, this instruction lacks the flexibility of the unconditional branch JMPA, which can use indexing or indirect addressing.

3.4.2.4 Data Words

An expression standing alone on a line (or, more precisely, an expression which by itself constitutes a statement) causes FASM to emit a singleword containing the value of the expression.

```
-1       ; A singleword with all bits set
%7+347.  ; A singleword containing 354 decimal
NAME*2   ; A singleword containing twice the value
         ;  of the symbol NAME
```

If two expressions appear on either side of ",", FASM emits a singleword with the left halfword set to the first expression and the right halfword set to the second.

```
30,,7    ;A singleword with 30 in its left
         ; halfword and 7 in its right halfword
```

The following example illustrates a simple use of a literal. Because the literal itself returns the address of the first word it emits, FASM generates four singlewords in all. At the next "LIT" pseudo-op in data space it generates three singlewords containing 1, 2, and 4 respectively. At the current location counter, it generates a singleword containing the value returned by the literal.

```
[   1
    2
    4   ]
```

3.5 Absolute and Relocatable Assemblies

An assembly is either absolute or relocatable. Initially it is assumed that the assembly is relocatable. Certain things in the input file may cause the assembler to try to change its mind if it is not too late. The pseudo-ops ABSOLUTE and RELOCA will force absolute and relocatable respectively. A LOC will force absolute.

In a relocatable assembly, there is one instruction space and one data space. These spaces may be interleaved in the input file (by use of the ISPACE, DSPACE and XSPACE pseudo-ops) but will be separated into two disjoint spaces in the output. The data space will be output immediately after the instruction space and it is up to the linker to further relocate it to begin on a page boundary (or whatever).

Whenever a word is assembled, the attributes of the expressions involved in the assembly of that word are passed on to the word itself. The assembler outputs instructions to the linker to relocate every IVAL by adding to it the starting address of the instruction segment, and similarly for every DVAL and the starting address of the data segment. Notice that this does *not* do the right thing for the *difference* between an IVAL and a DVAL. This is because the assembler does not keep track of whether the relocation should be positive or negative.

In an absolute assembly, no relocation is done. There may be multiple instruction and data spaces. The pseudo-ops IPAGE and DPAGE cause the assembler to move the location counter to a new page boundary and switch to the indicated space. The assembler output will contain multiple spaces which occur in the same order as the IPAGE and DPAGE statements. The LOC pseudo-op may be used to set the value of the location counter to any desired absolute address (with some restrictions). It cannot be used to change spaces.

An IPAGE, DPAGE, or LOC pseudo-op may not be used in a relocatable assembly, and an ISPACE, DSPACE, or XSPACE pseudo-op may not be used in an absolute assembly.

3.6 Pseudo-ops

The following lists all the pseudo-ops in alphabetical order.

If a "." appears in front of the pseudo-op here, then the "." is mandatory; otherwise it is optional.

Certain pseudo-ops require a string of characters, denoted by ⊗ text ⊗. This indicates that FASM regards the first character (other than a blank or tab) following the pseudo-op as the delimiter for the beginning of the string, and looks for a matching character to delimit the end of the string. Thus, for example, the following produce identical strings:

```
ASCII "Now is the time"
ASCII 'Now is the time'
ASCII bNow is the timeb
```

ABSOLUTE

Forces the assembly to be absolute.

.ALSO,< conditionally assembled text > rest of program
.ELSE,< conditionally assembled text > rest of program

These pseudo-ops conditionally assemble the text in brokets depending upon the success or failure of the immediately preceding conditional. There is an assembler internal symbol called .SUCC which is set when a conditional succeeds and is cleared when one fails. .ALSO will succeed if .SUCC is set and .ELSE will succeed if it is clear. If a conditional succeeds, .SUCC is set both at the beginning and at the end of the conditionally assembled text. This enables the inclusion of conditionals within conditionals while using .ALSO or .ELSE following any outer conditional. For example,

```
IFN A-B,<IFIDN <X>,<Y>,< ...>>
.ELSE < ...>
```

Here, the .ELSE tests the success of the IFN A-B independent of whether the IFIDN succeeded or failed.

ASCII ⊗ text ⊗

Assembles *text* as ASCII characters into consecutive quarterwords, padding the last used singleword with zeros. This pseudo-op may cause more than one word to be assembled as long as it is not enclosed in any level of brokets. However, the "value" of this pseudo-op is the value of the last word it would assemble. So if it is used in an expression, the arithmetic applies only to the last word. If it is enclosed in brokets, then all but the last word are thrown away. For example,

```
1+ASCII /ABCDEFG/
```

is the same as

```
ASCII /ABCD/
<ASCII /EFG/>+1
```

but not the same as

```
1+<ASCII /ABCDEFG/>
```

which is the same as

```
1+ASCII /EFG/
```

ASCIIV ⊗ text ⊗
Is the same as ASCII except that macro expansion and expression evaluation are enabled from the beginning of text as in PRINTV. "\", "‘", and "’" may be used as in PRINTV.

ASCIZ ⊗ text ⊗
Same as ASCII except that it guarantees that at least one null character appears at the end of the string.

ASCIZV ⊗ text ⊗
Is the same as ASCIIV except it does ASCIZ.

.AUXO <filename>
Prepares the file <filename> to receive auxiliary output. Auxiliary output can be generated with the AUXPRX and AUXPRV pseudo-ops. The auxiliary output file remains open until the next .AUXO or the end of the assembly is encountered. It is probably most appropriate to do the .AUXO during just one pass of the assembly. This can be done, for example by

```
IF3,<.AUXO MSG.TXT[P,PN]>
```

AUXPRX ⊗ text ⊗
The *text* is output to the auxiliary file. An error message is generated if no auxiliary file is open.

AUXPRV ⊗ text ⊗

Is the same as AUXPRX except that macro expansion and expression evaluation are enabled from the beginning of *text* as in PRINTV. "\", "`", and "' may be used as in PRINTV.

BLOCK `expression`

Adds expression*4 to the location counter. That is, the expression is the number of singlewords to reserve. The expression must be defined when the BLOCK pseudo-op is encountered.

BYTE `(s1)b11,b12,b13,... (s2)b21,b22,b23,...`

The BYTE pseudo-op is used to enter bytes of data. The s-arguments indicate the byte size to be used until the next s-argument. The b-arguments are the byte values. An argument may be any defined expression. The BYTE pseudo-op may *not* evaluate to more than one word. The s-values are interpreted in decimal radix. Scanning is terminated by either ⊃ or >, so a BYTE pseudo-op may be used in an operand or in an expression. For example,

```
MOV A,#⊂BYTE (7)15,12⊃
MOV B,[1+<BYTE (7)15,12>]
```

COMMENT `⊗ text ⊗`

The *text* is totally ignored by the assembler.

DEFINE `name argument-list`

This pseudo-op is used to define a macro. See the section on macros for a description.

DPAGE

If the current space is instruction space, it does an implicit LIT, advances the location counter to the next page boundary, and sets the space to data. If the current space is data, it merely advances to the next page boundary. This pseudo-op may not appear inside of a literal or in a relocatable assembly.

DSPACE

This is ignored if the current space is already data. Otherwise it switches to data space and restores the location counter from the last value it had in data space. This pseudo-op may not appear inside of a literal or in an absolute assembly.

END expression

Indicates the end of the program. The expression, which may be omitted, is taken to be the starting address. This pseudo-op may not appear inside of a literal. END forces an implicit LIT to be done first for both instruction and data space. The expression must be defined when the END pseudo-op is encountered.

EXTERNAL sym1,sym2,sym3,...

This pseudo-op defines the symbols in the list to be "external" symbols. The symbols in the list must not be defined anywhere in the program. Only one external reference may be made per expression. The value of the external will be ADDED by the linker to the word containing the expression regardless of the operation the expression says to perform on the external symbol.

IF1,<conditionally assembled text> rest of program
IFN1,<conditionally assembled text> rest of program
IF2,<conditionally assembled text> rest of program
IFN2,<conditionally assembled text> rest of program
IF3,<conditionally assembled text> rest of program
IFN3,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the assembler is in pass 1, 2 or 3 for IF1, IF2 and IF3 or if the assembler is not in pass 1, 2 or 3 for IFN1, IFN2, IFN3.

IFDEF symbol,<conditionally assembled text> rest of program
IFNDEF symbol,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the symbol is defined or not for IFDEF and IFNDEF respectively.

IFE expr,<conditionally assembled text> rest of program
IFN expr,<conditionally assembled text> rest of program
IFL expr,<conditionally assembled text> rest of program
IFG expr,<conditionally assembled text> rest of program
IFLE expr,<conditionally assembled text> rest of program
IFGE expr,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the condition is met. If the condition is not met, then the program is assembled as if the text from the beginning of the pseudo op to the matching > were not

present. For IFE the condition is "the expression has value zero," for IFN it is "the expression has non-zero value," etc. In any case the expression must not use any undefined or external symbols. The comma, < and > must be present but are "eaten" by the conditional assembly statement. In deciding which is the matching right broket, all brokets are counted, including those in comments, text and those used for parentheses in arithmetic expressions. Therefore one must be very careful about the use of brokets when also using conditional assembly. For example, the following example avoids a potential broket problem:

```
IFN SCANLSS,<
        SKP.NEQ A,#"<"          ;> MATCHING BROKET
        JMPA FOUNDLESS
>;END OF IFN SCANLSS
```

The broket in the comment is used to match the one in double quotes so that the conditional assembly brokets will match.

IFIDN <string1>,<string2>,<conditionally assembled text> rest of program
IFDIF <string1>,<string2>,<conditionally assembled text> rest of program
These are text comparing conditionals. The strings that are compared are separated by commas and optionally enclosed in brokets. If the strings are identical (different for IFDIF) then the text inside the last set of brokets is assembled as for arithmetic conditionals.

IFB <string>,<conditionally assembled text> rest of program
IFNB <string>,<conditionally assembled text> rest of program
These text testing conditionals compare the one string against the null string. They are equivalent to

```
IFIDN <string>,<>,< ... > ...
IFDIF <string>,<>,< ... > ...
```

.INSERT <filename>
Starts assembling text from the new file <filename>. When the end of file is reached in the new file, input is resumed from the previous file. .INSERTs may be nested up to a level of 10.

INTERNAL sym1,sym2,sym3,...
Defines each symbol in the list as an "internal" symbol. This makes the value of the symbol available to other programs loaded separately from the one in which this statement appears.

IPAGE

If the current space is data space, it does an implicit LIT, advances the location counter to the next page boundary and sets the space to instructions. If the current space is instructions, it merely advances to the next page boundary. This pseudo-op may not appear inside of a literal or in a relocatable assembly.

ISPACE

This is ignored if the current space is already instructions. Otherwise it switches to instruction space and restores the location counter from the last value it had in instruction space. This pseudo-op may not appear inside of a literal or in an absolute assembly.

.LENGTH ⊗ text ⊗

Has the value of the length of the string *text*. A CRLF counts as one character.

LIST

Increments listing counter. Listing is enabled when the count is positive. The count is set to one at the beginning of each pass. XLIST is used to decrement the count.

LIT

Forces all literals in the current space (instruction or data) that have not yet been emitted to be assembled starting at the current location counter. It has no effect on the literals in the "other" space. This pseudo-op may not appear inside of a literal.

LOC expression

Sets the location counter to the specified quarterword address. May not appear inside of a literal or in a relocatable assembly.

MLIST

Increments macro listing counter. Macro expansion listing is enabled when the count is positive. The count is set to one at the beginning of each pass. XMLIST is used to decrement the count.

PRINTV ⊗ text ⊗

Prints *text* on the console. It is identical to PRINTX except that macro expansion may occur within the text. \, ', and ' may be used within the text as in macro arguments and expression evaluation. See the section on special processing in macro arguments for an explanation of \ and '' processing. Macro expansion is intially enabled at the beginning of text and may be disabled with \.

PRINTX ⊗ text ⊗

Prints *text* on the console.

.QUOTE ⊗ text ⊗

Legal only inside a macro definition. It allows the assembler to see *text* without scanning it for a DEFINE or a TERMIN.

RADIX expression

Sets the current radix to expression. The radix may not be set less than two.

RELOCA

Forces the assembly to be relocatable.

REPEAT expression,<body>

Assembles *body* concatenated with a carriage return *expression* many times. The expression must be defined at the time the REPEAT pseudo op is encountered. The expression must be non-negative. If it is zero, the body will not be assembled.

TERMIN

This pseudo-op is legal only during a macro definition. It is used to terminate a macro definition. See the section on macros for a description.

TITLE `name other_text`
Sets the title of the program to *name*. Everything else on the line is ignored.

XLIST
Decrements listing counter. Listing is enabled when the count is positive. The count is set to one at the beginning of each pass. LIST is used to increment the count.

XMLIST
Decrements macro listing counter. Macro expansion listing is enabled when the count is positive. The count is set to one at the beginning of each pass. MLIST is used to increment the count.

XSPACE
Has the effect of ISPACE if the current space is data and DSPACE if the current space is instructions. This pseudo-op may not appear inside or a literal or in an absolute assembly.

3.7 Macros

The FASM macro facility shows a strong resemblance to those of FAIL (the macro assembler for the PDP-10 developed and used at the Stanford Artificial Intelligence Laboratory) and MIDAS (the macro assembler for the PDP-10 developed and used at the M.I.T. Artificial Intelligence Laboratory), which are hereby acknowledged.

Macros are essentially procedures that can be invoked by name at almost any point in the assembly. They can be used for abbreviating repetitive tasks or for moving quantities of information from one part of the assembly to another (in fact even from one pass to another). Macro operation is divided into two parts: definition and expansion.

The macro facility does differ in an important way from those of other assemblers, however. Macro expansion in FASM is performed at the "read-next-character" level, whereas in most other assemblers it is done at symbol lookup time during expression evaluation. Due to this difference, macro expansion in FASM inherently produces "string" output rather than evaluated expressions as is sometimes the case in other assemblers. Wherever a macro call is seen, the effect can be predicted by substituting the body of the called macro in place of the call.

3.7.1 Macro Definition

Macros are defined using the DEFINE pseudo-op, which has the following format:

```
DEFINE macroname argumentlist
  body of macro definition
TERMIN
```

This will define the symbol *macroname* to be a macro whose body consists of all the characters starting after the CRLF that ends *argumentlist* and ending with the character immediately preceding the TERMIN.

3.7.1.1 The Parameter List

Basically, the parameter list is a list of formal parameters for the macro. This is similar to the list of formal parameters for a procedure in a "high" level language. The parameters are symbol names and are separated by commas. The number of macro parameters must be in the range 0 . . 64. The macro parameter list is terminated by either a ; (which begins a comment, as usual) or a CRLF.

Each macro parameter has certain attributes associated with it. In FASM these attributes are *balancedness, gensymmedness,* and *parenthesizedness.* From now on, it shall be said that a parameter is or is not *balanced,* is or is not *gensymmed,* and that certain pairs of parentheses can or cannot *parenthesize* a parameter. If a parameter is not balanced or gensymmed then it is said to be *normal.*

Parameter attributes are specified by enclosing a string of characters in double quotes preceding a parameter in the parameter list. The attributes specified by that string are "sticky"; that is, they apply to all following parameters until the next such string is specified. The characters B and G may appear in the string to indicate that the parameter is to be balanced or gensymmed respectively. There are four parenthesis pairs: (and), [and], < and >, and { and }. Any of these characters may appear in the string to indicate that that set of parentheses may be used to parenthesize that parameter. One final thing that may appear in the string is a statement about the *concatenation* character for the macro body. If the string !=⊗ appears, where ⊗ is any character other than CRLF, then ⊗ will be the concatenation character. If the string 0! appears, then there will be no concatenation character. Only the last statement made in the parameter list about the concatenation character will apply to the macro body.

At the beginning of the parameter list, the attributes have the following defaults: ! is the concatenation character, parameters are neither balanced nor gensymmed, and any pair of parentheses may be used to parenthesize a parameter. Whenever an attribute string is encountered, the previous set of attributes are forgotten and the new one applies to future parameters until the next string is specified.

Here are some examples of valid macro definition lines:

```
DEFINE MAC
DEFINE MAC1 A,B,C
DEFINE MAC2 "!='" A,B, "G" C
DEFINE MAC3 "([B])" A, "[Ø!" B
```

With these definitions, MAC has no parameters and has ! for the concatenation character. MAC1 has three normal parameters A, B and C with ! for the concatenation character. MAC2 has two normal parameters A and B and a gensymmed parameter C, and uses ' as the concatenation character. MAC3 has a balanced parameter A, for which () and [] can be used as parentheses, and a normal parameter B, for which [] can be used as parentheses. MAC3 has no concatenation character.

3.7.1.2 The Macro Body

The macro body begins at the character following the CRLF at the end of the DEFINE line and ends with the last character before the matching TERMIN. Within the macro body, FASM replaces

all delimited occurrences of formal parameters with a mark that indicates where the actual argument should be substituted. Any character that is not a symbol constituent is considered a delimiter for this purpose. The concatenation character is also considered a delimiter. However, the concatenation character is deleted wherever it occurs and will not appear in the macro body definition. The concatenation character is useful to delimit a formal parameter where, without the concatenation character, the formal parameter would not have been recognized as such. For example,

```
DEFINE MAC A,B,C
PUSH.UP.S SP,B
PUSH.UP.S SP,C
JSR A!RTN
TERMIN
```

If the arguments X, Y, and Z were substituted for the formal parameters A, B, and C, then the third line would assemble as JSR XRTN. Without the concatenation character, it would always assemble as JSR ARTN regardless of the actual value of the parameter A.

In addition to scanning for formal parameters in the macro body, FASM also scans for occurrences of the names DEFINE and TERMIN. It keeps a count of how many it has seen so that it can find the TERMIN that matches the DEFINE that began the macro definition. This allows a macro body to contain a macro definition entirely within it. For example,

```
DEFINE MAC1 A
DEFINE MAC!A
....
TERMIN
TERMIN
```

defines a macro called MAC1 which contains a complete macro definition sequence within itself.

Note that FASM does *not* recognize either comments or text constants as special cases in its search for DEFINEs, TERMINs and formal parameters. Therefore, the user must be careful when using the words DEFINE and TERMIN in those places. They *will* be counted in order to find the TERMIN that marks the end of the current definition. There is a pseudo-op called .QUOTE that can be used if it is desired to inhibit FASM from seeing a DEFINE, TERMIN, or macro parameter name. .QUOTE is like an ASCIZ statement in syntax, taking the first nonblank character after the .QUOTE as a delimiter and passing all characters up to the matching delimiter through to the macro definition. For example,

```
DEFINE MAC
....            ;how to put a .QUOTE /DEFINE/ in a comment
TERMIN
```

will define MAC's body to be

```
....                    ;how to put a DEFINE in a comment
```

3.7.2 Macro Calls

A macro call occurs whenever a macro name is recognized in a context where macro calls are permitted. When this happens, the macro call is processed in two distinct phases. The first is argument scanning and the second is macro body expansion.

3.7.2.1 Argument Scanning

Argument scanning is the process of assigning text strings to the formal parameters of a macro. These text strings come from the input stream. If a formal parameter is not assigned a string by the call, then it is assigned the null string as its value, unless the argument is defined to be gensymmed. In that case, the argument is assigned a six character string beginning with G and followed by 5 decimal digits which represent the value of an internal counter which is incremented before being converted to a text string.

Argument scanning is performed for those macros that have formal parameters. If a macro does not have any formal parameters, then the character that terminates the macro name is left to be reprocessed after the macro expansion is complete, even if it is a comma.

If the macro has formal parameters, then how the argument scan is done depends on the character immediately following the macro name. If it is a CRLF, then the argument scan is terminated and all of the formal parameters are assigned the null string or are gensymmed as appropriate. The CRLF is left to be reprocessed after the macro expansion is complete.

If the character following the macro name is a space or a tab, then all immediately following spaces and tabs are thrown out. The entire sequence of spaces and tabs can be considered to be the macro name delimiter.

If the character following the macro name is a (, then the macro call is said to be a parenthesized call; otherwise it is a normal call. A parenthesized call differs from a normal call in the way argument scanning is terminated. In a normal call, argument scanning is terminated by either CRLF (or its surrogates, ? and ↔), semicolon, or the argument terminator for the last argument (which may be a comma). If terminated by a CRLF or semicolon, the terminator is left to be

reprocessed after macro expansion is complete. In a parenthesized call, only the matching) can terminate the call. The) is not reprocessed after the macro expansion is complete. The following paragraphs will describe the syntax of macro arguments and explain how they are terminated. The phrase "... macro call terminator" refers to the character that terminated either the normal or parenthesized call, as described in this paragraph.

3.7.2.2 Macro Argument Syntax

The first macro argument begins with the first character following either the (that demarks a parenthesized call or the macro name delimiter in a normal call. This character is looked at by FASM to determine how to scan the argument.

If the first character is a left parenthesizing character that belongs to the set of characters that may be used to parenthesize the argument that is being scanned (as determined by the character string in force at the time this formal parameter was seen in the macro define line), then the argument is taken to be all characters following that open parenthesis until, but not including, the matching closed parenthesis. *Any* characters may appear between the parentheses. Only the particular type of parentheses that enclose the argument are counted in finding the matching closed parenthesis. This type of argument is called a parenthesized argument.

If the first character is a comma, then the argument is the null string; the comma is taken to be an argument separator.

If the first character is a macro call terminator, then this argument and all further arguments are not assigned strings. That is, if the arguments are gensymmed, they will be assigned unique gensymmed strings, and if they are not gensymmed they will be assigned the null string.

If the first character is not one of the above, then argument scanning depends on whether the argument is to be balanced or not. If the argument is not to be balanced, then the argument is taken to be all characters from the first character until, but not including, a comma, CRLF (or ↔ or ?), semicolon, or the macro call terminator. If the argument terminator is a comma, it is thrown out; a macro call terminator, however, will be kept to terminate the macro call. If the argument terminator is not a comma, then it is usually a macro call terminator. However, if the call is parenthesized, a CRLF or semicolon will terminate the argument but not the macro call. In this case the remainder of the line (if the terminator was a semicolon) is ignored and the CRLF is thrown out. Argument scanning continues on the next line. This allows the arguments of a parenthesized call to take multiple lines; each CRLF acts as if it were a comma (with comments thrown out) allowing the next line to continue supplying arguments.

If the argument is to be balanced, then all types of parentheses are treated the same. A count is kept of the parenthesis level. If there are no unbalanced parentheses, then a comma or macro call terminator will terminate the argument as if it were a normal argument. Also, if the parentheses are

balanced, any close parenthesis will terminate the argument and the call. If it is a parenthesized call, the close parenthesis must be a) or an error is reported. If it is not a parenthesized call, the parenthesis will be left to be reprocessed after the macro call is complete. In either case, the remaining formal parameters are assigned the null string or gensymmed as appropriate.

3.7.2.3 Special Processing in Macro Arguments

Ordinarily, macro arguments are the quoted forms of the strings that appear between delimiters within the macro call. However, it is possible to call a macro or even evaluate an expression *within* a macro argument *during* the macro argument scan.

If a macro argument is not parenthesized, then the appearance of the character \ (backslash) in the argument will enable macro calls to be recognized during the scanning of the macro argument. The appearance of a second \ will again disable this feature. If a macro call is detected during this time, then that new macro is expanded and its expansion appears as if it were written in line in the macro argument that is currently being read. Every time a new macro call is seen and macro argument scanning is started, the macro-in-argument recognition feature is disabled until re-enabled by a \. The \ character itself is discarded.

Perhaps this will be clearer if explained in terms of the actual implementation. FASM maintains a flag, called the \ flag, which when set enables macro expansion. This flag is pushed when a macro name is recognized and initialized to be off at the beginning of the argument scan. It is complemented every time a \ is seen in the input. When the entire macro call has been scanned (but expansion has not yet started) the \ flag is popped.

In fact, the \ flag has wider application than just in macro calls. It is also applicable at expression evaluation time. Normally it is set during expression evaluation, thereby allowing macros to be expanded. It is perfectly legal to use \ during expression evaluation to inhibit macro expansion.

There is a second feature, analogous to the \ feature, which allows the expression evaluator to be called during a macro argument, or in fact even at expression evaluation time. If an expression is enclosed within "‘" and "’" characters, the expression evaluator is called upon to produce a value, which may possibly be null, which is then converted into a character string of digits representing that value in the current radix. The conversion always treats the value as a 36-bit unsigned integer. A null value is converted to the null string. The surrounding singlequotes act in a similar way to parentheses in arithmetic expressions, in that multiple lines may be used, but only the expression on the last line is converted. This converted string is used in place of the singlequoted expression. As in the case of \ this can occur in non-parenthesized macro arguments or in expression evaluation. The singlequote characters themselves are thrown out.

Following are some examples of the use of these features:

```
X←←1  F00'X':   JMPA F001
```

will assemble as

```
F001:   JMPA F001
```

If F00 was a macro name, it would have been expanded in the previous example. This could be inhibited with:

```
\F00\'X':       JMPA F001
```

Next consider:

```
X←←1
DEFINE MAC
X←←X+1
X!TERMIN

F00'MAC':
```

will define the label F002 while incrementing X to be 2. The next time F00'MAC': appears, the label F003: will be generated.

It is sometimes useful to extract the value of a symbol in a macro argument before the macro call changes that value:

```
DEFINE MAC A
BRR←←BRR+1
A*BRR
TERMIN

MAC 'BRR'
```

will call MAC with the current value of BRR. Without the singlequotes, the string BRR would be passed to the macro and used where "A" appears, which is after BRR is incremented.

4 The Mark IIA implementation

Implementation-dependencies belong in two categories: those which are not specified by the architecture but instead left to the implementor, and those which vary from the definition of the architecture.

4.1 Details about Performance Counters

Certain details of performance counters are not specified by the architecture, but are instead left to the implementor. The following discussion applies to the Mark IIA implementation.

The interrupt priority of counter traps is 10000b (sixteen decimal). This is hardwired into the machine.

There are 34 72 bit counters. All of them trap on overflow (from negative to positive). Counters 32 and 33 can count more than one tic per machine cycle.

Counter	Purpose
0 . . 3	Real-time (one tick per cycle)
4	Instructions
5 . . 7	Reserved
8	Instruction map cache accesses
9	Instruction map cache misses
10	Instruction cache accesses
11	Instruction cache misses
12	Data map cache accesses
13	Data map cache misses
14	Data cache accesses

15	Data cache misses
16	Conditional branches
17	Wrong branches
18 . . 31	Reserved
32	FLOPS
33	Reserved

Simultaneous (or nearly simultaneous) counter interrupts arrive as a single interrupt. If more than one counter interrupts before the first interrupt is handled, the second gets merged with the first. Only one of the interrupts is actually reported. The save area for traps and interrupts contains a singleword which specifies which counter caused the interrupt, but the prudent handler may choose to inspect all the other counters as well, in case merging has occurred.

The granularity of counter interrupts is about 64 cycles. For counters 32 and 33 it is two cycles.

4.2 Variances from the architecture

The Mark IIA implementation varies in some respects from the description of the architecture:

1. Segment bounds checking does not occur when an instruction is fetched; in other words, the instruction may cross a segment boundary. However, the hardware will always check that EXECUTE_PERMIT is on for the page containing the first word of the instruction and the page containing the singleword which lies two singlewords after it, regardless of the number of singlewords occupied by the instruction and its operands.

2. The USED bit in a PTE may, as a result of wrong-branch evaluation in the pipeline, indicate that a page was used when in fact it was not. A similar statement applies to the MODIFIED bit.

3. Attempting to take the FFT of a vector of more than 2^{14} elements causes an FFT_TOO_LONG soft trap.

4. Only the 11 low-order bits of address space IDs are significant. (The architectural prohibition against setting an address space ID to 0 still holds.)

5. Instructions for which rounding is inexact guarantee their results are monotonic--that is, if $x \geq y$ then $F(x) \geq F(y)$--with an error that is less than or equal to 0.75 of the least significant bit of the mantissa. Instructions for which rounding is exact guarantee an error less than or equal to 0.5 of the least significant bit.

The following instructions exhibit inexact rounding:

FRECIP
FCMAG, VFCMAG
FSQRT, VFSQRT
FLOG
FEXP
FSIN
FCOS
FSINCOS
FATAN, FATANV
VF2DIS, VF3DIS
FCFFT, FCFFTV

6. RETS.A will not copy CALL_TRACE_PEND from the value of CALL_TRACE_ENB in the saved PROCESSOR_STATUS. If one aborts a call or return instruction, one must intervene anyway to patch up the control flow of the program, and one can explicitly reinvoke tracing. RETS.A will handle TRACE_PEND as specified in section 1.9.2.

7. INTIOP will pass only the low-order byte of OP2 to the I/O processor.

8. All segmentito table entries (STEs) must be doubleword aligned. The DSEGP must be

doubleword aligned; in other words, the three low-order bits must be zero.

9. Operand S2 of DSHF.RT must evaluate to an integer in the range 0 . . Precision - 1, where "Precision" is 9 for the quarterword modifier, 18 for the halfword modifier, and 36 for the singleword modifier.

5 Index

Technical Information Department · Lawrence Livermore Laboratory
University of California · Livermore, California 94550