COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD. CA 94305

STAN-B-79-7 15

S-l ARCHITECTUREMANUAL

Brent T. Hailpern and Bruce L. Hitson

TECHNICALREPORTNO.161

January 1979

This report was prepared in order to document the S-l multiprocessor architecture. the central project of the Advanced Digital Processor Technology Base Development for Navy Applications, under subcontract from Lawrence Livermore Laboratory to Stanford University, Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract No. LLL P09083403. Other Lawrence Livermore Laboratory as well as Advanced Research Projects Agency contracts have supported the facilities at the Stanford Artificial Intelligence Laboratory, which was used in the execution of this work. The S-l project is supported at Lawrence Livermore Laboratory of the University of California by the Department of the Navy via ONR Order No. N00014-78-F0023.

STAN-CS-79-715

S-1 ARCHITECTURE MANUAL

Brent T. Hailpern and Bruce L. Hitson

TECHNICAL REPORT NO. 161

January 1979

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This report was prepared in order to document the S-1 multiprocessor architecture, the central project of the Advanced Digital Processor Technology Base Development for Navy Applications, under subcontract from Lawrence Livermore Laboratory to Stanford University, Computer Science Department, Principal Investigator Professor Gio Wiederhold, Contract No. LLL P09083403. Other Lawrence Livermore Laboratory as well as Advanced Research Projects Agency contracts have supported the facilities at the Stanford Artificial intelligence Laboratory, which was used in the execution of this work. The S-1 project is supported at Lawrence Livermore Laboratory of the University of California by the Department of the Navy via ONR Order No. N00014-78-F0023.

S-1 Architecture Manual

(SMA-3)

Brent T. Hailpern, Bruce L. Hitson

TECHNICAL REPORT NO. 161

January 1979

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

This manual provides a complete description of the instruction-set architecture of the S-1 Uniprocessor (Mark IIA), exclusive of vector operations. It is assumed that the reader has a general knowledge of computer architecture. The manual was designed to be both a detailed introduction to the S-1 and an architecture reference manual. Also included are user manuals for the FASM Assembler and the S-1 Formal Description Syntax.

KEY WORDS: S-1, architecture description, instruction set description, addressing modes, trapping mechanisms, high-speed architecture.

1 Introduction

This manual provides a complete description of the instruction-set architecture of the S-l Uniprocessor (Mark IIA), exclusive of vector operations. It is assumed that the reader has a general knowledge of computer architecture. The manual was designed to be both a detailed introduction to the S-l and an architecture reference manual.

This manual does not describe the S-l performance architecture, or any other implementation-related aspects of the S-l Uniprocessor, except as is necessary to make the S-l instruction-set architecture understandable.

The remainder of this chapter discusses the notation used throughout the manual. Chapter 2 describes the structure of the S-l's memory and registers, including the status words and the concept of address contexts. Chapter 3 defines various conceptual data types used in the discussion of the S- 1 instructions. Chapter 4 describes the formats of the S-l instructions and how operands are addressed. Chapter 5 describes the individual instructions in detail. Chapter 6 describes the architecture of traps and interrupts in the S-l. The remaining chapters provide examples and summaries. The two appendices summarize the FASM Assembler (because examples throughout the manual uses the FASM syntax) and the S-l Formal Notation (which is used to precisely define the instruction set).

1.1 Notation and Conventions

This section describes the notation used in the text of this manual. Many of the abbreviations used in this section may not be understood until later sections of the manual are read, but they are presented here for the sake of completeness. Most of the examples in the manual are stated in the syntax of the FASM assembler. That syntax is summarized in Section 10 with various aspects of it introduced at appropriate points in the main text as well. The syntax used to formally describe the S-l and its instructions is summarized in Section 11.

The notation "A . . B" (borrowed from PASCAL-like programming languages) means the range of integers from A to B inclusive, or the set of the elements of that range, depending on context.

The term ***field*** means a series of consecutive bits within memory or a register. The bits in a field are always numbered from left to right, starting at zero. Subfields of a field are specified by the notation X<*m*:*n*>. Here X is the name of the field, and the subfield being referenced is the bits of X whose numbers within X are in the range *m*,*n* . . . A reference to a single bit (X<*m*:*m*>) can be abbreviated to X<*m*>. The selection of a named subfield is indicated as X. SUB (X is the name of the field, SUB is the name of the subfield within X). Subfields, like like all fields, always have their bits numbered from left to right starting from *zero,* and so the bits of a subfield may not have the same bit numbers as those same bits within the superfield.

The term word is intended to mean a field of any of the four standard precisions (quarter-word, single-word, half-word, and double-word, which are 9, 18, 36, and 72 bits wide respectively). It is intended that if word is not modified then no specific precision is being described, or rather what is being said applies to words of all four precisions. Not every field 9 bits long is a quarter-word; the term word also implies alignment of the field to a *word boundary* (see Section 2.1). Words, like all fields, may have subfields.

For example, Figure 2-4 is reproduced below as Figure l-l. This picture of a single-word ‘shows the format of a page-table entry.

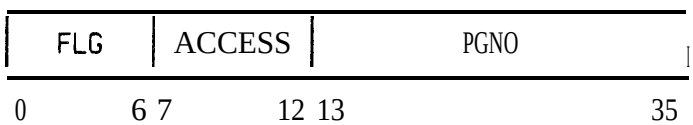

Figure l-l
PTE or STE

This single-word could have the name PTE (for reasons described in Section 2.3). In that case, PTE. FLG would be the same as PTE<0:6>, and PTE.ACCESS the same as PTE<7:12>. The second through fourth bits of PTE.ACCESS could be described as either PTE<8:10> or PTE. ACCESS< 1:3>.

A *byte* is a subfield of a single-word *or* double-word which is specified by a *byte* ***pointer***. A

byte may be of any length (not just eight bits, for example). The term *byte* bears no relation in this manual to the amount of memory used to contain a character code. (See Sections 3.5 and 3.6.)

The notation used to describe the concatenation of fields into a larger unit is cfield 1 || field2 || field33 (i.e., field 1, field2, and field3 are concatenated to form one unit). For example, figure 1-l could be described as ⊂FLG<0:6> || ACCESS<0:5> || PGNO<0:22>⊃. Unless otherwise stated, this new conglomerate is treated as a single unit (e.g., the concatenation of two quarter-words is a half-word, not merely two quarter-words). This distinction becomes important when considering alignment issues. If a field is repeated in the conglomerate then that may be specified using the notation n*field, where n is the number of times the field is repeated. For example, cfield 1 ||5*0|| field23 would be the same as cfield 1 || 0 || 0 || 0 || 0 || 0 || field2⊃.

The contents of register number n is R[n]. The contents of memory location A is M[A]. The terms OP 1, OP2, S 1, S2, and DEST refer to the *contents* of the appropriate locations. Some instructions operate on a pair of memory locations. If X is the first object of such a pair, then NEXT(X) is the second object of the pair. X and NEXT(X) are contiguous and have the same precision. The address of NEXT(X) is greater than the address of X by the length of X (which is the same as the length of NEXT(X)). As with OP1, NEXT(OP1) refers to the contents of the appropriate location (the same applies to the other terms given above). ADDRESS(OP 1) refers the the quarter-word (virtual) address of OP1. The term JUMPDEST represents an address. The terms SO (short operand), LO (long operand), and IL0 (indirect long operand) also refer to the contents of the appropriate locations (or to the values of immediate constants, if appropriate).

If a field X is to be interpreted as a two's-complement number, then the notation SIGNED(X) is used. When only part of a word (or the result of a computation), X, is to be used, the terms LOW-ORDER(X) and HIGH-ORDER(X) designate the least-significant and most-significant portion of X, respectively. When used informally, it should be obvious from the context how much of X is included; otherwise the precision will be stated explicitly. Unless otherwise stated, when moving a smaller field, X, into a larger field, Y, it is the case that X is right-justified into Y. The bits in Y that were not in X are specified by the moving operation. If ZERO-EXTEND(X) is used, then these extra bits are zero-bits. If SIGN-EXTEND(X) is used, then these extra bits are ueqla to the sign-bit of X. (The sign-bit of X is X<0>).

⌜Text appearing within four corner-brackets is intended as an illustrative example rather than as part of the main discussion. Typically an example will give sample data formats or sample instruction sequences. This text, on the other hand, is an example of an example.⌟

2 Memory and Registers

The S-l architecture provides for a very large (2^{28} single-word) virtual address space. Virtual-to-physical address transformation is handled by the hardware. Single-words are 36-bits long but the architecture allows for the accessing of memory in any of four different precisions (quarter-word, half-word, single-word, and double-word). Thirty-two general purpose register words are provided which can be accessed via special register operations or as memory locations. Separate address spaces and register-files are maintained for the user and the executive. The following sections in Chapter 2 describe these features in detail.

Each S-l processor has two private caches to reduce memory access times for those sections of memory that are frequently accessed. One cache is for instructions and the other is for data. The caches are described in Section 5.15.

2.1 Memory

The S-l architecture provides 2^{28} single-words of virtual address space. Each single-word is thirty-six bits long. The bits are numbered 0 . . 35 from most significant to least significant.

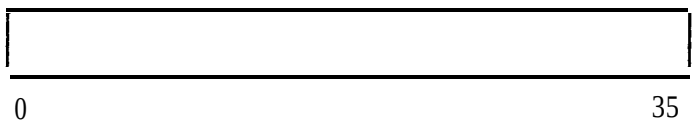

Figure 2-1
Single-Word

Memory may be accessed in any of four precisions: *quarter-word* (nine bits numbered 0 . . 8), *half-word* (eighteen bits numbered 0 . . 17), *single-word* (thirty-six bits numbered 0 . . 35), or double-word (seventy-two bits numbered 0 . . 71). Therefore, the single-word above could be considered to be two half-words, four quarter-words, or half of a double-word. Instructions are designed to access and operate on words of all four precisions with equal ease.

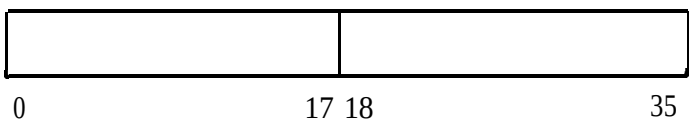

Figure 2-2
Two Half-Words

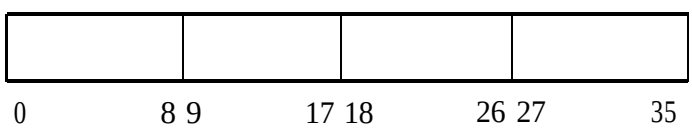

Figure 2-3
Four Quarter-Words

Quarter-words within a half-word, single-word, or double-word have increasing addresses from left to right. Thus if a quarter-word and a single-word have the same address, then the quarter-word is the high-order (most significant, or leftmost) quarter-word of the single-word. Similarly, the more significant single-word in a double-word has the lower address.

Unless otherwise stated, all addresses mentioned are quarter-word addresses. Therefore, the range of S- 1 addresses is $0 \ldots 2^{30}-1$. Half-words must be aligned on half-word boundaries, that is, the most-significant quarter-word of a half-word must have an even address. Similarly, single-words must be aligned on single-word boundaries (the most-significant quarter-word must have an address that is a multiple of four). Double-words must begin on single-word boundaries, but they need *not* begin on double-word boundaries. Depending upon the implementation, however, access to double-words beginning on double-word boundaries may be more efficient than those not so aligned.

References to the first 128 quarter-words of memory are interpreted as references to the thirty-two (single-word) registers. Registers are discussed in Section 2.2.

2.2 Registers

Registers can be used to hold information that must be accessed quickly or concisely. They are addressable by the use of register addressing modes, or as the first 128 quarter-words of memory. Some registers are dedicated to special-purpose applications, while others are available for general-purpose use. The instruction set has been designed to deal efficiently with registers and with memory locations addressed by a small offset from a register. In addition, special instructions are provided for saving and restoring registers during interrupts, traps, and subroutine calls. The registers and their uses are described in the following sections.

2.2.1 Register Files

There are sixteen *register files* (REG_FILEs) in the S-l architecture. Each consists of thirty-two single-word registers. REG_FILE[0] is reserved for use by the hardware and microcode. The other fifteen register files may be put to any use by software.

The processor status word selects which register files are being used by the current context and the previous context (one register file for each context). The user may access only the thirty-two registers in the register file associated with the current context. The executive, however, may access either context, and so which register file is used depends on which context is being accessed. The processor status word is discussed in Section 2.5.1. Contexts are discussed in Section 2.4.

The organization of registers into register files facilitates context switching. Each of several users may have his own register file that the executive can specify simply by changing a field in the processor status word. Similarly, each of several trap or interrupt handlers within the executive can have a dedicated register file and need not save the registers of the previous context.

22.2 General-Purpose Registers

The contents of the first single-word of the current register file is called REOJ, the second R[1], and so forth. When not otherwise modified, the term *register* will hereafter be used to mean one of the thirty-two registers in the current register file. Other registers (e.g., PC or STP) will be referred to specifically by name,

Many instruction formats can make special use of registers. Some registers have restrictions on, or extensions of, these special uses. Registers addressed as memory have no special properties.

Registers 8 through 31 can be used as general-purpose registers in all instructions that make special use of registers. Registers 0 through 7 have certain special-purpose uses but they can also be used as general-purpose registers, with some restrictions. Registers 0 through 3, for example, cannot be used in short-indexed mode (see Section 4.2.3.3). Other restrictions concerning references to register 3 are discussed in Section 2.2.3.1 and Section 2.2.3.2. Register uses and restrictions are summarized in Section 2.2.4.

2.2.3 Dedicated-Function Registers

Certain general-purpose registers in the S-l have special functions associated with them. One register serves as a stack pointer, while others may serve as operands in three operand instructions. These registers and their uses are described below. They are summarized in Section 2.2.4.

2.2.3.1 Program-Counter

The *program-counter* (PC) is a 30-bit register that points to (contains the address of) the instruction in memory that is currently being executed. Because instructions consist of single-words and so are aligned on single-word boundaries, the contents of the PC must always be a multiple of four. The PC always points to the beginning of the instruction being executed (that is, it is not advanced when the extended words of a multi-word instruction are fetched).

References to register 3 are interpreted as references to the PC in certain circumstances. PC is used instead of R[3] whenever register 3 is specified as an index register within an address calculation. This includes indexing in indirect address pointers (see Section 4.2.5). In all other cases, R[3] is treated as a general-purpose register. All non-indexing references to register 3 use R[3]. It should be emphasized that PC itself is *not* a general-purpose register, and does not reside in any register file.

2.2.3.2 Stack-Pointer (SP) and Stack-Limit (SL)

The S-l maintains a stack for saving values during traps, interrupts, and subroutine calls. The location and extent of the stack in memory is specified by the contents of two registers: the *stack-pointer* (SP) and the *stack-limit* (SL). SP points to the first free location on that (upward-growing) stack and SL points to the first location past the end of the area reserved for stack growth.

The five-bit SP_ID field in the user status word (see Section 2.5.2) specifies which general-purpose register will be used as SP. The register immediately following SP is interpreted as the SL register. Hence SP = R[SP_ID] and SL = R[SP_ID + 1]. The values 3 and 31 for SP-ID are illegal; an attempt to set SP-ID to either value will cause a hard trap.

The SP-ID can be set by special instructions (see Section 5.14). The usual practice is to use the two highest-address registers (registers 30 and 31) as the SP and SL respectively.

2.2.3.3 RTA and RTB

Registers 4 and 6 are given the special names RTA and RTB respectively. They are of special interest in three-address instructions, When double-word quantities are involved, then RTA is considered to be registers 4 and 5 together, and RTB is considered to be registers 6 and ‘7 together. Registers 5 and 7 also have the names RTA 1 and RTB 1 respectively. See Section 4.1.2 for a description of the uses of RTA and RTB.

2.2.4 Summary

The tables below summarize the uses of the registers that have been discussed in the previous sections.

Register	Primary Use	Other Uses/Restrictions	Pertinent Sections
R[0]	General-Purpose	Restricted indexing	2.2.2, 4.2.3.3
R[1 . . 2]	General-Purpose	No short indexing	2.2.2, 4.2.3.3
R[3]	General-Purpose	Indexing uses PC instead	2.2.2, 2.2.3.1
R[4]	General-Purpose	RTA	2.2.2, 2.2.3.3
R[5]	General-Purpose	Low-order half of RTA DW	2.2.2, 2.2.3.3
R[6]	General-Purpose	RTB	2.2.2, 2.2.3.3
R[7]	General-Purpose	Low-order half of RTB DW	2.2.2, 2.2.3.3
R[8 . . 31]	General-Purpose	---	2.2.2

Table 2-1
Registers and their Uses

Register	Primary Use	Other Uses/Restrictions	Pertinent Sections
PC	Program-Counter	Indexing uses PC for R[3]	2.2.3.1, 2.2.2
SP	Stack-Pointer	Cannot be R[3] or R[31]	2.2.3.2, 2.2.2
SL	Stack-Limit	Always register after SP	2.2.3.2, 2.2.2
RTA	Third Operand	Same as R[4] (or ⊂R[4]‖R[5]⊃)	2.2.3.3, 2.2.2
RTB	Third Operand	Same as R[6] (or ⊂R[6]‖R[7]⊃)	2.2.3.3, 2.2.2

Table 2-2
Dedicated-Function Registers and their Uses

2.3 Address Transformation

The S-l maps 30-bit, virtual, quarter-word addresses into 34-bit, physical, quarter-word addresses. The address transformation uses two levels of paging, specified by a segment table and up to 1024 page tables. A *page* is made up of 512 single-words (2" quarter-words). There are up to 2^{23} physical pages in memory; hence the physical address space contains 2^{34} quarter-words. A virtual address space contains up to 1024 *segments* (specified by the segment table). Each segment contains 512 pages (specified by one of the page tables). This gives a virtual address space of up to 2^{30} quarter-words.

The location of the current *segment table* is specified by two 34-bit registers: the *segment table* pointer (STP) and the *segment table limit* (STL). If the content of the STP is in the range 0 . . 127 (a register address), then *absolute addressing* is in effect; the mapping from virtual addresses to physical addresses is the identity mapping. Otherwise, the STP contains the physical address of the segment table, and the STL contains the physical address of the first location beyond the end of the segment table. STP<32:33> and STL<32:33> must equal zero, because table entries are single-words and therefore must-be aligned on single-word boundaries.

Each segment table consists of a contiguous list of *segment table entries* (STE) (also called page *table pointers).* Each *page table* consists of a contiguous list of 512 *page table entries* (PTE). Both segment table entries and page table entries have the following format: ⊂FLG<0:6> || ACCESS<0:5> || PGNO<0:22>⊃. Either may be *null* (FLG<0>=0), indicating that the entry specifies no page. FLG contains flag bits. ACCESS indicates the access bits and is used only in page table entries. PGNO is the physical *page number* (page number x 2^{11} = page address). (See Sections 2.3.1 and 23.2 for further discussion of the FLG and ACCESS fields.)

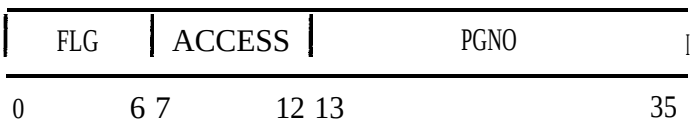

Figure 2-4
PTE or STE

Each STE specifies the physical address of a page table, or is null. A null STE indicates that the page table does not exist. STE. PGNO is used as the most-significant 23 bits of the physical address of the page table (the least-significant 11 bits are zero). page tables fill exactly one page (of 5 12 single-words). Each PTE specifies the physical address of a page, or is null. A null PTE indicates that the page does not exist. As with the STE, PTE.PGNO is used as the most-significant 23 bits of the physical address of the page (and the least-significant 11 bits are zero).

The segment tables and page tables are indexed by the 30-bit, virtual address (VA). The physical address (PA) is calculated as follows; VA<0:9> is interpreted as a single-word offset from the address contained in the STP. The physical address of the STE is STP+⊂VA<0:9>||2*0⊃. If

absolute addressing is not selected and the address of the STE is greater than or equal to the contents of STL then a hard trap occurs. If the selected STE is null then a hard trap occurs. STE. PGNO specifies the physical page number of the desired page table, that is, the desired page table starts at physical address ⊂STE. PGNO || 11*0⊃. VA<10:18> is interpreted as a single-word offset from the beginning of the page table. The. physical address of the PTE is, therefore, ⊂STE. PGNO || VA<10:18> || 2*0⊃. If the selected PTE is null then a hard trap occurs. PTE. PGNO specifies the physical page number of the desired page (i.e., the page starts at physical address ⊂PTE. PGNO || 11*0⊃). VA<19:29> specifies the quarter-word offset from the beginning of the page. The physical address is, finally, PA=⊂PTE. PGNO || VA<19:29>⊃.

In general, an address transformation involves two memory references, the first to the segment table, the second to the page table. No memory reference is needed for the STP or STL since they are hardware registers inside the processor. Two page *map* caches inside each processor contain (for the most recently used pages) the complete translation from virtual page address to physical page address. One page map is for addresses of instructions, the other for addresses of data. Whenever a necessary translation is not resident in a page map, the necessary entry is fetched from memory and placed in the page map. Another page map entry may be evicted in the process. The evicted entry is not written out to memory (because it cannot have changed).

The processor hardware actually contains two sets of segment table pointer/limit registers, one set for the executive (EXECSTP and EXECSTL) and the other set for the user (USERSTP and USERSTL). A pointer/limit pair specifies an *address space* (i.e., a segment table/page table/page mapping). The address space specified by EXEC_STP and EXECSTL registers is called the *executive address space.* Similarly, the USER_STP and USERSTL registers specify the user *address space.* The CRNT_MODE and PREV_MODE fields of the PROCSTATUS word determine which address space is referenced during an address calculation (see Sections 2.5.1 and 2.4). Each hardware page map entry contains a *base-bit* which identifies which of the two address spaces (executive or user) the entry is associated with.

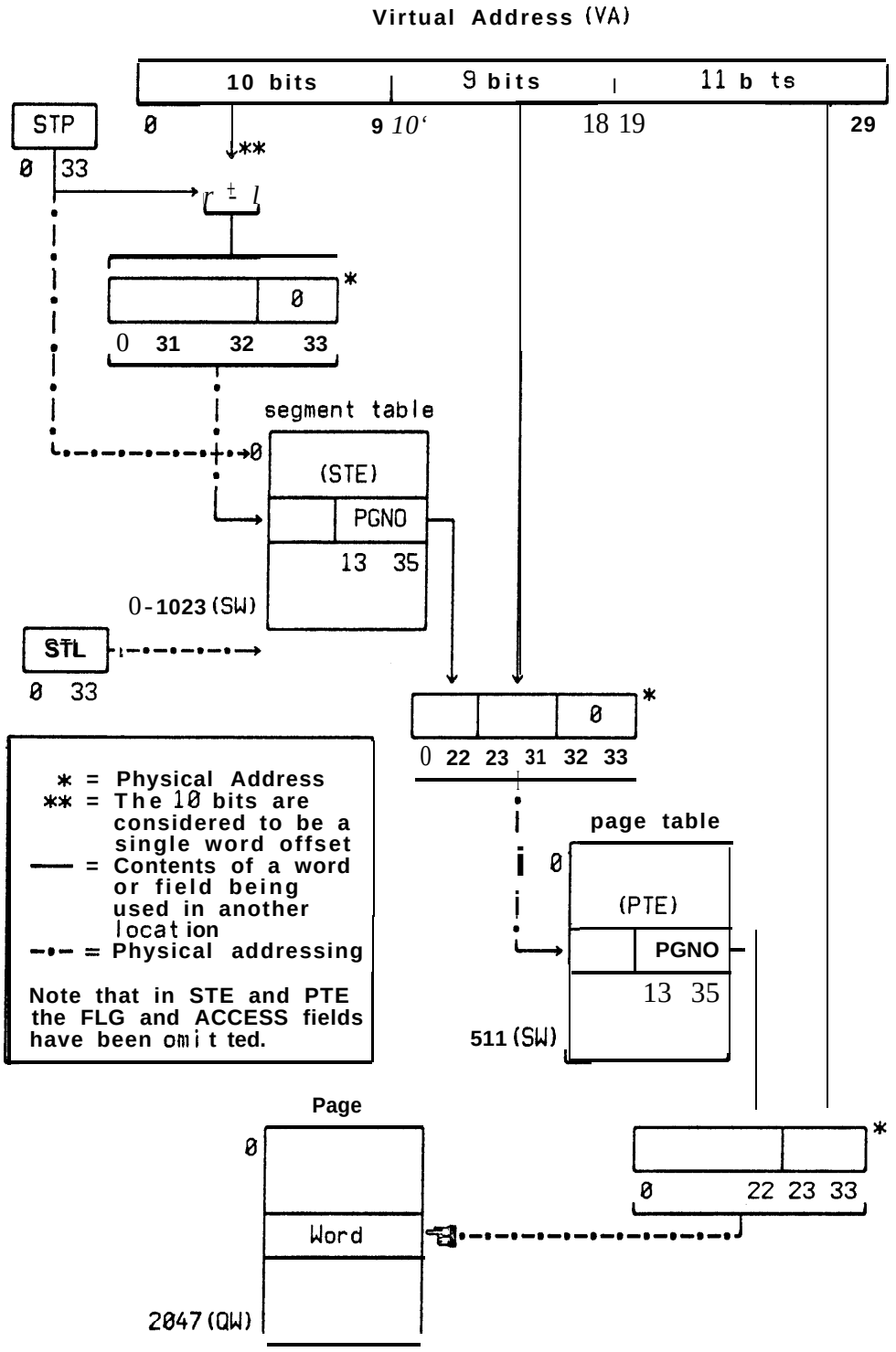

Figure 2-5
Virtual-to-Physical Address Translation

2.3.1 Flag Bits: The FIX-field

Each STE and PTE has a 7-bit FLG field. This field is used to indicate whether the table entry is valid and to record software flags. FLG<0> is called the VALID bit. If VALID=0 then the STE (or PTE) is considered to be a null entry; that is, it specifies no page. If VALID=1 then the STE (or PTE) is not null and is interpreted as a pointer to a physical page as described in Section 2.3.

The bits of FLG<1:6> are reserved for software flags . They can be used by programs (e.g., an operating system) to record information concerning the STE or PTE. They have no defined function within the architecture.

2.3.2 Access Modes

Both STEs and PTEs contain an ACCESS field. STE.ACCESS is unused. PTE.ACCESS, however, specifies any restrictions on accessing the page pointed to by the PTE. PTE. ACCESS can distinguish pages used for instructions and those used for data. It also controls when data cache entries are allocated and when changes to the data cache go through to physical memory. (The cache is discussed in Section 5.15). Many different high-level access modes (e.g., "local data" and "static code") can be specified using combinations of the ACCESS bits.

It should be noted that absolute addressing (see Section 2.3) does not utilize the access modes in the standard way. This is because absolute addressing bypasses the segment table/page table address transformation. The approach to access modes for absolute addressing is discussed in Section 2.3.2.1.

INSTRUCTIONS — PTE.ACCESS<0> specifies whether a word on the indicated page may be used as an instruction. If INSTRUCTIONS=0 then a hard trap will occur when a location from the indicated page is accessed as an instruction.

DATA — PTE. ACCESS& specifies whether a word on the indicated page may be used as data. If DATA-O then a hard trap will occur when a location from the indicated page is accessed as an operand of an instruction (except as noted in the instruction descriptions, Section 5).

READALLOCATE — PTE.ACCESS<2> indicates the course of action after encountering a read miss. If READ_ALLOCATE=1 then any read miss will allocate and fill a data cache entry. If READ_ALLOCATE=0 then a read miss will not allocate a data cache entry, but will cause data to be read directly from memory.

WRITEALLOCATE — PTE.ACCESS<3> indicates the course of action after encountering a write miss. If WRITE_ALLOCATE=1 then any write miss will allocate and update a data cache entry. If WRITE_ALLOCATE=0

then a write miss will not allocate a data cache entry. All write hits will simply update the data cache entry.

WRITE-ONLY — PTE.ACCESS<4> is used to prohibit reading from a page that is write-only. Reading of an operand from a page marked with WRITE-ONLY-1 will cause a hard trap. (Note that WRITE_ONLY=1 does not necessarily mean that the page in question can be written into; that is controlled by the WRITEALLOCATE and WRITE-THROUGH bits.)

WRITE_THROUGH — PTE.ACCESS<5> controls the updating of memory upon a write to the data cache. If WRITE_THROUGH=1 then any write will update memory. If the write is a data cache hit then the data cache will be updated as well. If the write is a data cache miss, then a data cache entry will be allocated and written if and only if WRITE_ALLOCATE=1.

Certain combinations of access bits are given special meanings by the hardware. The combination WRITE_ALLOCATE=0 and WRITE_THROUGH=0 specifies that a page is *read-only*. An attempted write to a read-only page will cause a hard trap. The combination of INSTRUCTIONS=0 and DATA=0 specifies an l/O *page*. If an instruction other than an I/O instruction operates on an I/O page then a hard trap will occur.

Various combinations of the above six bits provide useful, high-level access modes. A page may be specified to be for *local data* with the combination DATA=1, WRITE_ALLOCATE=1, and READ_ALLOCATE=1. A data cache miss caused by reading an operand from a local-data page causes the missed word to be read from memory and placed in the data cache. Writes to local-data pages do not necessarily write through to main memory. Whenever it is important that the memory shadow of a local-data page be made identical to the cache, cache control instructions must be executed to update memory, It is intended that the private variables of a process be identified as local-data pages. (All other access bits are zero.)

Cached read data may be specified by DATA= 1 and READ_ALLOCATE= 1. A data cache miss in a cached-read-data page causes the missed word to be read from memory and placed In the data cache. No writes are allowed to a cached-read-data page because WRITE_ALLOCATE=0 and WRITE-THROUGH-O. Instructions cannot be fetched from a cached-read-data page. (All other access bits are zero.)

Static *code* is specified by INSTRUCTION&l, DATA=1, and READ_ALLOCATE=1. A static-code page is similar to a cached-read-data page; however, locations on a static-code page can be accessed as instructions. It is intended that shared routines will be identified as static-code. (All other access bits are zero.)

Shared data is indicated by DATA=1 and WRITE_THROUGH=1. Words from shared-data pages are never placed in the data cache. A write to a shared-data page writes through to main

memory without writing in the data cache (WRITE_ALLOCATE=0), and a read from a shared page reads directly from main memory (provided that the data cache does not already contain the word). Locations that are heavily shared by multiple processors are intended to be on shared-data pages, eliminating the necessity to perform repeated cache sweeps when passing small amounts of data between processors. (All other access bits are zero..)

The S-l hardware does not check for illegal combinations of access bits. Such checking should be performed by operating system software when setting up PTEs.

2.3.2.1 Access Modes and Absolute Addressing

When absolute addressing is selected (STP < 128) no choice is given for the access bits. Instead, the bits INSTRUCTIONS= 1, DATA=1, READALLOCATE= 1, WRITE-ALLOCATE=], WRITE_ONLY=0, and WRITE_THROUGH=0 are always used. However, no trap will occur due to a violation of these bits while in absolute addressing mode (e.g., I/O can be done to a page even though it is not an I/O page). The bits are used only to indicate the caching algorithm for-absolute addressing.

2.3.2.2 Summary

Bit	Name	Description
0	INSTRUCTIONS	If = 0 then cannot access locations on this page as instructions.
1	DATA	If = 0 then cannot access locations on this page as data.
2	READALLOCATE	If = 1 then a read miss will allocate a cache entry.
3	WRITEALLOCATE	If = 1 then a write miss will allocate a cache entry.
4	WRITE-ONLY	If = 1 then cannot read an operand from this page.
5	WRITE-THROUGH	If = 1 then any write will update memory.

Table 2-3
Bits of STE.ACCESS and PTE.ACCESS

Use	Combination (Bits specified = 0)
Read Only	WRITEALLOCATE, WRITE-THROUGH
I/O Page	INSTRUCTIONS, DATA

Table 2-4
Special Defined Combinations of ACCESS bits

Use	Combination (Bits specified = 1)
Local Data	DATA, WRITEALLOCATE, READALLOCATE
Cached Read Data	DATA, READALLOCATE
Static Code	INSTRUCTIONS, DATA, READALLOCATE
Shared Data	DATA, WRITE-THROUGH

Table 2-5
Useful Combinations of ACCESS bits

2.4 Address Contexts

Section 2.3 describes the existence of the two address spaces maintained in the S-l architecture, executive and user. Instructions, however, do not refer directly to either the user or executive address space. They refer to the current or previous address space.

When a program (either executive or user) refers to itself or its data (Le., its own address space), it refers to the current *address space.* Access to the current address space is controlled by PROC-STATUS. CRNT-MODE. (See Section 2.5.1 for a description of PROCSTATUS.) If CRNT_MODE=0 then the current address space is the user address space. If CRNT_MODE=1 then the current address space is the executive address space. User programs operate exclusively in the current address space with CRNT_MODE=0.

Executive programs may be called by other programs (both user and executive) as the result of any one of various traps (see Section 6). In this situation the executive program is able to refer to the address space of the program that called it. The calling program's address space is called the previous address *space.* Access to the previous address space is con trolled by PROC-STATUS. PREV_MODE in the same way that PROCSTATUS. CRNT-MODE controls the access to the current address space (PREV_MODE=0 gives user address space, PREV_MODE=1 gives executive address space). User programs cannot access the previous address sp ace.

Instruction operands select between the current and previous address space by means of the P-bit in extended operands and indirect address pointers. The P-bit is discussed in Section 4.2.6.

Current (previous) *context* includes both the current (previous) address and the current (previous) register file. PROC-STATUS, CRNT_FILE (PROC-STATUS, PREV_FILE) specifies which register file should be accessed when an addressing calculation specifies the current (previous) address space.

2.4.1 Shadow Memory

The first thirty-two single-words of an address space are called shadow memory. This term is derived from the fact that they overlap or are shadowed by the currently selected register file (because references to the first 128 quarter-words of an address space are normally interpreted as references to the current register file instead). Shadow memory cannot be accessed by the user, but is accessible to the executive (when accessing the previous address space).

The use of shadow memory is controlled by the USE-SHADOW-PREV bit in the processor status word (See Section 2.5.1). When USE_SHADOW_PREV= 1, all references to addresses 0 .. 127 in the previous context will cause the shadow memory of the previous context to be accessed. When USE_SHADOW_PREV=0, the previous register file is accessed instead.

Assume the USE-SHADOW-PREV bit in the processor status word is set. The following instruction loads the second shadow memory word from the previous context into the location

whose (hypothetical) symbolic name is SECOND.

```
        MOV SECOND,<!P4>        ;"!P" means access previous context
```

2.5 Status Words

Status words partially define the current state of a program's execution. They contain information about current and previous contexts, and about conditions such as arithmetic overflow and trace modes. There are two types of status: processor status and user status. As a general rule, processor status contains privileged information which the user may not modify, and user status contains per-user information which the user program may modify at will. (The user status does not apply just to user mode programs, Programs running in executive mode are also affected by the user status. However, the user status is automatically changed whenever a switch from user mode to executive mode occurs, and so the executive may be thought of as a distinct "user" so far as user status is concerned.)

2.5.1 Processor

The processor status word (PROCSTATUS) contains information about the current state of a process. This includes information such as the extent of the stack and the currently accessible address space. The fields in their order of occurrence from most-significant bit to least-significant bit are shown below.

CRNT_FILE<0:3> Current register file. This is the number of the register file that will be accessed in all references to the current context. Note that REG_FILE[0] is reserved for use by hardware and microcode, and so CRNT_FILE will normally have a non-zero value.

PREV_FILE<0:3> Previous register file. This is the number of the register file that will be accessed in all references to the previous context. (Such references may be additionally controlled by the USE-SHADOW-PREV bit, however.) Note that REG_FILE[0] is reserved for use by hardware and microcode, and so PREV_FILE will normally have a non-zero value.

USE_SHADOW_PREV Use shadow memory. When set to one, this bit causes references to memory locations 0 . . 127 in the previous context to reference shadow memory instead of registers. The user is not allowed to access the previous context (P-bit=1 will cause a hard trap to occur), and therefore the user cannot access shadow memory. See Section 2.4.1 for more on shadow memory. Address spaces and the P-bit are discussed in Section 4.2.6.

PRIO<0:2> Processor priority level. Interrupts with INTUPT_AT_LVL<*i*>=1 where *i* < PRIO will cause the S-l to be interrupted. See Section 5.16 for a description of the interrupt architecture.

EMULATION<0:1> Emulation mode. When equal to zero, causes the S-l native instruction set to be executed. When non-zero, specifies the emulation of one of three other instruction sets.

TRACE_ENB	Trace-trap enable. Used to enable trace-traps after each instruction. See Section 6.3 for a description of the trace feature.
TRACE_PEND	Trace-trap pending.. Used to indicate that a trace-trap is pending. See Section 6.3 for a description of the trace feature.
CRNT_MODE	Current mode. Specifies whether the current context is executive or user. Zero means user, one means executive.
PREV-MODE	Previous mode. Specifies whether the previous context is executive or user. Zero means user, one means executive.
UNUSED<O: 17>	Reserved for future use.

Changing the processor status word causes a change in state for the currently executing process. This change of state often involves changing the current context (see Section 2.4). In order to make this change of context correctly, PROCSTATUS cannot be loaded in its entirity from an arbitrary 36-bit word. If the execution of an instruction causes the loading of a new PROC-STATUS (e.g., traps, interrupts), then the new PREV-MODE must be loaded from the old CRNT-MODE. Similarly, the new PREVFILE must be loaded from the old CRNT_FILE. The PREV-MODE and PREV_FILE fields of the word which is being loaded into PROCSTATUS are ignored. This operation is called loading *partial processor status*. PROCSTATUS is always loaded in this way unless specifically mentioned otherwise. The only instructions that load the entire PROC-STATUS word are RETFS and WFSJMP (see Sections 5.9 and 5.14).

A similar process is involved when loading a new PROCSTATUS while checking for trace-traps (see Section 6.3). In this case a change in state occurs when the TRACE-.PEND bit of PROC-STATUS is updated during the instruction-execution sequence.

2.5.2 User

User status is contained in a single register named USER-STATUS. It contains a large number of subfields, each of which is described below. CARRY and the error-bits FLT_OVFL, FLT_UNFL, FLT_NAN, INT-OVFL, and INTZDIV are described as being *not sticky*. This means that they are either set or cleared by any instruction that can affect them. As an example, if an ADD instruction produces an integer overflow while trapping is disabled (INT_OVFL_MODE=1), the INT-OVFL bit of PROCSTATUS will be set to one. If a MULT instruction is then executed and no integer overflow occurs during the multiplication, INT-OVFL will be reset to zero. Each error bit is also reset when the appropriate trap is initiated, before a copy of USER-STATUS is saved on the stack. The conditions that affect CARRY and the error-bits for both integer and floating-point instructions are described in Section 5.2.3 and Section 5.3.2. The fields of USER-STATUS are shown below in order of occurrence from most significant to least significant.

SP_ID<0:4> Stack-pointer identity. Specifies the register that will be used in all references to the stack-pointer (SP). The stack-limit register (SL) is considered to be the next contiguous register. SP_ID=3 or SP_ID=31 is illegal. See Section 2.2.3.2. for details.

CARRY Carry-out of arithmetic operations. Set to zero or one by the most recently executed integer arithmetic instruction. Note that CARRY is not sticky. See Section 5.2.3.1.

FLT_OVFL Floating overflow. Always set by floating-point arithmetic instructions. Set to one if the result of the most recently executed floating-point instruction was greater than or equal to MAXNUM (i.e. MOVF). This bit is not sticky. See Section 5.3.2.1.

FLT_UNFL Floating-underflow. Always set by floating-point arithmetic instructions. Set to one if the result of the most recently executed floating-point instruction was less than or equal to MINNUM+ 1 (i.e. MUNF). This bit not sticky. See Section 5.3.2.1.

FLTNA N Floating-point result is "Not A Number" (NAN). Always set by floating-point arithmetic instructions. Set to one whenever NAN is the result of a floating-point operation. This bit is not sticky. See Section 5.3.2.

INT_OVFL Integer overflow. Set to one when the result of the most recently executed integer arithmetic instruction is greater than or equal to MAXNUM. This bit is not sticky. See Section 5.2.3.2.

INTZDIV Integer-zero-divide. Set to one when a divide-by-zero has occurred in the most recently executed integer instruction . This bit is not sticky. See Section 5.2.3.3.

FLT_OVFL_MODE<0:1> Determines the action that is taken when floating overflow occurs. FLT_OVFL_MODE=0 causes the instruction to soft-trap without storing a result. FLT_OVFL_MODE= 1 causes the floating point infinity of correct sign (either OVF or MOVF) to be stored as the result. FLT_OVFL_MODE=2 causes a floating-point number of correct mantissa and sign, but with wrapped-around exponent to be stored as the result. FLT_OVFL_MODE=3 is undefined (an attempt to set FLT_OVFL_MODE to 3 will cause a hard trap).

FLT_UNFL_MODE<0:1> Determines the action that is taken when floating underflow occurs. FLT_UNFL_MODE=0 causes the instruction to soft-trap without storing a result. FLT_UNFL_MODE= 1 causes the floating point

infinitesimal of correct sign (either UNF or MUNF) to be stored as the result. FLT_UNFL_MODE=2 causes a floating-point number of correct mantissa and sign, **but** with wrapped-around exponent to be stored as the result. FLT_UNFL_MODE=3 is undefined (an attempt to set FLT_UNFL_MODE to 3 will cause a hard trap).

FLT_NAN_MODE<0:1> Determines the action that is taken when NAN is the result of a floating-point operation. FLT_NAN_MODE=0 causes the instruction to soft-trap without storing a result. FLT_NAN_MODE=1 causes NAN to be stored as the result. FLT_NAN_MODE=[2,3] are undefined (an attempt to set FLT_NAN_MODE to 2 or 3 will cause a hard trap).

INT_OVFL_MODE Determines the action that is taken when integer-overflow occurs. INT_OVFL_MODE=0 causes the instruction to soft-trap without storing a result. If trapping is disabled (INT_OVFL_MODE=1), all instructions except for SHFA to the (true) left store the low-order bits of the result. SHFA to the (true) left stores the correct sign followed by the low-order bits of the (true) result.

INT_Z_DIV_MODE Determines the action that is taken when integer divide-by-zero occurs. INT_Z_DIV_MODE=0 causes the instruction to to soft-trap without storing a result. INT_Z_DIV_MODE=1 causes zero to be stored as the result.

RND_MODE<0:4> Rounding mode. Selects the rounding mode to be used. See Section 5.3.1 for a description of the rounding modes.

UNUSED<0:7> Reserved for future use.

FLAGS<0:3> Contains various software-definable flag bits. These bits have no defined meaning in the architecture.

3 Data Types

Data in the S-l is uniformly represented as quarter-, half-, single- or double-words. For many operations it is useful to interpret the bits in these words in various ways. Each of these ways of viewing data constitutes a *data type.* Instructions may interpret their operand data as being of a certain type. The *same* data may be interpreted in different ways by different instructions.

S-l instructions operate on the following data types: boolean, integer (signed and unsigned), floating-point, indirect address pointer, byte (single-word and double-word), byte pointer, block, and flag. To be fetched as the operand of an instruction, data must be on pages marked with DATA= 1 (see Section 2.3.2). The data types are described below.

3.1 Boolean

The *boolean* data type is a bit vector in any of the four standard precisions (quarter-word, half-word, single-word, and double-word). The bits are numbered from left to right, as shown in the figures of Section 2.1.

For example, the following assembles as the QW bit vector 001000101.

```
105
```

3.2 Integer

The S-l has two different formats for integers: unsigned and signed. *Unsigned integers* represent only non-negative quantities while *signed integers* can represent both negative and non-negative quantities in two's-complement notation. Either format may be represented in any of the four standard precisions (quarter-word, half-word, single-word, and double-word). For example, quarter-word, unsigned integers can represent quantities in the range 0 . . 511 whereas quarter-word, signed integers represent quantities in the range -256. . 255.

For ease of description the largest positive signed integer in a given precision is termed *MAXNUM*. Correspondingly, the negative signed integer with the largest magnitude is termed *MINNUM*. For example, in quarter-word precision MAXNUM=255(377_8) and MINNUM=-256 (400_8). More generally, in any precision MAXNUM has all bits but the leftmost set to one, and MINNUM has all bits but the leftmost set to zero. (This is a consequence of the nature of the two's-complemen t representation of integers.)

The following shows signed and unsigned interpretations of various integer quarter-word constants.

```
105     ;signed and unsigned interpretation is 105
673     ;unsigned 673, signed -105
-105    ;unsigned 673, signed -105
-1      ;unsigned 777, signed -1
```

The bit pattern for the first example is 001000101, and for the next two is 110111011. The leftmost bit is interpreted as the sign bit (1=negative) in the signed case. Note that in all precisions the signed value -1 has all bits set to one.

3.3 Floating-point

S-l floating-point numbers are always (implicitly) normalized and may be represented in three different precisions (half-word, single-word, and double-word). The floating-point representation is made up of three fields SIGN, EXP, and MANT. These fields, along with an implicit *hidden bit*, determine the value of the floating-point number. The formats are: ⊂SIGN || EXP<0:5> || MANT<0:10>⊃ for half-words, ⊂SIGN || EXP<0:8> || MANT<0:25>⊃ for single-words, and ⊂SIGN || EXP<0:10> || MANT<0:59>⊃ for double-words.

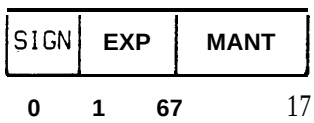

Figure 3-l
Half-word Floating-Point Format

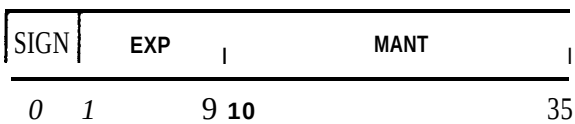

Figure 3-2
Single-word Floating-Point Format

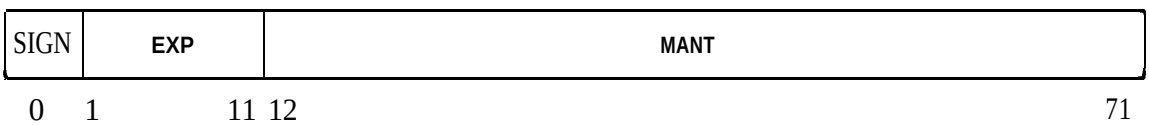

Figure 3-3
Double-word Floating-Point Format

SIGN represents the sign of the floating-point number (O=non-negative, 1=negative). EXP specifies the exponent. For half-word precision, EXP is the exponent in excess-32 format. For single-words, EXP is the exponent in excess-256 format, and for double-words, EXP is the exponent in excess- 1024 format. SIGN, MANT, and the hidden-bit make up the *mantissa.* The hidden-bit is always the complement of SIGN, so for positive numbers the hidden-bit equals one. The mantissa, for postive numbers, can be written as the concatenation of SIGN, the binary-point, the hidden-bit, and MANT, that is, mantissa=⊂SIGN || . || hidden-bit || MANT⊃ with "." representing the binary-point. (This is, of course, a slight abuse of the concatenation notation, as the binary point is not really a field.) Positive floating-point numbers have their mantissa in the range 0.5≤mantissa< 1. Floating-point zero is represented as integer zero (which is an exception to the SIGN/hidden-bit correspondence, because zero has SIGN=0 and hidden-bit=0).

The following shows the octal representation of some non-negative floating point numbers in various precisions.

```
0                       ;0.0 in all precisions
204000                  ;1.0 HW
004000                  ;2↑(-32) HW
377777                  ;(2↑32)-(2↑20) HW
200400,,0               ;1.0 SW
000400,,0               ;2↑(-256) SW
377777,,-1              ;(2↑256)-(2↑228) SW
200100,,0↔0             ;1.0 DW
000100,,0↔0             ;2↑(-1024) OW
377777,,-1↔-1           ;(2↑1024)-(2↑962) DW
```

The full specification of a floating-point number (including both positive and negative numbers) is as follows. Note that the one's-complement and two's-complement operations are performed in the same number of bits as the argument to the operation.

Definition	Positive Numbers	Negative Numbers
mantissa	⊂SIGN‖ . ‖ hidden-bit ‖MANT⊃	*2's-comp*(⊂SIGN‖ . ‖ hidden-bit ‖MANT⊃)
exponent	EXP - excess	*1's-comp*(EXP) - excess
number	mantissa $*(2^{exponent})$	- mantissa $*(2^{exponent})$

Floating-point *zero* is represented as integer zero

Table 3-l
Floating-Point Representation

Negative floating-point numbers have hidden-bit=0 because SIGN= 1. Negative number mantissas are in the range 0.5<mantissa≤1. Note that the above definition specifies that mantissas are always non-negative (hence the minus sign in the above table description of the value of a negative number).

The following shows the octal representation of some negative floating point numbers in various precisions.

```
574000                  ;-1.0 HW
```

```
774000              ;-(2↑-32) HW
400000              ;-(2↑32) HW
577400,,0           ;-1.0 SW
777400,,0           ;-(2↑-256) SW
400000,,0           ;-(2↑256) SW
577700,,0↔0         ;-1.0 DW
777700,,0↔0         ;-(2↑-1024) DW
400000,,0↔0         ;-(2↑1024) DW
```

The floating-point format permits a simple translation between positive and negative floating-point numbers. The floating-point representation of -x is equal to the two's-complement of the floating-point representation of x. (The entire word is two's-complemented, ignoring sub-field boundaries. The hidden bit is determined by the new SIGN bit.)

An outline for a proof that two's-complement negation works correctly on floating-point numbers follows. 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, 1.0 and the floating-point number was $-2^{exponent}$. When this number is two's-complemented, the MANT-field is still zero but the EXP-field is two's-complemented. The mantissa becomes 1/2 and the carry from the fraction has increased the exponent by one. This gives $(1/2)*2^{exponent+1}$ or $2^{exponent}$, 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 1/2 and the floating-point number was $(1/2)*2^{exponent}$. When this number is two's-complemented, the MANT-field is still zero but the EXP-field is two's complemented. The mantissa becomes 1.0 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 $-(1.0)*2^{exponent-1}$ or $-(1/2)*2^{exponent}$, the negative of the original number.

Besides zero, there are five floating-point numbers that have special meanings attached to them. The positive, floating-point number with the greatest magnitude (in a given precision) has the meaning of *positive infinity* This number is designated *OVF*. (It should be noted that the largest, positive, signed-integer, in a given precision, is termed MAXNUM. Correspondingly, the negative, signed-integer with the largest magnitude is termed MINNUM. It is often convenient to speak of a floating-point number in terms of the signed-integer with the same bit representation. For example, OVF is the same as MAXNUM in that if MAXNUM is interpreted as a floating-point number, it turns out to be the largest floating-point number (i.e., OVF).) The two's-complement of OVF (i.e., MINNUM+1) has the meaning *negative infinity*. It is termed *MOVF*. (The terms OVF and MOVF come from *overflow* and *minus overflow*, respectively.) The smallest, positive, floating-point number has the meaning of *positive infinitesimal* and is termed *UNF;* it has the same bit representation as the integer 1. The largest, negative, floating-point

number has the meaning of *negative infinitesimal* and is termed *MUNF*. MUNF is the two's-complement of UNF, and so has the same bit representation as the integer -1. (The terms UNF and MUNF come from and *minus underflow*, respectively). The floating-point number with the same bit representation as MINNUM has the meaning of *undefined.* It is termed *NAN*, meaning *not a number.* Floating-point instructions take these special interpretations into account. Certain bits of USER-STATUS control the action taken when one of the exceptions associated with these special numbers occurs (e.g, overflow with OVF). See Section 2.5.2 for details of USER-STATUS and see Section 5.3.2 for details of floating-point exception handling.

Name	Meaning	Equivalent integer representation
OVF	Positive overflow	MAXNUM
MOVF	Negative overflow	MINNUM+ 1 (-MAXNUM)
UNF	Positive infinitesimal	+1
MUNF	Negative infinitesimal	-1
NAN	Indeterminate ("not a number")	MINNUM

Table 3-2
Floating-Point Exception Representation

NOTE: The signed integer (Section 3.2) and floating point formats employed in the S-l have an important and useful property: the *same* algorithms can be used to compare the value of a datum interpreted in *either* format. However, special floating-point symbols such as OVF and NAN are not properly interpreted by integer instructions.

3.4 Indirect Address Pointer

An *indirect address pointer* (IAP) is a single memory word that is interpreted as a pointer into memory. Its format is shown below. IAP. P denotes the address space being referenced. IAP. IREG and IAP. ADDR together describe the memory location to be addressed. The IAP, as used for indirect addressing, is discussed in Section 4.2.5. The P-bit is described in Section 4.2.6.

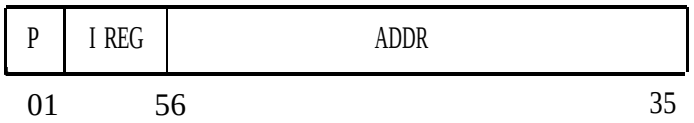

Figure 3-4
Indirect Address Pointer

3.5 Byte

A *single-word byte* is a bit vector with a length in the range 0 . . *36* . A *double-word byte* is a bit vector with a length in the range 0 . . 72. (A zero-length byte of course contains no information, but it is permitted to use a byte pointer specifying such a byte.) The position and length of a byte are specified by a byte pointer, as described in Section 3.6.

3.6 Byte Pointer

A *byte pointer* completely specifies a byte somewhere in memory. The byte pointer consists of two single-words. The first single-word is an indirect address pointer (IAP). The IAP specifies a memory single-word or double-word which contains the byte. The second single-word of the byte pointer is a *byte selector*. It has two half-word fields POSITION and LENGTH (<POSITION<0:17>|| LENGTH& 17>>). POSITION is the bit number of the first bit in the byte. LENGTH is the number of bits in the byte.

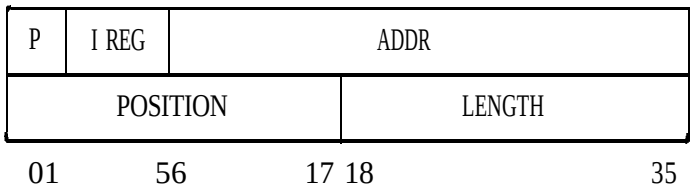

Figure 3-5
Byte Pointer

3.7 Block

A block consists of a contiguous list of words. The words may be any of the four standard precisions (quarter-word, half-word, single-word, double-word). Ail of the words within a block, however, are of the same precision. Some instructions which operate on blocks implicitly treat the elements of the block as being of some other specific type; for example, STRCMP (Section 5.13) treats the block elements as signed integers.

3.8 Flag

The *flag* is a single-word data type with only two values: the bit representations which are all bits zero and ail bits one (i.e., integer 0 or -1 in two's-complement notation). A flag of ail ones means *true,* ail zeros means *false.*

4 Instruction Formats and Addressing Modes

4.1 Instruction Classes

The S-l provides a rich variety of ways in which the operands for a given operation may be accessed. These ways are called *addressing modes.* All S-l instructions can be specified with no more than three single-words. The first word specifies the instruction selected. In general, the second and third words are optional in that they specify extended addressing modes if needed. Therefore, depending on the number of extended operands, S-l instructions may consist of one, two, or three words.

The general format for the first word of an instruction is ⊂OPCODE<0:11> || OD1<0:11> || OD2<0:11>⊃. The first twelve bits specify the opcode, the second twelve describe how the first operand is accessed, and the last twelve bits describe how the second operand is accessed. (Note that in jump instructions the second operand is called J, not OD2.)

The opcode indicates which instruction is being selected. It also specifies the precision of the arguments (the data values the instruction operates on). Depending on which instruction is selected, the opcode may also indicate more information so as to fully describe the instruction (e.g., which direction to shift, what condition to skip on, etc.). Sections 4.1.2, 4.1.1, 4.1.3, 4.1.4, and 4.1.5 describe the five classes to which instructions belong: two-address (XOP), three-address (TOP), skip (SOP), jump (JOP), and hop (HOP).

OD1 and OD2 are *operand descriptors* (OD). They describe the arguments upon which the instruction operates. The full specification of an operand may require an extra instruction-word per argument. This use of extra instruction-words is termed *extended addressing.* The process whereby the value described by the OD is determined is called *operand evaluation* The result of the operand evaluation of OD1 is called OP1, and that for OD2 is called OP2. The various means of describing operands (addressing modes) are discussed in Section 4.2.

The evaluation of all operands (including jump or skip destinations) logically occurs before the execution of the instruction and before the PC is updated. The order of operand evaluation is undefined. Operand evaluation produces no side effects.

4.1.1 Two-Address (XOP)

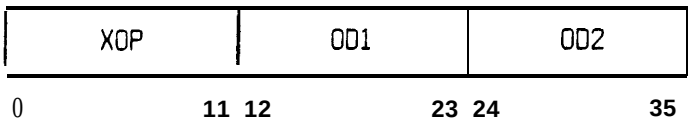

Figure 4-l
XOP

The two-address instructions are generally used to specify operations that involve one source and one destination. Typically OP1 is used as the destination and OP2 as the source. The XOP field is the opcode. OD1 and OD2 are the ODs that describe the arguments to the instruction. The results of the operand evaluation of OD 1 and OD2 are OP 1 and OP2, respectively. When an XOP instruction stores two results, it stores OP2 before OP1.

Some XOP instructions leave one or both operand descriptors unused. As a rule, an XOP instruction with only one operand uses OD1, and OD2 must be zero.

An XOP instruction is written as the instruction mnemonic followed by OD 1 and OD2 specifications, in that order. For example, let X and Y be SWs. The following illustrates an XOP instruction which sets X to Y (that is, the single-word register or memory location whose symbolic name is X is made to contain the contents of Y).

```
MOV X,Y          ;X is the destination, Y is the source
```

If only one operand descriptor is specified, then FASM will use it for both OD1 and OD2, or just OD1, depending on whether or not both operands are used by the instruction.

```
INC COUNT        ;COUNT←COUNT+1; INC uses both OD's
RUS RTA          ;RTA←USER_STATUS; RUS uses only OD1
```

4.1.2 Three-Address (TOP)

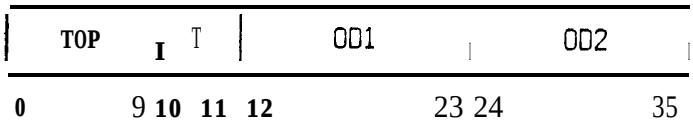

Figure 4-2
TOP

Three-address instructions allow the specification of three arguments (generally two sources and one destination). They specify two general memory locations (which may, of course, be registers) and possibly one of the registers RTA or RTB. This format provides most of the power of full three-address instructions (instructions specifying three general memory locations) but only costs two bits in the instruction word (for T) as compared to twelve bits for a third general operand descriptor.

The TOP field is that portion of the opcode that indicates the instruction selected, the precision, and any- other information needed to fully specify the operation. OD 1 and OD2 are general operand descriptors. OP1 and OP2 are the results of the operand evaluations of OD 1 and OD2, respectively. T specifies how OP1, OP2, RTA, and RTB are to be used as arguments to the operation. The first argument to the operation is called S1, the second is called S2, and the third DEST. In most (but not all) cases the instruction takes S1 and S2 as input and uses DEST as the location for its output. When a TOP instruction stores more than one result, it stores S2 before S 1, and S 1 before DEST. The following table shows how the T field selects S 1, S2, and DEST.

T	DEST	1	S2
00	OP1	OP1	OP2
01	OP1	RTA	OP2
10	RTA	OP1	OP2
11	RTB	OP1	OP2

Table 4-l
Specification of S 1, S2, DEST

A TOP instructionis written as an opcode mnemonic followed by DEST, S1, and S2 in that order. For example, let X and Y be SWs. The following shows the various T fields.

```
ADD X, X, Y      ;T field = 00; X←X+Y
ADD X, RTA, Y    ;T field = 01; X←RTA+Y
ADD RTA, X, Y    ;T field = 10; RTA←X+Y
ADD RTB, X, Y    ;T field = 11; RTB←X+Y
```

In the case T=0, where by definition OP1 is used for both S1 and DEST, it is not necessary to

write the operand twice. Thus the first example above may be written:

```
ADD X,Y          ;T field = 00;  X←X+Y
```

FASM automatically fills in the T field based on the operand descriptors written after the opcode mnemonic.

The selection of DEST, S1, and S2 by the T field is asymmetric with respect to OD 1 and OD2. As a general rule (which has exceptions), whenever a TOP instruction is not symmetric with respect to S1 and S2, it comes in two forms, an ordinary form and a "reverse" form. The reverse form is just like the normal form except that the use of S1 and S2 is reversed.

For example, one can write:

```
SUB X,RTA,Y      ;X←RTA-Y
```

but one cannot write:

```
SUB X,Y,RTA      ;illegal!
```

because no T-field value corresponds to that arrangement of operands. One can get the intended effect by using the reverse form of the SUB instruction.

```
SUBV X,RTA,Y     ;X←Y-RTA
```

because whereas SUB computes S 1-S2, SUBV computes S2-S1.

4.1.3 Skip (SOP)

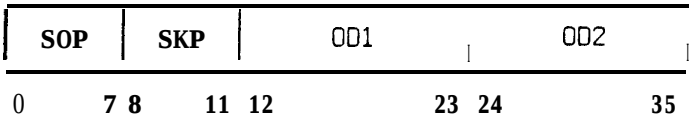

Figure 4-3
SOP

Skip instructions are used for short range transfers of control. The format allows a forward skip of 1 . . 7 single-words, a stationary skip of *zero* single-words, or a backward skip of 1 . . 8 single-words relative to the first word of the current instruction. (In this respect the word *skip* is used more broadly than in other machine architectures, because the S-l can *skip* backwards, and forwards over more than one instruction.) The SOP field specifies the opcode (including the condition on which the skip will be taken). OD1 and OD2 are general operand descriptors and the results of their operand evaluation are OPI and OP2 respectively.

The SKP field specifies the number of instruction single-words to skip. SKP is considered to be a signed constant in the range -8 . . 7 . If the skip instruction results in *not skipping,* then control flow is not interrupted (Le., the instruction following the skip instruction is executed next). If the instruction results in *skipping,* then the next instruction to be executed has an address of PC+1+SIGNED(SKP) (i.e., the address of the skip instruction offset by SKP single-words).

A skip instruction is written as an opcode mnemonic followed by the two operand descriptors and the name of the location to be skipped to. For example, let X and Y be single-words. The following ensures that X≤Y. FASM automatically determines the PC offset in the skip instruction. (If only the larger or smaller of X and Y were of interest, then the MAX or MIN instructions might be used instead; this piece of code makes X the larger *and* Y the smaller of the two.)

```
        SKP.GEQ X,Y,NEXT        ; if X≥Y then go to NEXT
         EXCH X,Y               ; else swap X and Y
NEXT:   ...                     ;continue with program
```

. As another example, this code computes the product of all odd integers from 1 to 15.

```
        MOV X,#1                ;X counts odd integers
        MOV RESULT,#1           ;RESULT accumulates product
LOOP:   ADD X,#2                ;step X to next odd integer
        MULT RESULT,X           ;multiply it in
        SKP.LSS X,#15.,LOOP     ; if X<15. then go to LOOP
```

4.1.4 Jump (JOP)

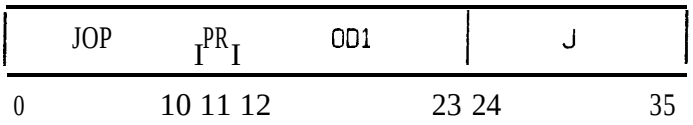

Figure 4-4
JOP

The jump instructions allow two different ways of specifying the destination of the jump, *PC-relative* and general. The choice depends on the PR bit (PR=1 for PC-relative and PR=O for general). The JOP field is the opcode and OD1 is a general operand descriptor. The result of the operand evaluation of OD1 is termed OP1. The PC-relative bit PR selects how J is to be interpreted as the jump-destination (JUMPDEST). If PR=1 then J is considered to be a signed 12-bit constant and is used as the number of single-words to offset from the PC. Therefore, JUMPDEST=PC+4*SIGNED(J); the range of a relative jump is from PC-(2048 single-words) to PC+(2047 single-words) If PR=0, JUMPDEST is set equal to the address of the operand that is computed by interpreting J as an OD-field. With PR=0 any address can be specified (at the possible expense of an-extra instruction-word). It should also be noted that with PR=0, J may not specify an immediate constant or a register.

A JOP instruction is written as the opcode mnemonic followed by the operand (if applicable) and the jump destination. For example, let X be a SW. FASM determines the value of the PR bit in the following instruction, depending on how far away the location named AWAY is from the jump instruction.

```
JMPZ.GEQ X, AWAY          ; go to AWAY if X≥0
```

4.1.5 Hop (HOP)

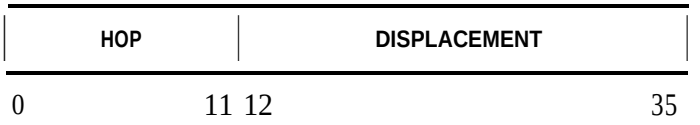

Figure 4-5
HOP

There is only one hop instruction, JPATCH. The HOP field is the opcode. It does not have an OD1 or OD2 field. Instead, bits 12 to 35 of the instruction word as used as a 24-bit signed displacement, which is added to the PC to form an unconditional-jump address.

An HOP is written as the opcode mnemonic followed by the the jump destination, as for a JOP.

```
JPATCH PATCH. AREA          ; go to PATCH.AREA
```

4.2 Addressing Modes

The addressing modes of the S-l are efficient and powerful. Many operands can be specified using only the fields in a single instruction-word, If it is necessary to access the full 2^{28} single-ward address space then extended addressing may be employed at the expense of an extra instruction single-word per extended address. Indirection is also available in (and only in) extended addressing mode.

The addressing modes were designed with both high-level and low-level languages in mind. All of the common addressing modes used in assembly language programming are available. Addressing modes designed explicitly to implement high-level language constructs have also been included. An important example of this is the concept of pseudo-registers, in which data within a small offset of a register pointer (e.g., a stack pointer) may be accessed using only a single instruction-word.

Unless otherwise stated, all addresses are quarter-word addresses. They are 30-bit integers in the range $0 . . 2^{30}-1$. *Operand evaluation* is the process of fetching the argument of an instruction. Address calculations within operand evaluation have no side effects (and are restartable). Such address calculations produce results which are truncated to the low-order thirty bits and do not affect such arithmetic flags as carry or overflow. During an instruction's execution, the PC remains unchanged.

4.2.1 Operand Descriptor Format

An *operand descriptor* (OD) is a 12-bit field of an instruction-word, and describes an argument to that instruction. The OD has three subfields: X, MODE, and F. OD. X specifies short (0) or extended (I) addressing. As a rule, if an X bit of an operand descriptor is 1 then a corresponding extended word follows the instruction word for use by that operand descriptor. (Recall, however, that in a JOP instruction with PR=1, the J (OD2) descriptor has no X bit.) If both operand descriptors have OD.X=1, then the extended word for OD2 follows the first 'instruction-word, and after that is the extended word for OD1. OD. MODE and OD. F are used to determine an addressing mode *or* to calculate a memory location. If an OD is unused in an instruction then it must be identically zero. (If it is non-zero, a hard trap will occur.)

· The numbering of the bits in the diagram below is relative to the start of the field.

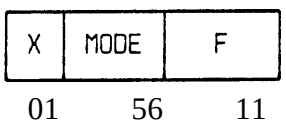

Figure 4-6
Operand Descriptor (OD)

4.2.2 Extended Addressing Formats

If an instruction requires more than a single-word to specify an operand, additional single-words called *extended-words* (EWs) are used. The possible formats of the EWs are described in the following sections.

4.2.2.1 Long-Constant Format

Long-constants are used to specify immediate values that are too large to represent in an OD. They require an additional instruction-word of the format shown below.

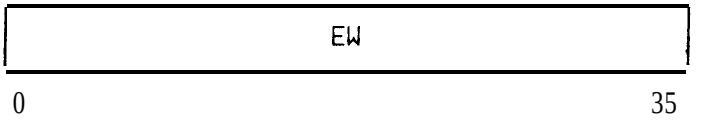

Figure 4-7
Constant Extended-Word (EW)

4.2.2.2 Fixed-Based Format

In those cases when the OD cannot specify a particular memory location, extended addressing is required. Fixed-based addressing requires an extra instruction-word (shown below).

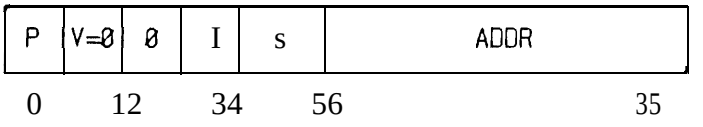

Figure 4-8
Fixed-Based Extended-Word (EW)

4.2.2.3 Variable-Based Format

When indexing through two registers, or a register and a pseudo-register, variable-based addressing must be used. Variable-based addressing uses an additional instruction-word of the format shown below.

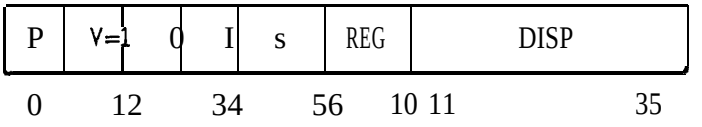

Figure 4-9
Variable-Based Extended-Word (EW)

4.2.3 Short-Operand Addressing

An operand descriptor (OD) fully describes a *short operand (SO).* If OD.X=0 *(short-operand* mode) then the argument to the instruction is exactly SO. If OD.X=1, then SO is used in later

phases of the operand evaluation procedure (see Section 4.2.4). Short-operand mode gives access to the 32 registers, short (integer) constants in the range -32.. 31 , and memory locations indexed through the registers and offset by no more than a short constant. The decision as to which of the above is to be accessed depends on the contents of OD.MODE. Only the current address space may be referenced. (See Section 2.3 for a description of the concept of address space.)

Note that OD. MODE=2 is reserved for future use and if used will result in a hard trap.

4.2.3.1 Register-Direct

OD. MODE-O gives *register-direct* mode, that is, the result of the operand evaluation (SO) is the contents of one of the 32 registers. The register number is specified by OD.F and must be in the range 0 . . 31 or a hard trap will occur. (SO=R[OD.F].)

For example, here OD 1 and OD2 are register direct. The instruction negates RTA.

```
NEG RTA          ;RTA←-RTA (same as NEG.S RTA,RTA)
```

4.2.3.2 Short-Constant

OD. MODE= 1 gives *short-constant* mode. In this case, SO=SIGNED(OD.F), which is a constant in the range -32 . . 31 .

For example, here the #0 is assembled as a short constant:

```
MOV RTA, #0      ;RTA←0
```

4.2.3.3 Short-Indexed

OD. MODE in the range 3 . . 31 gives *short-indexed* mode, which allows easy access to small memory areas indicated by registers. The memory locations that can be accessed in this addressing mode are called *pseudo-registers.* The address calculation uses R[OD. MODE1 as a base and then offsets that base by SIGNED(OD. F) single-words (i.e., range -32 . . 31 single-words). SO is the contents of the resulting address (SO=M[R[OD.MODE]+1*SIGNED(OD.F)]). If OD. MODE=3 then PC is used instead of R[3] (see Section 2.2.3.1). Note that R[0], R[1], and R[2] cannot be used in short-indexed mode because OD. MODE=0 selects register-direct mode, OD. MODE= 1 selects short-constant mode, and OD. MODE=2 is reserved and therefore hard-traps.

An interesting special case of pseudo-registers is the top few locations **on** the stack. Let SP be the stack pointer specified **by** SP_ID (and **assume** SP_ID is not 0, 1, or 2). The following instructions access stack locations in short-indexed mode. In this way local variables can be kept on the stack and easily accessed.

```
ADD -1(SP),#7          ;add 7 to top SW on stack
EXCH -2(SP),-1(SP)     ;swap top two single-words of stack
SKP.EQL -5(SP),-1(SP)  ;skip next instruction if equal
```

As another example, suppose that register R contains the address of a record structure. Then short-indexed mode can **be used** to access components of the record.

```
MOV Y,1(R)             ;move second uord of register to Y
MULT RTB,(R),2(R)      ;product of first and third words to R T B
```

4.2.3.4 Summary

MODE	Mode Name	Short-Operand(SO)	F-field Range
0	Register-Direct	R[OD.F] .	0.. 31
1	Short-Constant	SIGNED(OD. F)	-32 . . 31
2	Reserved	(hard trap)	---
3	Short-Indexed	M[PC+4*SIGNED(OD.F)]	-32.. 31
4 .. 31	Short-Indexed	M[R[OD.MODE]+4*SIGNED(OD.F)]	-32.. 31

Table 4-2
Short-Operand Mode

4.2.4 Extended Addressing

Unlike short-operand addressing, extended addressing allows an instruction to access the entire 2^{28} single-word address space. This generality requires an additional instruction-word for each extended operand.

OD. X= 1 is used to select extended addressing. OD, MODE specifies how the extended-word (EW) will be interpreted (i.e., long-constant, fixed-based, or variable-based). The interpretation of OD. F depends on OD. MODE, and is described in detail in the following sections. The result of an extended address calculation is itself an address. A *long-operand* (LO) is the contents of memory at that address, except in the case of long-constant mode, where LO is the result of the evaluation of the constant (there being no intermediate addresses).

Indirection is specified by setting EW. I= 1. A full discussion of indirect addressing appears in Section 4.2.5. EW. S is used to facilitate array indexing and is described in Section 4.2.4.4. EW. P controls access to the previous address space and is discussed in Section 4.2.6.

4.2.4.1 Long Constant

Long constants are specified by setting OD.X=1 and OD.MODE=1. The address calculation then uses OD. F to indicate how the EW is to be interpreted (i.e., how the EW should be extended to a double-word or which register should be used for indexed long-constant mode). In this context OD. F is considered to be an *unsigned* constant in the range 0 . . 63 .

It should be noted that having OD.F=0 is a special case and is not long-constant addressing mode. It will be discussed further in the sections on fixed-based and variable-based addressing (Sections 4.2.4.2, 4.2.4.3). OD. F in the range 4 . . 31 results in a hard trap since these values are reserved for future use.

4.2.4.1.1 Immediate Long-Constant

If OD. F is in the range 1 . . 3 then the address calculation *is* in *immediate long-constant* mode. In this mode, LO=SIGNED(EW). If LO is to have precision smaller than a single-word (i.e., quarter-word or half-word), then the low-order bits of EW are used, and the bits not so used are ignored. If the precision is single-word, then all of EW is used. Thus for quarter-word, half-word, and single-word precisions, the values 1, 2, and 3 for OD.F all behave alike. If the precision is double-word, however, then OD.F specifies how the single-word EW is extended into the double-word format. OD.F=1 right-justifies EW into LO and sign-extends into the high-order word. OD. F=2 also right-justifies EW into LO but zero-extends into the high-order word. OD.F=3 left-justifies EW into LO and zeros out the low-order word.

The various types of long constant syntax appear below:

```
MOV RTB,#c106125103113⊃;RTB←arbitrary SW constant
```

The following sequence of instructions illustrates the several cases of sign extension. The two columns on the comment field indicate the value in RTA (DW) after the execution of each instruction.

```
                                ;high order SW of RTA:   low order SW of RTA:
MOV.D.D RTA,#c2⊃        ;              0                        2
ADD.D RTA,#c1↔0⊃        ;              1                        2
ADD.D RTA,#c!S↔-1⊃      ;              1                        1

ADD.D RTA,#c-1⊃         ;              2                        0
```

When an immediate long constant is used as a half-word or quarter-word then no check for overflow is made. Instructions may not require NEXT(immediate operand), as it is undefined and will result in a hard trap.

4.2.4.1.2 Indexed Long Constant

Indexed long constant mode is selected by having OD.MODE=1 and OD, F in the range 32..63. In this mode, the extended word is indexed by a register, selected by OD.F; LO=SIGNED(EW)+R[OD.F-32]. Overflow is not checked during the addition of EW and the register's contents. This sum is truncated to 36 bits. Quarter-word and half-word precisions use the low order bits of this result as the LO. Double-word precision uses this result, sign-extended into the high-order word, as the LO.

For example, the following instructions illustrate various uses of indexed constants. The comment field gives an alternative instruction with a similar effect. (The effects may not be identical because indexing does not detect arithmetic carry or overflow. This fact may sometimes be used to advantage.)

```
NOV RTA,#c200⊃(RTB)                    ;ADD RTA,#c200⊃,RTB
SKP.GEQ #c1⊃(RTA),#c-1⊃(RTB),FOO       ;SKP.GEQ #c2⊃(RTA),RTB,FOO
```

The following instruction sets RTA to (RTA+ l)x(RTA-1) (which is RTA^2-1) in a single instruction. There is no alternative implementation of this operation. It is assumed that RTA contains neither MAXNUM or MINNUM.

```
MULT RTA, #c1⊃ (RTA) , #c-1> (RTA)
```

4.2.4.1.3 Summary

OD.F	Extended-Word Interpretation
0	Special case of fixed- or variable-based addressing (SO=0)
1	EW right-justified, sign-extended into high-order single-word
2	EW right-justified, zero-extended into high-order single-word
3	EW left-justified, zeros to low-order single-word
4 . . 31	Reserved for future use (hard trap)
32 . . 64	Indexed constant: SIGNED(EW)+R[OD. F-321

Table 4-3
Long-Constant Mode

4.2.4.2 Fixed-based Addressing

Fixed-based addressing is used to access locations that are offset by up to 2^{30} quarter-words from the value specified by SO. A fixed-based address calculation uses EW. ADDR and EW. S as well as SO to compute the address of a LO.

Address calculation occurs in stages. SO is calculated first as described in Section 4.2.3 and then shifted left EW. S places. (For a full discussion of EW. S see Section 4.2.4.4.) The result is then added to the 30-bit base address EW.ADDR to produce the address of a LO, that is, $LO=M[EW.ADDR+SO*2^{EW.S}]$. If EW.I=1, indirect addressing is then used (see Section 4.2.5).

Fixed-based addressing is selected in two different ways. If OD. X=1, EW. V=O and OD. MODE ≠ 1 or 2 then the operand is computed as described above. If OD. X= 1, EW. V=0, OD. MODE= 1 and OD.F=0, then the operand is computed (as described above) with zero used in place of SO.

For example, let SP be the stack pointer, and let TABLE be the address of a table of QWs. The following instructions illustrate fixed-base addressing.

```
MOV RTA,<30>          ;alternative to MOV RTA,RTB (address in QWs)
MOV.H.H RTA,<22>      ; set high order HW of RTA equal to low order HW
```

The following sets RTA to the QW in TABLE indexed by the top stack element.

```
MOV.Q.Q RTA,<TABLE>(-1(SP))
```

The following two instructions set RTB to the address of a table of quarter-words, and then RTA to the second QW in the table.

```
MOVADR RTB,TABLE
MOV.Q.Q RTA,<1>(RTB)
```

4.2.4.3 Variable-based Addressing

Variable-based addressing uses EW. DISP and EW. REG to supply additional information for the operand evaluation. EW.DISP is interpreted as a signed offset from R[EW.REG]. The offset is in the range -2^{24} .. 224-l ,

Address calculation occurs in stages. The first stage involves adding R[EW.REG] to SIGNED(EW. DISP). This produces a base value which is used in subsequent calculations. The rest of the operand calculation proceeds as for fixed-base addressing, using this computed base

value in place of EW.ADDR. SO is calculated (see Section 4.23) and then is shifted left EW. S places. (For a full discussion of EW. S-field see Section 4.2.4.4). The resulting value is added to the base value to produce the address of the LO. Therefore, $LO=M[R[EW.REG]+SIGNED(EW.DISP)+SO*2^{EW.S}]$.

Variable-based addressing is selected in two different ways. If OD. X= 1, EW. V=l and OD. MODE ≠ 1 or 2 then the operand is computed as described above. If OD. X=1, EW. V= 1, OD. MODE= 1 and OD. F=0, then the operand is computed (as described above) with *zero* used in place of SO.

For example, let TABLE be the address of a table of QWs, and SP be the stack pointer. The following instructions illustrate various uses of variable-base addressing. The first two instructions set RTA to the RTA-th QW in the table.

```
MOVAOR RTB,TABLE
MOV.Q.Q RTA,<(RTB)>(RTA)
```

The following sets RTA to the RTA-th QW in the table, counting from the QW given by the top SW on the stack.

```
MOV.Q.Q RTA,<TABLE(RTA)>(-1(SP))
```

4.2.4.4 **Indexing** Into Data Structures: The S-field **(EW.S)**

EW. S is included in the fixed-based and the variable-based extended formats to facilitate indexing into data structures (e.g., arrays). It is often the case that many elements of a data structure are accessed sequentially. If one wanted to access a quarter-word structure in such a manner, one could use OD. X-l, OD. MODE=index register, OD. F=0, and EW.ADDR=base address of the structure. The contents of the index register would be an offset to the address in EW.ADDR. It (the contents of the index register) would also be the index of the element in the structure. To access the next element in the structure the contents of the index register would be incremented by one. It must be remembered, however, that addresses on the S-l are quarter-word addresses. If the elements of the structure are not of quarter-word precision then it would no longer be correct to add one to the index register to obtain the offset for the next element of the structure. Either the offset in the register would have to be shifted after incrementing, or an increment larger than one (1) would be needed (e.g., for single-words, four would be added). Using an additional shift instruction is undesirable because it would decrease code density, and also because it would cause a pipeline interlock which would slow the execution of the code. Using a larger increment would make it difficult to use the index register's contents as the index into the structure because the offset in the register would be some multiple of the actual index. The solution chosen by the designers of the S- 1 is to use a field EW. S to specify how many bits to shift the SO to make memory appear to be

the desired precision. EW. S equal to 0, 1, 2, or 3 causes the "apparent memory precision" to be quarter-, half-, single-, or double-word, respectively. If one wanted to access a single-word data structure (using fixed-based addressing), the method outlined above would work if one set EW. S to 2. The contents of the index register would then specify both the "single-word offset" (i.e., the quarter-word offset divided by 4) to the base of the structure *and* the index of the element within the structure. The address calculation would then shift this "single-word offset" left two bits, converting it into a quarter-word offset. The resulting address would be the actual location of the data element. To increment the index, the register contents would be incremented by one. The shift by EW. S takes care of adjusting the precision, and since it is part of the operand calculation, no pipeline-interlock occurs.

For example, let SP be the stack pointer and let TABLE be the address of a table of SWs. The following illustrates how the shift field facilitates indexing into this table. RTA is set to the SW element of the table one SW beyond the SW indexed by the top SW in the stack. Informally, RTA←table(stack(SP-1)+ 1). The shift field EW.S is specified by the number following the up-arrow "↑".

```
MOVADR RTB,TABLE
MOV RTA,<4(RTB)>(-1(SP))↑2
```

4.2.5 Indirect Addressing

Indirect addressing may be used during extended addressing by setting EW. I= 1. It is used for accessing memory through pointers that are stored as single-words in memory. With EW. I= 1, the LO that is calculated in previous addressing stages is now interpreted as an indirect address pointer (IAP) (see Section 3.4). The fields of the IAP are then used to compute the address of the actual operand. This operand is termed the indirect long *operand* (ILO).

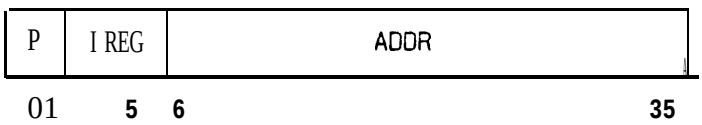

Figure 4- 10
Indirect Address Pointer

There are two different types of indirection which can be selected. IAP.IREG determines which one is used. If IAP.IREG=O then IAP.ADDR is used as the address of the ILO. Thus, ILO=M[IAP. ADDR]. This is termed *simple indirection.* If IAP. IREG≠0 then *indexed indirection* is used. In this case, R[IAP.IREG] is added to IAP. ADDR to produce the address of the ILO so that ILO=M[R[IAP.IREG]+IAP.ADDR]. Note that R[0] can not be used in the above computation, since IAP.IREG=0 specifies simple indirection.

Like all addressing operands, the IAP operand evaluation logically occurs before the instruction execution and before the PC is updated, Since it has no side effects, it is restartable. The IAP calculation is done modulo 2^{30} and does not set carry or overflow flags. See Section 4.2.7 for more details on addressing restrictions and exceptions. The interpretation of the P-bit is discussed in section 4.2.6. .

For example, assume resister P contains the address of the first word of any node in a circular, doubly-linked list of nodes consisting of three single words: a "next link", a "last link" and a "data pointer" which points to a SW quantity. The following illustrates use of indirection.

```
MOV P,(P)                    ;advance P to point at the "next" node
MOV P,1(P)                   ;backup P to point at the "last" node
MOV P,@(P)+                  ;advance P to the "next" of "next" node
MOV P,@1(P)                  ;this does the same thing a different way
EXCH @8+(0(P)),@8+(1(P))     ;swap data-pointer(last) with data-pointer(next)
EXCH @@8+(0(P)),@@8+(1(P))   ;swap data(last) with data(next)
```

4.2.5.1 Summary

IAP. IREC	Mode Name	ILO
0	Simple Indirection	M[IAP.ADDR]
1 . . 31	Indexed Indirection	M[R[IAP.IREG]+IAP.ADDR]

Table 4-4
Indirect Address Pointer (IAP)

4.2.6 Address Space Switching: The P-bit

Bit zero of fixed-based EWs, variable-based EWs, and IAPs is interpreted as a ***previous context bit*** (P-bit). The P-bit specifies the address space that will be used in the computation of an extended operand. The interpretation of the P-bit is always done as the last step of a given phase of address calculation (e.g., it is done just before LO is fetched, and again just before IL0 is fetched in an indirect address calculation). The PREV_MODE, CRNT-MODE, PREV_FILE, CRNT_FILE, and USE_SHADOW_PREV fields of PROCSTATUS determine the effects of the P-bit. (See Section 2.5.1 for a description of PROCSTATUS.)

The purpose of the P-bit is to facilitate communication between a program and the executive. If a (user or executive) program traps, then the P-bit allows the executive routine that handles the trap to access the memory space of the program that trapped. CRNT-MODE (PREV_MODE) indicates whether the current (previous) context is in user or executive mode. CRNT-MODE-O (PREV_MODE=0) means that the current (previous) context is in user mode. CRNT_MODE=1 (PREV_MODE=1) means that the current (previous) context is in executive mode.

P=0 means that the address space being referenced is the same as that selected by CRNT-MODE. It is used by both the executive and the user each to access its own address space. The executive may access operands in the previous address space by using a P-bit equal to 1. If a user (i.e., a program with CRNT-MODE-O) attempts to access the previous address space by using a P-bit equal to 1, a hard trap will occur.

Only one change of address space is allowed in the evaluation of a single operand since this is all that is needed to allow the executive to access the trapping program's address space. Therefore, if a P-bit equal to 1 has already been encountered in an address calculation, encountering another one will cause a hard trap.

Since the interpretation of the P-bit is always done as the last step of the address calculation, if an IAP is fetched from a given address space (either current or previous), then the IREG and ADDR fields are also interpreted as being in that same address space. After all these other fields have been evaluated, the P-bit of the IAP is then interpreted. If IAP. P=0, then the IL0 is fetched from the same address space as the IAP. If IAP. P= 1 and the IAP is in the current address space, then the IL0 is fetched from the previous address space. All other cases will hard-trap.

The first instruction below uses the P-bit in the extended word to access the RTB-th single-word in TABLE in the previous address space. The second uses an IAP to achieve the same effect. Note that the @ symbol causes FASM to set the P-bit in the IAP constant, but specifies indirection in the EW.

```
MOV RTA,c!P TABLE (RTB)↵
MOV RTA,c@[@ TABLE (RTB) I↵
```

4.2.7 Addressing **Restrictions and** Exceptions

Without exception, instructions that require NEXT(OP) or ADDRESS(OP) where OP is either a short-constant or a long-constant will hard-trap.

If an instruction requires two EWs, the first is used to calculate OP2, and the second to calculate OP 1.

All instructions which move addresses (e.g., MOVADR) perform the address interpretation procedure to the point just before the virtual-to-physical translation, and store the resulting 36-bit number (possibly with the P-bit=1) in the destination. See Section 2.3 on virtual-to-physical address translation.

A hard trap will occur if an instruction has a jump destination which is in the previous context. Jumps to registers are undefined.

Note that the PC is a 30-bit positive number (Le., is zero-filled to the left in indexing). References to register R[3] are interpreted as references to the PC under certain conditions. PC is used instead of R[3] whenever R[3] is specified as an index register within an address calculation. This includes indexing off of R[3] in indirect address pointers (see Section 4.2.5). All other references to R[3] refer to the contents of general-purpose register number 3.

For an instruction to be executable, the two words following the first word of the instruction must be valid instruction words (Le., they must exist in the address space and be on a page with access mode INSTRUCTION=1). This applies even when those two words are not part of the instruction and even when they cannot possibly be executed as part of any instruction. This is an effect of pipelining.

There are two cases where crossing the memory/register boundary may cause hard traps. Instructions that begin within two single-words of the boundary (inclusive) will cause a hard trap when executed. Instructions that have operands or sequences of operands (e.g., NEXT(operand)) that are addressed in register-direct mode (see Section 4.2.3.1) and that cross the memory/register boundary will cause a hard trap. Operands that are accessed as the first 128 quarter-words of *memory* will never cause a memory/register boundary hard trap (but may cause traps such as alignment error, etc.).

4.2.8 Addressing Summary

Short-Operand	SO	OD.MODE	OD.F
Register-direct	R[OD.F]	0	0..31
Short-constant	SIGNED(OD.F)	1	-32..31
Short-indexed	M[PC+4*SIGNED(OD.F)]	3	-32.31
Short-indexed	M[R[OD.MODE]+4*SIGNED(OD.F)]	4..31	-32..31

OD.X = 0

Table 4-5
Short-Operand Addressing Summary

Long-Constant	LO	OD.F	EW extension to double-word
Immediate	SIGNED(EW)	1	right-justified, sign-extended
Immediate	SIGNED(EW)	2	right-justified, zero-extended
Immediate	SIGNED(EW)	3	left-justified, low order zero
Indexed	SIGNED(EW)+R[OD. F-321	32..63	

OD.X = 1, OD.MODE = 1

Table 4-6
Long-Constant Addressing Summary

LO	OD. MODE	OD.
M[EW.ADDR+SO*2^EW.S]	≠ 1,2	OD.F
M[EW.ADDR]	. 1	0

OD.X = 1, EW.V = 0

Table 4-7
Fixed-Based Addressing Summary

LO	OD. MODE	OD.F
M[R[EW.REG]+SIGNED(EW.DISP)+SO*2^EW.S]	≠ 1,2	OD.F
M[R[EW.REG]+SIGNED(EW.DISP)]	1	0

OD.X = 1, EW.V = 1

Table 4-8
Variable-Based Addressing Summary

ILO	IAP.IREG
M[IAP.ADDR]	0
M[R[IAP.IREG]+IAP.ADDR]	1.. 31

EW.I = 1

Table 4-9
Indirect Addressing Summary

4.2.9 FASM **Addressing Summary**

In the following tables, lower case symbols denote FASM expressions (these tables correspond one-to-one with the previous section).

Short-Operand	SO	FASM
Register-direct	R[r]	%r
Short-constant	sc	#sc
Short-indexed	M[PC+4*sc]	sc(3)
Short-indexed	M[R[r]+4*sc]	sc(r)

r = register 0 . . 31 , sc = short constant -32 . . 31

Table 4-10
FASM Short-Operand Addressing Summary

Lone-Constant	LO (DW)	FASM
Immediate	SIGN_EXTEND(lc) ↔ lc	#⊂!S ↔ lc⊃
Immediate	0 ↔ lc	#⊂lc⊃
Immediate	lc ↔ 0	#⊂lc ↔ 0⊃
Indexed	lc+R[r]	#⊂lc⊃(r)

lc = long constant (SW), r = register

Table 4-l 1
FASM Long-Constant Addressing Summary

LO	FASM
$M[x+so*2^{sh}]$	c x ↑(so)↑sh
M[x]	c x ↑

x = address, so = short operand, sh = shift 0 . . 3
(in x: @ = indirect, !P = previous context)

Table 4-12
FASM Fixed-Based Addressing Summary

LO	FASM
$M[R[r]+x+so*2^{sh}]$	c x(r)↑(so)↑sh
M[R[r]+x]	c x(r)↑

x = offset, r = register, so = short operand, sh = shift 0 . . 3
(in x: @ = indirect, !P = previous context)

Table 4-13
FASM Variable-Based Addressing Summary

IL0	FASM
M[x]	@x
M[R[r]+x]	@x(r)

x = address, r = register

Table 4-14
FASM Indirect Addressing Summary

5 Instruction Descriptions

The instruction set of the S-l contains many powerful instructions for manipulating various data types. The instructions are designed to make the implementation of high-level languages easier and more efficient in terms of both storage and speed. The formats for the instructions are described in Section 4.1.

Ail S-l instructions are written as an opcode name followed by zero or more modifiers. These modifiers are separated from the opcode field and from each other by the "." character (i.e., *opcode*{ . modifier)). In the instruction descriptions **that** follow, ail the possible values of **a** modifier-field are listed within curly brackets **at the** position which they should occur in the instruction. One modifier from each set in the curly brackets must be used. (An exception to this rule is that if precision modifiers are omitted, then single-word precision is assumed.) The order of the modifier-fields is important (e.g., MOV.Q.S is *not* the same as MOV.S.Q).

Essentially ail three-operand instructions that are asymmetric with respect to S1 and S2 in their operation are- provided in reverse form (i.e., where an instruction uses S1 *operation S2,* the reverse instruction uses S2 *operation* Sl). This is indicated **by** appending the letter "V" **to** the end of the opcode name (e.g., SUB and SUBV, or SHF and SHFV). Instructions for commutative operators such **as** ADD are symmetric in S1 and S2, and so need no reverse forms.

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

5.1 Instruction-Execution Sequence

The execution of an instruction can be logically divided into a number of stages which make up the *instruction-execution sequence.* These **stages** are described in order in the following paragraphs.

The first stage is concerned with processing *interrupts.* (See Section 5.16 for a description of the interrupt architecture.) If an interrupt is pending at this time, the interrupt is serviced by jumping to the interrupt handler specified in the appropriate interrupt vector. Return from the interrupt handier restarts the instruction-execution sequence, so that if further interrupts are pending, they will also be serviced. If no interrupts are pending, control passes immediately to the next stage.

The second stage of the instruction-execution sequence processes *trace-traps* trace-trap. TRACE-PEND is sampled and reset. If a trace-trap is pending (TRACE_PEND=1), then a trap occurs and the trace-trap handier is executed. Upon return, the trapping instruction is restarted from the beginning. Interrupts are processed again. The TRACE-PEND flag is sampled again, but unless the trace-trap handler changed the saved PROCSTATUS, TRACE-PEND is necessarily zero, since was reset before the trace-trap began. If a trace-trap is not pending (TRACE_PEND=0), control passes to the next stage.

Before-instruction exceptions are handled in the third stage. These include exceptions such as page-faults and illegal memory-access traps that can be detected before instruction execution has begun. If any before-instruction exception is detected, the exception is handled, and when the exception handier reutrns, control is passed back to the beginning of the first stage. Interrupts are processed again. The TRACE-PEND flag is sampled again, but unless the exception handler changed the saved PROCSTATUS, TRACE-PEND is (again) necessarily zero. Thus, repeated before-instruction exceptions can occur without causing superfluous trace-traps.

The fourth stage of instruction execution simply saves the value of TRACE-ENB for use after the part of instruction execution which may change PROCSTATUS. We call this saved value $\text{TRACE\_ENB}_{\text{OLD}}$.

During the fifth stage of instruction execution, the instruction body is executed, possibly affecting the user state.

Some lengthy instructions are interruptable. Interrupts occurring within interruptable instructions save INSTRUCTION-STATE (an otherwise inaccessible hardware register) on the stack. The saved INSTRUCTION-STATE allows the interrupted instruction to restart at the proper point when the interrupt handier returns. A zero value for INSTRUCTION-STATE means that the instruction body has not begun execution, i.e., that the instruction can be restarted from the beginning.

In the sixth stage of instruction execution, TRACE-PEND is set to TRACE-PEND v $\text{TRACE\_ENB}_{\text{OLD}}$. Thus, if tracing was enabled when this instruction commenced (or if this instruction itself sets TRACE-PEND), a trace-trap will occur after this instruction completes (i.e., at the beginning of the next instruction). Hence, the trace-trap handier receives a trap after the last instruction in a sequence of instructions to be traced, as well as before the first instruction in the sequence.

After-instruction exceptions such as integer overflow are handled in the seventh and last stage of instruction execution. If the handler of an after-instruction exception restarts the instruction (which will not normally be the case), another trace-trap may occur immediately (depending upon the value of TRACE-PEND). A second trace-trap is appropriate in this case, since 'the instruction is actually being executed twice.

The formal description of the above instruction-execution sequence for a single S-l processor (S-1_Uniprocessor) is shown below.

```
define S-1_Uniprocessor≡
    do forever
        program-counter ← pc-nxt-instr next .
        Check-Interrupts next
        if Trace-Trap--Pending
          then Trace-Trap
          else Fetch-Instruction-Word next
               Decode-Opcode
        fi next
        Trace-Trap-Pending ← Trace-Trap-Pending v Trace-Trap-Enable
    reverof od;
```

5.2 Integer

5.2.1 Signed Integer

Signed integer instructions operate upon the signed integer data type (see Section 3.2). The instructions perform addition, subtraction, multiplication, integer division, remainder, and modulus functions. Negation, absolute value, min, and max are also provided. Non-commutative operations such as subtraction are provided in both normal and reverse forms . These reverse instructions are indicated by a "V" as the last character of the opcode string. (e.g., SUB becomes SUBV). Instructions that allow extended-precision operations (e.g., multiplying two single-word integers and producing a double-precision result) have an "L" as either the last or penultimate character of the opcode.

Two different remainder functions are provided: *rem* and *mod*. The result of mod has the same sign as the divisor of the operation (or is zero), whereas the result of *rem* has the same sign as the dividend (or is zero). In both cases, however,

$$\text{DIVIDEND} = (\text{DIVISOR} * \text{QUOTIENT}) + \text{REMAINDER}$$
and
$$\text{ABS(REMAINDER)} < \text{DIVISOR}$$

For example, -5 *mod* 3 = 1 ($QUOTIENT_{mod}$=-2) while -5 *rem* 3 = -2 ($QUOTIENT_{rem}$=-1).

Integer division (QUO, DIV, etc.) produces QUOTIENT,.,,, not $QUOTIENT_{mod}$. For example, the result of (-1)/2 is zero, not -1. The SHFA.RT instruction can be used to produce $QUOTIENT_{mod}$ in the case that the divisor is a power of two. By contrast, the QUO2 series of instructions produces QUOTIENT,.,,, like all QUO instructions. This may all be summarized by noting that QUO and DIV instructions always round the quotient towards zero, while SHFA.RT rounds towards negative infinity. (See Section 5.7 for shift instructions.)

Section 5.2.3 describes the possible side effects of signed-integer instructions (CARRY, INT_OVFL, and INT_Z_DIV).

ADD

Instruction: ADD . {Q,H,S,D}
Class: TOP

Integer add

Purpose: DEST←S1+S2. The integer sum of S1 and S2 is stored in DEST.

Side Effects: CARRY, INT_OVFL

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

Formal Description:

```
define ADD.p:qhsd ≡   TOP [p;p;p] Add(S1,s2)→ sum, c, ov next
                                   Int_Overflow? nextS-1
                                   (dest ⇐ sum also Carry ←c);
```

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,#c333>,#c777>          ;RTA=332
```

ADDC

Instruction: **ADDC . {Q,H,S,D}**
Class: TOP Integer add with carry

Purpose: DEST←S1+S2+CARRY

Side Effects: CARRY, INT_OVFL

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

Formal Description:

```
define ADDC.p:qhsd≡   TOP [p;p;p] Add_With_Carry(S1, s2, Carry) → sum, c, ov next
                                  Int_Overflow? next
                                  (dest← sum also Carry ← c);
```

Carry is set after the execution of the first instruction, and cleared after the second. 1

```
ADD.Q RTA,#c666ↄ,#c777ↄ          ;RTA=665
ADDC.Q RTA,RTA,#1                ;RTA=667
```

The following adds two long 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 NEXT(X).

```
ADD.D X+10,Y+10
ADDC.D X,Y
```

Similarly, suppose that NUM1 and NUM2 are two blocks of single-words, 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 (Ntl)-word block NUM3 in this manner:

```
        MOV RTB,#cN-1ↄ                      ;RTB counts words
        ADD RTA,cNUM1ↄ(RTB),cNUM2ↄ(RTB)     ; add low-order words
        MOV cNUM3+1ↄ(RTB),RTA               ; store low-order result
LOOP:   ADDC RTA,cNUM1-1ↄ(RTB),cNUM2-1ↄ(RTB) ;add next words plus carry
        MOV cNUM3ↄ(RTB), RTA                ;store next word
        DJMPZ.GTR RTB, LOOP                 ;DJMPZ doesn't alter carry!
        CMPSF.LSS RTA,cNUM1ↄ                ;produce sign-extension of
        CMPSF.LSS,RTB,cNUM2ↄ                ;NUM1 and NUM2
        ADDC cNUM3ↄ,RTA,RTB                 ;produce high-order result
```

SUB

Instruction: **SUB . {Q,H,S,D}**
Class: TOP

Integer subtract

Purpose: DEST←S1-S2

Side Effects: CARRY, INT_OVFL

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

Formal Description:

```
define SUB. p:qhsd≡      TOP [p;p;p]   Subtract(S1, s2) → dif, c, ov next
                                       Int_Overflow? next
                                       (dest ⇐ dif also Carry ← c);
```

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

```
SUB RTA,#-1,#1   ;RTA=-2
```

SUBV

Instruction: **SUBV . {Q,H,S,D}**
Class: TOP Integer **subtract** reverse

Purpose: DEST←S2-S 1

Side Effects: CARRY, INT_OVFL

Precision: S 1, S2, **and** DEST all **have the precision** specified by **the** modifier.

Formal Description:

```
define SUBV.p:qhsd =    TOP[p;p;p] Subtract(s2, S1) → dif, c, ov next
                                                    Int_Overflow? next
                                                    (dest ⇐ dif also Carry ← c);
```

The long constant below is a SW minus one in signed interpretation.

```
        SUBV RTA,#c777777777777⊃,#1      ;RTA=+2
```

SUBC

Instruction: SUBC . {Q,H,S,D}
Class: TOP Integer **subtract** with carry

Purpose: DEST←S1-S2-1+CARRY

Side Effects: CARRY, INT_OVFL

Precision: S 1,S2, **and** DEST all have the precision specified by **the** modifier.

Formal Description:

```
define SUBC.p:qhsd = TOP [p;p;p] Subtract-With-Carry (S1,s2, Carry) → dif, c, ov next
                     Int_Overflow? next
                     (dest ⇐ dif also Carry ← c);
```

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.0 X,Y
SUBC.D X+10, Y+10
```

SUBCV

Instruction: **SUBCV . {Q,H,S,D}**
Class: TOP Integer subtract with carry reverse

Purpose: DEST←S2-S1-1+CARRY

Side Effects: CARRY, INT_OVFL

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

Formal Description:

```
define SUBCV. p:qhsd₈  TOP[p;p;p]  Subtract-With-Carry (s2,S1, Carry) → dif, c, ov next
                                   Int_Overflow? next
                                   (dest⇐ dif also Carry ←c);
```

The following illustrates SUBCV,

```
SUB RTA,#2,#1          ;RTA=+1, carry clear
SUBCV RTA, #2, RTA     ;RTA=-2, carry set
```

MULT

Instruction: **MULT . {Q,H,S,D}**
Class: TOP Integer multiply

Purpose: DEST←LOW_ORDER(S1*S2)

Side Effects: INT-OVFL

Precision: S 1, 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 larger than can fit in nine bits: 1221 octal.

```
MULT.Q RTA,#c333o,#3     ;RTA=221
```

MULTL

Instruction: **MULTL . {Q,H,S}**
Class: TOP Integer multiply long

Purpose: DEST←S1*S2

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

The following instruction does *not* set INT,OVFL since the result fits in a halfword.

```
MULTL.Q RTA,#c333ↄ,#3    ;RTA=001221
```

QUO

Instruction: **QUO . {Q,H,S,D}**
Class: TOP Integer q uotien t

Purpose: DEST←S1/S2. QUO rounds its result towards zero.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following illustrates a simple quotient calculation.

```
QUO.Q RTA,#c345>,#3      ;RTA=114
```

QUOV

Instruction: **QUOV . {Q,H,S,D}**
Class: TOP **Integer quotient** reverse

Purpose: DEST←S2/S1 QUOV rounds its result towards zero.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: S 1, S2, **and** DEST **all have** the precision specified **by the** modifier.

The following illustrates a quotient calculation. 1

```
QUOV.Q RTA, #c114ↄ,#c345ↄ          ;RTA=3
```

QUOL

Instruction: **QUOL . {Q,H,S}**
Class: TOP Integer quotient **long**

Purpose: DEST←S1/S2. QUOL rounds its result towards zero.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: S 1, NEXT(S1), S2, DEST **have the same** precision as **the** modifier. S 1 **has** a precision **twice that** of **the** modifier.

The following illustrates taking a quotient with a long dividend.

```
QUOL.Q RTA, #c1221ↄ,#3    ;RTA=333
```

QUOLV

Instruction: QUOLV.{Q,H,S}
Class: TOP **Integer quotient** long reverse

Purpose: DEST←S2/S1. QUOLV rounds its result towards zero.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: S 1 and DEST have the same precision as the modifier. S2 has a precision *twice* that of the modifier.

The following illustrates taking a quotient with a long dividend.

```
QUOLV.Q RTA,#c333ↄ,#c1221ↄ        ;RTA=3
```

QUO2

Instruction: QUO2 . {Q,H,S,D}
Class: TOP Integer quotient by power of 2

Purpose: DEST←S1/2^{S2}. QUO2 rounds its result towards zero. The SHFA.RT instruction may be used to divide by a power of two, rounding towards negative infinity. S2 may be negative, in which case a multiplication by a positive power of two is performed.

Side Effects: INT-OVFL (INT-OVFL is not set during the 2^{S2} portion of the operation. This exponentiation is done with unlimited precision.)

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following divides -3 by +2, giving a different result than SHF.RT with the same operands.

```
QUO2 RTA,#-3,#1          ;RTA=-1
```

QUO2V

Instruction: **QUO2V . {Q,H,S,D}**
Class: TOP Integer quotient by power of 2 reverse

Purpose: DEST←S2/2^{S1}. QUO2V rounds its result towards *zero.* The SHFAV.RT instruction may be used to divide by a power of two, rounding towards negative infinity. S1 may be negative, in which case a multiplication by a positive power of two is performed.

Side Effects: INT-OVFL (INT_OVFL is not set during the 2^{S1} portion of the operation. This exponentiation is done with unlimited precision.)

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The second instruction illustrates the use of negative shifts.

```
QUO2V RTA,#1,#-2        ;RTA=-1
QUO2V RTA,RTA,#1        ;RTA=2
```

QUO2L

Instruction: **QUO2L. {Q,H,S}**
Class: TOP Integer quotient by power of 2 long

Purpose: DEST←S1/2^{S2}. QUO2L rounds its result towards zero. S2 may be negative, in which case a multiplication by a positive power of two is performed.

Side Effects: INT-OVFL (INT_OVFL is not set during the 2^{S2} portion of the operation. This exponentiation is done with unlimited precision.)

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

The following divides the long operand by 16 (decimal). 1

```
QU02L.Q RTA,#c1221>,#4;RTA=51
```

QUO2LV

Instruction: **QUO2LV . {Q,H,S}**
Class: TOP Integer quotient by power of 2 long reverse

Purpose: DEST←S2/2^{S1}. QUO2LV rounds its result towards zero. S1 may be negative, in which case a multiplication by a positive power of two is performed.

Side Effects: INT-OVFL (INT-OVFL is **not** set during **the** 2^{S1} portion of the operation. This exponentiation is done with unlimited precision.)

Precision: S I and DEST **have** the same precision as **the** modifier. S2 has **a** precision **twice that** of the modifier.

In the first instruction RTA is to be interpreted as a HW destination. In the second instruction RTA is to be interpreted as a QW destination, a QW shift argument, and a HW operand, respectively. Note that the second instruction leaves the contents of RTA unchanged (independent of its interpretation).

```
QUO2LV.H RTA,#-11,#11    ;RTA=11000 (HW)
QUO2LV.Q RTA, RTA,RTA    ;RTA=11 (QW)
```

REM

Instruction: **REM . {Q,H,S,D}**
Class: TOP Integer remainder

Purpose: DEST←S1 *rem* S2. The result is the remainder produced by a division that rounds towards zero (as in the QUO instruction). The result (DEST) has the same sign as the dividend (S 1), or is zero. Note that the MOD function provided in many high-level languages such as PASCAL actually performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following illustrate the results of various combinations of signs.

```
REM.Q RTA,#5,#3        ;RTA=2
REM.Q RTA,#5,#-3       ;RTA=2
REM.Q RTA,#-5,#3       ;RTA=-2
REM.Q RTA,#-5,#-3      ;RTA=-2
```

REMV

Instruction: **REMV . {Q,H,S,D}**
Class: TOP Integer remainder reverse

Purpose: DEST←S2 *rem* S1. The result is the remainder produced by a division that rounds towards zero (as in the QUOV instruction). The result (DEST) has the same sign as the dividend (S2), or is zero. Note that the MOD function provided in many high-level languages such as PASCAL actually performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following illustrate the results of various combinations of signs.

```
REMV.Q RTA,#3,#5        ;RTA=2
REMV. Q RTA, H-3, #5    ;RTA=2
REMV.Q RTA,#3,#-5       ;RTA=-2
REMV.Q RTA,#-3,#-5      ;RTA=-2
```

REML

Instruction: **REML . {Q,H,S}**
Class: TOP Integer remainder **long**

Purpose: DEST←S1 *rem* S2. The result is the remainder produced by a division that rounds towards zero **(as in the** QUOL **instruction).** The result (DEST) has **the same** sign as the dividend (Sl), or **is zero.** Note that **the** MOD function provided in many high-level languages **such as** PASCAL actually performs the REM operation, **not the** MOD operation.

Side Effects: INT_Z_DIV

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

The following illustrates the remainder using a long dividend.

```
REML,Q RTA,#c12345ↄ,#c300ↄ          ;RTA=245
```

REMLV

Instruction: **REMLV . {Q,H,S}**
Class: TOP Integer remainder **long** reverse

Purpose: DEST←S2 *rem* S1. The result is the remainder produced **by** a division **that** rounds towards *zero* (as in **the** QUOLV instruction). The result (DEST) has **the same** sign as the dividend (S2), or is zero. Note that **the** MOD function provided in **many** high-level **languages such** as PASCAL **actually** performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

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

The following illustrates a remainder **using a** long dividend.

```
REMLV. Q RTA, #c300ↄ,#c12345ↄ      ;RTA=245
```

MOD

Instruction: **MOD . {Q,H,S,D}**
Class: TOP

Integer modulus

Purpose: DEST←S1 *mod* S2. The result is the remainder produced by a division that rounds towards negative infinity, The result (DEST) has the same sign as the divisor ⟨S2⟩, or is zero. Hence when the divisor is positive the result is the number-theoretic reduction of S1 in the modulus S2. Note that the MOD function provided in many high-level languages such as PASCAL actually performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following illustrates the result of various combinations of signs.

```
MOD.Q RTA,#5,#3          ;RTA=2
MOD.Q RTA,#5,#-3         ;RTA=-1
MOD.Q RTA,#-5,#3         ;RTA=1
MOD.Q RTA,#-5,#-3        ; RTA--2
```

MODV

Instruction: **MODV . {Q,H,S,D}**
Class: T O P Integer **modulus reverse**

Purpose: DESTcS2 *mod* S1. The result is the remainder produced by a division that rounds towards negative infinity. The result (DEST) has **the same sign as the** divisor (Sl), or is zero. Hence **when the** divisor is positive the result is **the number-theoretic** reduction of **S2 in the modulus** S1. Note that **the** MOD function provided **in many** high-level **languages** such as PASCAL actually performs **the** REM operation, not the MOD operation.

Side Effects: INT Z DIV

Precision: S 1, S2, and DEST **all have the** precision specified by the modifier.

The following illustrates the result of various **combinations** of signs.

```
MODV.Q RTA,#3,#5        ;RTA=2
MODV.Q RTA, #-3, #5     ; RTA=-1
MODV.Q RTA,#3,#-5       ;RTA=1
MODV.Q RTA,#-3,#-5      ;RTA=-2
```

MODL

Instruction: **MODL . {Q,H,S}**
Class: TOP Integer modulus long

Purpose: DEST←S1 *mod* S2. The result is the remainder produced by a division that rounds towards negative infinity. The result (DEST) has the same sign as the divisor (S2), or is zero. Hence when the divisor is positive the result is the number-theoretic reduction of S1 in the modulus S2. Note that the MOD function provided in many high-level languages such as PASCAL actually performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

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

The following illustrates the modulo operation using a tong dividend.

```
MODL.Q RTA,#c12345ↄ,#c300ↄ      ;RTA=245
```

MODLV

Instruction: **MODLV . {Q,H,S}**
Class: TOP Integer modulus long reverse

Purpose: DEST←S2 *mod* S1. The result is the remainder produced by a division that rounds towards negative infinity. The result (DEST) has the same sign as the divisor (S 1), or is zero. Hence when the divisor is positive the result is the number-theoretic reduction of S2 in the modulus Sl. Note that the MOD function provided in many high-level languages such as PASCAL actually performs the REM operation, not the MOD operation.

Side Effects: INT_Z_DIV

Precision: Sl and DEST have the same precision as the modifier. S2 has a precision *twice* that of the modifier.

The following illustrates the modulo operation using a long dividend.

```
MODLV.Q RTA,#c300ↄ,#c12345ↄ      ;RTA=245
```

DIV

Instruction: **DIV . {Q,H,S,D}**
Class: T O P Integer divide

Purpose: DEST←S1/S2; NEXT(DEST)←S1*rem* S2. DIV is like doing **both** a QUO instruction and a REM instruction.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: S 1, S2, DEST, and NEXT(DEST) all have the **same** precision as the modifier.

The following produces a quotient-remainder result.

```
DIV.Q RTA,#<345>,#3      ;RTA=114001 (two QWs)
```

DIVV

Instruction: **DIVV . {Q,H,S,D}**
Class: TOP Integer divide reverse

Purpose: DEST←S2/S 1; NEXT(DEST)←S2 *rem* Sl. DIVV is like doing **both** a QUOV instruction and a REMV instruction.

Side Effects: INT_OVFL, INT_Z_DIV

Precision: Sl, S2, DEST, **and** NEXT(DEST) ail **have the same** precision as the modifier.

The following produces a quotient-remainder result. 1

```
DIVV.Q RTA,#3,#c345>     ;RTA=114001 (two QWs)
```

DIVL

Instruction: DIVL . {Q,H,S}
Class: TOP Integer divide long

Purpose: DEST←S1/S2; NEXT(DEST)←S1 *rem* S2. DIVL is like doing both a QUOL instruction and a REML instruction.

Side Effects: INT_OVFL, INTZDIV

Precision: S2, DEST, NEXT(DEST) have the same precision as the modifier. Sl has a precision *twice* that of the modifier.

The following produces a quo'tient-remainder for a long operand.

```
DIVL.Q RTA,#c12345ↄ,#c300ↄ      ;RTA=33245(two QWs)
```

DIVLV

Instruction: **DIVLV . {Q,H,S}**
Class: TOP Integer divide long reverse

Purpose: **DEST←S2/S** 1; **NEXT(DEST)←S2** *rem S* 1. DIVLV is like doing both a QUOLV instruction and a REMLV instruction.

Side Effects: **INT_OVFL,** INTZDIV

Precision: S I, DEST, NEXT(DEST) have the same precision as the modifier. S2 has a precision *twice* that of the modifier.

The following produces a quotient-remainder for a long operand.

```
DIVLV.Q RTA,#c300ↄ,#c12345ↄ      ;RTA=33245 (two QWs)
```

INC

Instruction: **INC . {Q,H,S,D}**
Class: XOP

Integer increment

Purpose: OP 1←OP2+ 1

Side Effects: CARRY, INT_OVFL

Precision: OP 1 and OP2 have the same precision as the modifier.

Formal Description:

```
define INC.p:qhsd ≡     XOP[p;p]Add(op2,1)→ sum, c, ov next
                                Int_Overflow? next
                                (op1 ⇐ sum also Carry←c);
```

The following adds one to RTA.

```
INC RTA,RTA      ;RTA←RTA+1
```

FASM allows this instruction to be abbreviated simply to:

```
INC RTA          ;RTA is both source and destination
```

DEC

Instruction: **DEC . {Q,H,S,D}**
Class: XOP

Integer decrement

Purpose: OP1←OP2-1

Side Effects: CARRY, INT_OVFL

Precision: OP 1 and OP2 have the same precision as the modifier.

Formal Description:

```
define DEC. p:qhsd≡   XOP [p;p]  Subtract (op2,1)→ dif , c, ov next
                                 Int_Overflow? next
                                 (op1← dif also Carry ←c);
```

The following subtracts one from RTA.

```
DEC RTA          ;RTA←RTA-1
```

This instruction subtracts one from BAR and puts the result in FOO.

```
DEC FOO,BAR      ;FOO←BAR-1
```

TRANS

Instruction: **TRANS . {Q,H,S,D} . {Q,H,S,D}**
Class: XOP Integer transfer

, Purpose: OP1←SIGN_EXTEND(OP2). Take the integer specified by OP2 and sign-extend it to make it an integer of the precision of the first modifier. Store the result in OP1. More precisely, OP2 is sign-extended if OP1 is longer than OP2. It is unchanged if OP 1 and OP2 are the same length (in which case TRANS behaves just like MOV). If OP1 is shorter than OP2, then a "sign-compressed" copy of OP2 is stored in OP1, provided the correct numerical value of OP2 can be expressed in the precision of OP1; if it cannot, INT-OVFL is signalled.

Side Effects: 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

Instruction: NEG.{Q,H,S,D}

Class: XOP Integer negate

Purpose: OP 1←*two's-complement*(OP2)

Side Effects: CARRY, INT_OVFL

Precision: OP 1 and OP2 have the same precision as the modifier.

Formal Description:

```
define NEG. p:qhsd =    XOP[p;pl  Subtract (0,op2) → dif , c, ov next
                                  Int_Overflow? next
                                  (op2 ⇐ dif also Carry ← c);
```

The following negates the value in RTA.

```
NEG RTA              ;RTA←-RTA
```

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

```
NEG RTA, HUNOZ           ;RTA←-HUNOZ
ANOCT RTA, HUNOZ         ;RTA←one's-complement(RTA)∧HUNOZ
JMPZ.EQL 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.

ABS

Instruction: **ABS . {Q,H,S,D}**
Class: XOP **Integer absolute value**

Purpose: OP 1←*abs*(OP2)

Side Effects: CARRY, INT_OVFL

Precision: OP 1 and OP2 have the same precision as the modifier.

Formal Description:

```
define ABS.p:qhsd =    XOP[p;p] if op2 ≥ 0
                                then (op1 ⇐ op2 also Int_Ovfl ⇐ 0)
                                else Subtract (0,op2)→ dif, c, ov next
                                     Int_Overflow? next
                                     op2 ⇐ dif
                               fi;
```

The following takes the absolute value of RTB and puts it in RTA. 1

```
ABS RTA,RTB      ;RTA←|RTB|
```

MIN

Instruction: **MIN . {Q,H,S,D}**
Class: TOP Integer minimum.

Purpose: DEST←*min*(S1,S2). The smaller of the signed integers S 1 and S2 is placed in DEST.

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

Formal Description:

```
define MIN. p:qhsd≡      TOP [p;p;p]  dest ⇐ (if S1 < s2  then  S1  else  s2 fi);
```

The following sets RTA to 0 if RTA is negative.

```
        MIN RTA,RTA,#0
```

1

MAX

Instruction: MAX . {Q,H,S,D}
Class: TOP **Integer maximum**

Purpose: DEST←*max*(S1,S2). The **larger** of the signed integers S 1 and S2 is placed **in** DEST.

Precision: S 1, S2, and DEST **all** have **the** precision specified by **the** modifier.

Formal Description:

```
define MAX. p:qhsd≡      TOP[p;p;p]  dest ⇐ (if S1 > s2  then  S1  else  s2 fi);
```

The following sets RTA to 100 if RTA is greater than 100.

```
MAX RTA,RTA,#c100.↵
```

5.2.2 Unsigned Integer

Unsigned integer instructions operate upon the unsigned integer data type (see Section 3.2). The instructions perform unsigned multiplication and unsigned integer division. Instructions that allow extended-precision operations (e.g., multiplying two single-word integers and producing a double-precision result) have an "L" as the last character of the opcode.

These instructions were designed to be used for arithmetic on numbers of arbitrarily great precision (as exemplified by "bignums" in MacLISP). Note that ADD and SUB work correctly for bignum arithmetic.

Section 5.2.3 describes the possible side effects of unsigned-integer instructions (INT_OVFL and INTZDIV).

UMULT

Instruction: **UMULT . {Q,H,S,D}**
Class: TOP Unsigned integer multiply

Purpose: Do an unsigned multiplication of S1 and S2 and place the low-order {quarter, half, single, double]-word of the result in DEST.

Side Effects: INT-OVFL

Precision: S 1, 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 $2^{18}-2^{10}$tl, that is, 001. Since the full result is greater than 2^9-1, INT-OVFL is also set.

```
UMULT.Q RTA,?777,?777
```

The only difference between UMULT and MULT is that UMULT sets INT-OVFL whenever MULT does, and, in addition, whenever the high order bit of one of its operands is set, and the (unsigned) magnitude of the other operand is greater than unity.

UMULTL

Instruction: UMULTL . {Q,H,S}
Class: TOP Unsigned integer multiply long

Purpose: Do an unsigned multiplication of S1 and S2 and place the result in DEST.

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

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
```

UDIV

Instruction: **UDIV. {Q,H,S,D}**
Class: TOP Unsigned integer divide

Purpose: The result of unsigned, integer division, **S1/S2,** is placed in **DEST**. The unsigned, integer remainder, **S1** *rem* S2, is placed in **NEXT(DEST)**;.

Side Effects: **INT_OVFL, INT_Z_DIV**

Precision: S 1, S2, DEST, and NEXT(DEST) all have the same precision as the modifier.

The following sets RTA to the unsigned quotient-remainder of 2^9-3 divided by twenty-two.

```
UDIV.Q RTA,?775,?26      ;RTA=027003( two QWs)
```

UDIVL

Instruction: **UDIVL . {Q,H,S}**
Class: TOP Unsigned integer divide long

Purpose: The **result** of **unsigned,** integer division, **S1/S2,** is placed in DEST. The unsigned, integer remainder, **S1** rent S2, is placed in **NEXT(DEST);.**

Side Effects: INT_OVFL, INTZDIV

Precision: **S2,** DEST, and **NEXT(DEST)** all have **the same** precision as **the** modifier. **S1** has a precision twice that of **the** modifier.

The following sets RTA to the unsigned quotient-remainder of 377377 (octal) divided by 777 (octal).

```
UDIVL.Q RTA;?377377,?777        ;RTA=377776 (two QWs)
```

5.2.3 Instruction Side Effects

USER-STATUS records three types of side effects that can occur during the execution of an integer instruction. (See Section 2.52 for a description of USER-STATUS.) They are: CARRY, INT_OVFL (integer overflow), and INT_Z_DIV (divide-by-zero). All of these bits in USER-STATUS are not *sticky,* that is, if an instruction can set one of these bits, it must either set or clear that bit.

5.2.3.1 CARRY

For each instruction shown, USER_STATUS.CARRY is set if the following formula is true with the indicated substitutions. CARRY is cleared if the formula is false. C_IN refers to the state of CARRY at the beginning of the instruction (used in ADDC, SUBC, and SUBCV).

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

In the following table, the result of the instruction equals X1+X2+X3; "~" 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	Sl	~S2	1	
SUBV	~S1	s2	1	
SUBC	S1	~S2	C-IN	
SUBCV	~S1	s2	C-IN	
INC	1	OP2	0	(i.e., OP2 = -1)
DEC	-1	OP2	0	(i.e., OP2 ≠ 0)
NEG	0	~OP2	1	(i.e., OP2 = 0)
ABS	0	~OP2	1	(i.e., OP2 = 0)

Table 5-l
Conditions for setting CARRY

No other instructions change CARRY.

For example, the following instruction sets CARRY.

```
INC RTA,#-1      ;RTA←0
```

5.2.3.2 INT_OVFL

USER-STATUS. INT-OVFL is set when the result of an operation will not fit in the destination, that is, if the destination precision is {Q,H,S,D}, then overflow occurs if the result is not between $-2^{\{8,17,35,71\}}$ and $2^{\{8,17,35,71\}}-1$ inclusive. Instructions which set/clear INT-OVFL are: ADD, ADDC, SUB, SUBV, SUBC, SUBCV, INC, IJMP, IJMPZ, IJMPA, ISKP, DEC, DJMP, DJMPZ, DJMPA, DSKP, FIX, SHFA, SHFAV, MULT, QUO, QUOV, QU02, QUO2V, QUOL, QUOLV, QUO2L, QUO2LV, DIV, DIVV, DIVL, DIVLV, NEG, ABS, TRANS, UMULT, UDIV, and UDIVL. No other instructions change INT-OVFL. It should be noted that INT_OVFL is not set during the exponentiation in the QUO2 class of instructions. For these instructions, unlimited precision is available for the 2^S section of the computation.

The condition for determining INT-OVFL is simplified when considering the addition and subtraction instructions (ADDs, SUBs, INC, DEC, IJMPs, DJMPs, ISKP, and DSKP). With these instructions, INT-OVFL is set when the carry into the high-order bit of the result is not the same as the carry out of that bit.

When an integer overflow occurs, the action taken depends on the USER-STATUS. INT_OVFL_MODE bit. If equal to zero, a trap occurs and no value is stored. If equal to one, all instructions (except SHFA to the left) store the low-order bits of the result. SHFA to the left stores the correct sign followed by the low-order bits of the result.

For example, the following instruction sets INT-OVFL.

```
INC RTA, #c377777,,777777ↄ          ;RTA←MINNUM; constant is MAXNUM
```

5.2.3.3 INT_Z_DIV

USER-STATUS, INT_Z_DIV is set when a divide-by-zero occurs in an integer division. Instructions which set/clear INT_Z_DIV are: QUO, QUOV, QUOL, QUOLV, REM, REMV, REML, REMLV, MOD, MODV, MODL, MODLV, DIV, DIVV, DIVL, DIVLV, UDIV, UDIVL. No other instructions change INT_Z_DIV.

When an integer divide-by-zero occurs, the action taken depends on the USER_STATUS.INT_Z_DIV_MODE bit. If INT_Z_DIV_MODE=0 then a trap occurs and no value is stored in the destination. If INT_Z_DIV_MODE=1 then zero is stored and no trap occurs.

5.3 Floating Point

Floating-point instructions operate on the floating-point data type (see Section 3.3). The instructions include addition, subtraction, multiplication, division, absolute value, negation,

minimum, maximum, and scaling by powers of two. Reverse instructions are provided for the non-commutative operations (subtraction, division, and scaling). These reverse instructions have a "V" added to the end of the opcode mnemonic (e.g., FSC becomes FSCV). Extended-precision operations are provided for multiplication and division (e.g., multiplying two single-word floating-point numbers and producing a double-precision result). Multiplication (FMULTL) produces an extended-precision product and division (FDIVL, FDIVLV) utilizes an extended-precision dividend.

All operations producing a floating-point result normalize that result. (See Section 3.3 for a discussion of the floating-point format. This format does not permit the representation of unnormalized numbers.)

5.3.1 Rounding Modes

During floating-point operations, rounding of the result may be necessary. With the exception of the FIX instruction, the rounding mode used is specified by USER_STATUS.RND_MODE, as described below. The FIX instruction allows the explicit specification of a rounding-mode or the use of RND-MODE.

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 RND-MODE have the following functions (reversals of rounding direction accumulate):

RND_MODE<0> 0: Round as specified by RND_MODE<1:4>.
1: Reserved.

RND_MODE<1> 0: If $F \neq 0$, round as specified by RND_MODE<2:4>; otherwise deliver R exactly.
1: If F = 1/2 then round as specified by RND_MODE<2:4>; otherwise round to the floating-point number nearest to R.

RND_MODE<2> 0: Round toward negative infinity.
1: Round toward positive infinity.

RND_MODE<3> 0: No effect.
1: If and only if N's mantissa's least significant bit is a one, reverse the rounding direction.

RND_MODE<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:

RND MODE (octal)	Function	Modifier for FIX
0	Floor	FL
1	Diminished Magnitude	DM
4	Ceiling	CL
5	Augmented Magnitude	
12	Half Rounds Toward Positive	HP
14	PDP-10 FIXR Rounding	
15	App. PDP-10 FLTR Rounding	

Table 5-2
Useful Rounding Modes

5.3.2 Instruction Side Effects

USER-STATUS records three types of side effects that can occur during the execution of a floating-point instruction. (See Section 2.5.2 for a description of USER-STATUS.) They are: FLT-OVFL (floating overflow), FLT_UNFL (floating underflow), and FLT_NAN (floating undefined). All of these bits in USER-STATUS are *not sticky*, that is, if an instruction can set one of these bits, it must either set or clear that bit.

5.3.2.1 FLT,OVFL and FLT,UNFL

USER_STATUS.FLT_OVFL is set when a floating-point instruction produces a result with an exponent that is too large to be represented in the EXP-field of the destination (i.e., OVF or MOVF). (See Section 3.3 for a description of the floating-point data type.) In a similar way, FLT_UNFL is set when a floating-point instuction's result has a negative exponent whose magnitude is too large to be represented in the destination's EXP (i.e., UNF or MUNF). Floating underflows and overflows generally occur in two situations. The first situation is that the result of an op.eration (e.g., FMULT) is out of range of the EXP-field. The second situation is when the result 'will fit, but the post-normalization of that result causes the exponent not to fit.

All instructions that produce floating-point results set/reset FLT-OVFL and FLT_UNFL. It should be noted that FSC and FSCV do not set either overflow or underflow during their exponentiation calculations. In these two instructions, the 2^S part of the calculation is done with unlimited precision.

When a floating underflow (overflow) occurs, the action taken depends on the USER-STATUS. FLT_UNFL_MODE (USER-STATUS, FLT_OVFL_MODE) field.

FLT_UNFL_MODE<0:1>	Result
0	Trap and do **not store** any value in the result
1	Store the infinitesimal with the correct sign (UNF or MUNF)
2	Store the floating-point number with the correct sign and mantissa, but with a wrapped-around exponent
3	Not defined

Table 5-3
USER-STATUS-UNFL-MODE

FLT_OVFL_MODE<0:1>	Result
0	Trap and do not store any value in the result
1	Store the infinity with the correct sign (OVF or MOVF)
2	Store the floating-point number with the correct sign and mantissa, but with a wrapped-around exponent
3	Not defined

Table 5-4
USER_STATUS_OVFL_MODE

See Section 5.3.2.3 for a discussion of how OVF, MOVF, UNF, and MUNF propagate in floating-point instructions (when they do not trap).

The first instruction sets FLT_OVFL, the second sets FLT_UNFL.

```
FSUBV. H RTA, #0,#c400000ɔ
FSC. H RTA,#c004000ɔ,#-1
```

5.3.2.2 FLT,NAN

USER-STATUS, FLTNAN is set when a NAN is the result of a floating-point operation. All instructions that require floating-point arguments and produce floating-point results set/reset FLT_NAN.

When an undefined floating-point number (NAN) is produced the action taken depends on the USER_STATUS.FLT_NAN_MODE bit. If FLT_NAN_MODE=0 then a trap occurs and no value is stored in the destination. If FLT_NAN_MODE=1 then NAN is stored and no trap occurs.

See Section 5.3.2.3 for a discussion of how NAN propagates in floating-point instructions (when it does not trap).

5.3.2.3 Exception Propagation

When the traps are disabled (as explained in the previous sections) the exception values (OVF, MOVF, UNF, MUNF, NAN) can propagate through floating-point instructions. The diagrams below describe how the exceptions propagate through addition, multiplication, and division. Floating-point subtraction behaves with respect to exception propagation as if FNEG were applied to the second argument, and then FADD applied,

FMIN and FMAX propagate the exceptions as regular floating-point numbers (i.e., MOVF<-X<MUNF<0<UNF<X<OVF), but the result is NAN if either argument is NAN. FNEG(MOVF)=OVF, FNEG(OVF)=MOVF, FNEG(MUNF)=UNF, and FNEG(UNF)=MUNF. Similarly, FABS(MOVF)=OVF and FABS(MUNF)=UNF. FTRANS acts as an identity function for all five exceptions. FIX of any special floating-point symbol produces an intermediate NAN result and stores the result on the basis of FLT_NAN_MODE. The exponentiation portion of the FSC and FSCV is effectively done in infinite precision and will not produce an exception; the subsequent multiplication follows the rules given below.

In the following tables, X and Y are assumed to be any positive floating-point numbers, excluding the special floating-point symbols 0, UNF, and OVF.

Addition (A+B)

A ↓ B →	MOVF	-Y	MUNF	0	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	NAN	Y	OVF	NAN
0	MOVF	-Y	MUNF	0	UNF	Y	OVF	NAN
UNF	MOVF	-Y	NAN	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

Figure 5-l
Floating-point Exception Propagation (t)

Multiplication (A*B)

A ↓ B →	MOVF	-Y	MUNF	0	UNF	Y	OVF	NAN
MOVF	OVF	OVF	NAN	0	NAN	MOVF	MOVF	NAN
-X	OVF	X*Y	UNF	0	MUNF	-X*Y	MOVF	NAN
MUNF	NAN	UNF	UNF	0	MUNF	MUNF	NAN	NAN
0	0	0	0	0	0	0	0	NAN
UNF	NAN	MUNF	MUNF	0	UNF	UNF	NAN	NAN
X	MOVF	-X*Y	MUNF	0	UNF	X*Y	OVF	NAN
OVF	MOVF	MOVF	NAN	0	NAN	OVF	OVF	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Figure 5-2
Floating-point Exception Propagation (*)

Division (A/B)

A ↓ B →	MOVF	-Y	MUNF	0	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
0	0	0	0	NAN	0	0	0	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

Figure 5-3
Floating-point Exception Propagation (/)

FADD

Instruction: **FADD . {H,S,D)**
Class: TOP Floating-point add

Purpose: The floating-point **sum, S1** plus S2, is rounded according to **RND_MODE** and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLT_NAN

Precision: S 1, **S2,** and DEST all **have the** precision specified **by the** modifier.

To add 1.0 to RTA either of the first two instructions could be **used.** Note that FASM provides an interpretation of floating-point **constants.** The **third** instruction **doubles** RTA. Alternatively, FMULT, FSC, or FDIV **might be used.**

```
FADD RTA, #c200400,,0>
FADD RTA,#c1.0>
FADD RTA,RTA              ;RTA←2*RTA; FSC RTA,#1 is perhaps cheaper
```

FSUB

Instruction: **FSUB . {H,S,D}**
Class: TOP Floating-point subtract

Purpose: The floating-point difference, **S1** minus **S2,** is rounded according to **RND_MODE** and stored in DEST.

Side Effects: FLT,OVFL, **FLT_UNFL,** FLTNAN

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

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

```
FSUB RTA,#c1.0>          ;RTA←RTA-1.0
```

FSUBV

Instruction: **FSUBV .{H,S,D}**
Class: TOP Floating-point subtract reverse

Purpose: The floating-point difference, S2 **minus S1,** is rounded according to **RND_MODE** and stored in DEST.

Side Effects: **FLT_OVFL, FLT_UNFL,** FLTNAN

Precision: S **1, S2,** and DEST all have the precision specified by the modifier.

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

```
FSUBV RTA,#<1.0>                ;RTA←1.0-RTA
```

FMULT

Instruction: **FMULT . {H,S,D}**
Class: TOP Floating-point multiply

Purpose: The floating-point product, S1 times S2, is rounded according to RND_MODE and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

The following instruction doubles the value in RTA. Alternately, FSC, FADD, or FDIV might be used for this purpose.

```
FMULT RTA,#c2.0>          ;RTA←RTA*2.0
```

FMULTL

Instruction: **FMULTL.{H,S}**
Class: TOP Floating-point multiply long

Purpose: The floating-point product, S 1 times S2, is rounded according to RND,MODE and stored in DEST. Note that the long result format will have more than twice as many MANT bits as either operand.

Side Effects: **FLT_OVFL,** FLTUNFL, FLTNAN. (These can occur only if one of the floating-point exception values occurs as an argument. If both arguments are ordinary floating-point numbers, the result cannot overflow or underflow, because the long result format has a larger EXP field than the operands do.)

Precision: S 1 and S2 have the same precision as the modifier. DEST has precision *twice* that of the modifier.

The following instruction will give RTA ail significant bits of the square of the value in X (unless overflow or underflow occurs).

```
FMULTL RTA,X,X;RTA←X↑2
```

FDIV

Instruction: **FDIV . {H,S,D}**
Class: TOP Floating-point divide

Purpose: The floating-point quotient, Sl divided by S2, is rounded according to RND,MODE and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN

Precision: S 1, S2, and DEST ail have the precision specified by the modifier.

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

```
FDIV RTA,#c200000,,0>    ;RTA←RTA/0.5=2*RTA
```

FDIVV

Instruction: **FDIVV . {H,S,D}**
Class: TOP Floating-point divide reverse

Purpose: The floating-point quotient, S2 divided by S1, is rounded according to RND_MODE and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN

Precision: S 1, S2, and DEST **ail have** the precision specified by the modifier.

The following code might be used to set RTA to its reciprocal.

```
FOIVV RTA,RTA,#c1.0ↄ
```

FDIVL

Instruction: **FDIVL . {H,S}**
Class: TOP Floating-point divide long

Purpose: The floating-point quotient, S1 divided by S2, is rounded according to RND_MODE and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN

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

The following uses a long 1.0 to reciprocate RTA. Note that this is NOT the same constant as would be used for FDIV.

```
FDIVL RTA,#c200100000000+0>,RTA
```

FDIVLV

Instruction: FDIVLV .{H,S}
Class: TOP Floating-point divide long reverse

Purpose: The floating-point quotient, S2 divided by S1, is rounded according to RND_MODE and stored in DEST.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN

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

The **following uses a** SW 1.0 **to reciprocate** RTA. Note **that** this is NOT the **same** constant as would **be** used for FDIV.H.

```
FDIVLV.H RTA,#c200400,,0
```

FSC

Instruction: **FSC . {H,S,D}**
Class: TOP Floating-point scale

Purpose: The floating-point product, S1 times 2^{S2}, is rounded according to RND_MODE and stored in DEST. S 1 is a floating-point number and S2 is a signed integer.

Side Effects: FLT_OVFL, FLT_UNFL, FLT_NAN. (FLT_OVFL and FLT_UNFL are not set during the 2^{S2} portion of the operation. This exponentiation is done with unlimited precision.)

Precision: S 1 and DEST have the same precision as the modifier. S2 is a single-word.

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

```
FSC RTA,#1        ;RTA←RTA*2↑(1)=2*RTA
```

FSCV

Instruction: **FSCV . {H,S,D}**
Class: TOP Floating-point scale reverse

Purpose: The floating-point product, S2 times 2^{S1}, is rounded according to RND_MODE and stored in DEST. S2 is a floating-point number and S1 is a signed integer.

Side Effects: FLT-OVFL, FLT_UNFL, FLTNAN. (FLT-OVFL and FLTUNFL are **not** set during **the** 2^{S1} portion of **the** operation. This exponentiation is done with unlimited precision.)

Precision: S2 **and** DEST **have the same** precision as the modifier. S1 is a single-word.

The following two instructions set RTA to the average of X and Y.

```
FADD RTA,X,Y
FSCV RTA,#-1,RTA
```

FIX

Instruction: FIX. {FL,CL,DM,HP,US} . {Q,H,S,D} . {H,S,D}
Class: XOP Fix floating-point number

Purpose: Convert the floating-point number specified by OP2 into an integer and store it in OP1. Use the rounding mode specified by the first modifier.

Side Effects: INT_OVFL

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

The following converts a floating point value in RTA into an integer. The exact result depends on the value and the rounding mode specified in USER_STATUS.RND_MODE.

```
FIX.US RTA,RTA
```

FLOAT

Instruction: **FLOAT . {H,S,D} . {Q,H,S,D}**
Class: XOP Float fixed-point number

Purpose: Convert the integer specified by OP2 into a floating-point number and store it in OP 1.

Side Effects: FLT_OVFL. (This can occur only in the cases of FLOAT.H.S and FLOAT.H.D.)

Precision: OP 1 has the precision of the first modifier. OP2 **has** the precision of the second modifier.

The following loads RTA with the floating point value 1.0.

```
FLOAT RTA,#1      ;RTA=200400,,0 (SW)
```

FTRANS

Instruction: **FTRANS . {H,S,D} . {H,S,D}**
Class: XOP Floating-point transfer

Purpose: Take the floating-point number specified by OP2 and make it a floating-point number of the precision of the first modifier. Store the result in OP1.

Side Effects: FLT_OVFL, FLT_UNFL, FLTNAN. If OP2 has no greater precision than OP 1, 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 the USER_STATUS.RND_MODE.

```
FTRANS.S.D RTA,#c200100000000 ↔ 0ↄ      ;RTA=200400,,0=1.0
```

FNEG

Instruction: **FNEG . {H,S,D}**
Class: XOP Floating-point **negate**

Purpose: Take **the** floating-point negation of OP2 **and** store it in OP1. The primary difference **between** NEC **and** FNEC **is that** FNEC properly propagates the floating-point exception **values.** They **also have** different side effects.

Side Effects: FLT-OVFL, FLT_UNFL, FLTNAN

Precision: OP 1 and OP2 **have the** same precision as the modifier.

These **examples show how floating-point** exceptions are propagated **by** FNEC. 1

```
FNEG.H RTA,#c000001>     ;RTA←MUNF, s i gna l FLT_UNFL
FNEG.H RTA,#c400001>     ;RTA←OVF, s i gna l FLT-OVFL
FNEG . H RTA,#c400000>   ;RTA←NAN, s i gna l FLT_NAN
```

FABS

Instruction: FABS.{H,S,D}
Class: XOP Floating-point absolute value

Purpose: Take the floating-point absolute value of OP2 and store it in OP1. The primary difference between ABS and FABS is that FABS properly propagates the floating-point exception values. They also have different side effects.

Precision: OP 1 and OP2 have the same precision as the modifier.

Side Effects: FLT-OVFL, FLT_UNFL, FLTNAN

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

```
ABS. H RTA, #c-1>          ;RTA←UNF, no side effects
FABS.H RTA,#c-1>           ;RTA←UNF, signal FLT_UNFL
ABS. H RTA, #c377777>      ;RTA←OVF, no side effects
FABS. H RTA, #c377777>     ;RTA←OVF, signa l FLT-OVFL
ABS.H RTA ,#c-400000>      ;RTA←NAN, signal INT_OVFL
FABS.H RTA,#c-400000>      ;RTA←NAN, signa lFLT_NAN
```

FMIN

Instruction: **FMIN.{H,S,D}**
Class: TOP Floating-point minimum

Purpose: DEST←*min*(S1,S2). The smaller of the floating-point numbers S1 and S2 is placed in DEST. The primary difference between MIN and FMIN is that FMIN properly propagates the floating-point exception values.

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

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

```
FMIN RTA,X,#<43.0>
```

FMAX

Instruction: FMAX . {H,S,D}
Class: TOP Floating-point **maximum**

Purpose: DEST←*max*(S1,S2). The larger of the floating-point numbers S1 and S2 is placed in DEST. The primary difference between MAX and FMAX is **that** FMAX properly propagates **the** floating-point exception **values.**

Precision: S 1, S2, **and** DEST all have **the** precision specified **by** the modifier.

This **sequence of instructions** takes **the number** FOO and "clips" it to **be** within the window [0.0, 1 .0].

```
FMAX RTA,FOO,?0.0      ; larger of FOO and 0.0 to RTA
FMIN FOO,RTA,?1.0      ;smaller of that and 1.0 to FDO
```

5.4 Move

Move instructions are used to move operands and/or addresses of operands to memory locations and/or registers. Many words may be moved by the single instructions MOVMQ and MOVMS. Single registers can be saved and loaded with a single instruction using SLR or SLRADR. Virtual or physical addresses can be loaded using MOVADR or MOVPHY. The precisions associated with each move instruction are described in the instruction descriptions.

MOV

Instruction: **MOV . {Q,H,S,D} . {Q,H,S,D}**
Class: XOP Logical **move**

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, and** FTRANS to perform moves of floating-point numbers.

Precision: The two modifiers specify the **precisions** of OP1 and OP2 respectively.

Formal Description:

```
define MOV. p1:qhsd.p2: qhsd ≡  XOP[p1;p2]  op1⇐ low (p1, zero-extend(op2, 72));
```

The following copies the low-order QW of RTA into the high-order QW.

```
MOV.Q.Q RTA,c23⊃
```

MOVMQ

Instruction: **MOVMQ .{2..32,64,128}**
Class: XOP Move many quarter-words

Purpose: Moves the number of quarter-words, specified by the modifier, from the locations starting at ADDRESS(OP2) to the locations starting at ADDRESS(OP1). If the source and destination regions overlap, the result is undefined. If either OP1 or OP2 is an immediate constant, a hard trap will occur.

Precision: This instruction deals with quarter-words for both source and destination precisions.

The following copies the three high-order QWs from RTA into RTB. 1

```
MOVMQ.3 RTB,RTA
```

MOVMS

Instruction: **MOVMS . { 2 .. 32 }**
Class: XOP Move many single-words

Purpose: Moves the number of single-words, specified by the modifer, from the locations starting at ADDRESS(OP2) to the locations starting at ADDRESS(OP1). If the source and destination regions overlap, the result is undefined. If either OP1 or OP2 is an immediate constant, a hard trap will occur.

Precision: This instruction deals with single-words for both source and destination precisions.

The following saves all the registers from RTA on in a block starting at SAVEBK.

```
ROVRS.28 SAVEBK,RTA
```

EXCH

Instruction: **EXCH . {Q,H,S,D}**
Class: XOP Exchange words

Purpose: Exchange the values OP1 and OP2. If either OP 1 or OP2 is an immediate constant, a hard trap will occur.

Precision: OP 1 and OP2 each have the precision specified by the modifier.

Formal Description:

```
define EXCH.p:qhsd ≡   XOP[p,RW;p,RW]  let temp = op2
                                        then op2 ← op1 next op1 ← temp;
```

The following swaps RTA and RTB.

```
EXCH RTA, RTB
```

SLR

Instruction: **SLR . { 0 .. 31 }**
Class: XOP **Save and** load register

Purpose: OP1 is replaced by the contents of the register named by the modifier. The contents of the register is then replaced by OP2.

Precision: A **11** operands involved are single-words.

Formal Description:

define **SLR.***n*:*n*0to31 ≡ *X O P* **[S;S] let temp = R[***n***]**
then R[*n***]←op2** next **op1⇐ temp;**

The first instruction moves RTA into RTB and **zeros** RTA. The **second** and third instructions illustrate the results when one of the operands is the register specified in the instruction. The fourth illustrates the result when the operands are the same.

```
SLR.4 RTB, #0      ; RTB←RTA, RTA←0
SLR.4 RTA, FOO     ;alternate NOP
SLR.4 FOO,RTA      ;alternate ROV FOO,RTA
SLR.4 FOO,FOO      ;alternate EXCH RTA,FOO
```

SLRADR

Instruction: **SLRADR .{ 0 . . 31 }**
Class: XOP Save and load register with address

Purpose: OP1 is replaced **by** the **contents** of **the** register named **by** the modifier. The contents of the register is then replaced by ADDRESS(OP2).

Precision: A 11 operands involved are single-words.

Formal Description:

define **SLRADR**, n: *n0to31* = *XOP* [S;S,A] let **temp** = **R**[*n*]
then **R**[*n*]← *Address* (op2) next op1⇐ **tempt**

The first instruction moves RTA into RTB and puts ADDRESS(FOO) in RTA. The second **and third instructions-illustrate the results when one of the operands** is the register specified in **the** instruction. The fourth illustrates the result when the operands are the same.

```
SLRAOR.4 RTB,FOO      ; RTB←RTA, RTA←ADDRESS (FOO)
SLRAOR. 4 RTA, FOO    ;alternate NOP
SLRADR.4 FOO,RTA      ;alternate MOV FOO,RTA; ROVAOR RTA,RTA
SLRADR.4 FOO,FOO      ;alternate ROV FOO,RTA; ROVADR RTA,FOO
```

MOVADR

Instruction: MOVADR
Class: XOP Move address

Purpose: OP1←ADDRESS(OP2). If OP2 is an immediate constant, a hard trap will occur.

Precision: OP 1 is a single-word.

Formal Description:

```
define MOVADR = XOP[S;S,A] op1← Address(op2);
```

The first instruction loads RTA with the address of the operand FOO.

```
MOVADR RTA,FOO  ;RTA←ADDRESS(FOO)
MOVADR--RTA,RTA  ;RTA←20 octal (RTA is register 4, at address 4*4=20)
```

MOVPHY

Instruction: **MOVPHY**
Class: XOP Move physical address

Purpose: OP1←PHYSICAL_ADDRESS(OP2). If OP2 is an immediate constant, a hard trap will occur. If ADDRESS(OP2) is in the range 0 . . 127 **then** the physical address of the corresponding shadow memory location will **be used.** See Section 2.4.1 for a discussion of **shadow memory.**

Restrictions: Illegal in user mode.

Precision: OP1 is a single-word.

Formal Description:

```
define MOVPHY = XOP-IS; S, PA3 op1 ⇐ Physical-Address (op2);
```

The following **loads** RTA with **the** *physical* address of FOO.

```
MOVPHY RTA,FOO ;RTA←PHYSICAL_ADDRESS(FOO)
```

5.5 Flag

Flag instructions produce results that are of the flag data type. The flag data type is discussed in Section 3.8. The flag results are always single-words. A flag is either all zeros or all ones. All zeros means true. All ones means false.

CMPSF compares two words according to a specified condition. It returns true if the condition was satisfied and false if it was not. BNDSF checks if its argument is within a given bounds and returns the appropriate flag.

CMPSF

Instruction: **CMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D}**
Class: TOP Compare and set flag

Purpose: DEST←S 1 *condition S2,* where *condition* is the first modifier.

Precision: S 1 and S2 have the same precision as the modifier. DEST is a single-word.

Formal Description:

define CMPSF. *rel: acond.p; qhsd* ≡ *TOP* [S;*p*;*p*] dest ⇐ **(if** *rel*(S1,s2) **then -1 else** 0 **fi);**

Let X, Y, and Z be single-words, 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 RTA,Z,#0
MOV RTA, <Y>(RTA)          ;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 RTA,NUM,#0       ;all bits of RTA get the sign bit of NUM
```

The effect of CMPSF.Icond can be obtained by an AND or ANDCT followed by a CMPSF.EQL or CMPSF.NEQ.

```
ANDCT RTA,FOO,BAR          ;this behaves as would the fictional
CMPSF. EQL RTA ,#0         ;instruction CMPSF.NON RTA,FOO,BAR
```

BNDSF

Instruction: **BNDSF . {B,MIN,M1,0,1} . {Q,H,S,D}**
Class: TOP Bounds-check and set flag

Purpose: The first modifier determines if S2 is compared against a constant and S1, or against Sl and NEXT(S 1). If the first modifier is B then if S 1≤S2≤NEXT(S1) then DEST←TRUE else DEST←FA LSE. If the first modifier is one of MIN, M 1, 0, and 1 then if *constant*≤S2≤S 1 then DEST←TRUE else DEST←FALSE. *Constant*=- 1 if the first modifier is M 1. *Constant*=0 if the first modifier is *0. Constant4* if the first modifier is 1. If the first modifier is MIN then *constant* is the negative number with the greatest magnitude for the precision specified by the second modifier.

Precision: Sl and S2 have the same precision as the second modifier. DEST is a single-word. If NEXT(S 1), 0, 1, -1, or MIN is used it also has the same precision as the second modifier.

The following two instructions are alternate implementations for setting RTA to -1 if X contains the ASCII representation of a digit, and to 0 otherwise. In the first instruction FASM places the string "09" on a data page automatically.

```
BNDSF.B.Q RTA,["09"],X
BNDSF.0.Q RTA,#11,#c-"0"⊃(X)      ;X must be a register
```

5.6 Boolean

Boolean instructions operate upon the boolean data type (see Section 3.1). All boolean instructions can operate on any of the four data precisions (QW,HW,SW,DW). Both operands must be of the same precision. The result of a boolean operation has the same precision as the operands. Note that none of the condition bits (e.g., carry or integer-overflow) can be set by boolean instructions.

The three-operand boolean instructions ANDTC, ANDCT, ORTC, and ORCT are not symmetric in their use of S 1 and S2. Nevertheless, instructions named ANDTCV, ANDCTV, ORTCV, and ORCTV are not provided. This is because the reverse form of ANDTC is provided by ANDCT, of ANDCT by ANDTC, of ORTC by ORCT, and of ORCT by ORTC.

NOT

Instruction: NOT.{Q,H,S,D}
Class: XOP

Logical (bit-wise) NOT

Purpose: OP 1←*one's-complement*(OP2)

Precision: OP 1 and OP2 have the same precision as the modifier,

Formal Description:

define NOT, *p:qhsd*≡ *XOP*[*p*;*p*] op1 ⇐ ¬ op2 ;

The following is an alternate to NEG RTA. 1

```
NOT RTA,#c-1⊃(RTA)        ;RTA←-RTA
```

AND

Instruction: **AND . {Q,H,S,D}**
Class: TOP Logical (bit-wise) AND

Purpose: DEST←S1∧S2

Precision: S **1, S2, and** DEST **all have the** precision specified **by the** modifier.

Formal Description:

define AND. *p*:*qhsd* E TOP[*p*;*p*;*p*] dest ⇐ S1 A s2;

The following instruction illustrates **the** effect of all possible combinations of bits in the operands.

```
AND.Q RTA,#3,#5          ; RTA=1
```

ANDTC

Instruction: **ANDTC . {Q,H,S,D}**
Class: TOP Logical (bit-wise) A ND true/complement

Purpose: DEST←S1∧*one's-complement*(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.

Precision: S 1, S2, **and DEST all have** the precision specified by the modifier.

Formal Description:

```
define ANDTC.p:qhsd = TOP[p;p;p] dest ← S1 ∧ (¬ s2);
```

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
ANDTC.Q RTA,#3,#5                ;RTA=2
```

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 decremented as an integer "in place" **in** WORD, without affecting **non-selected bits** of WORD. This can be done as follows.

```
AND RTA,WORD,MASK        ;RTA←WORD with non-selected bits zeroed
DEC RTA                  ;zeroed bits propagate the borrow
AND RTA,MASK             ;mask out non-selected bits
ANDTC WORD,MASK          ;mask out SELECTED bits in WORD
OR WORD,RTA              ;merge the two results
```

ANDCT

Instruction: ANDCT.{Q,H,S,D}
Class: TOP Logical (bit-wise) AND complement/true

Purpose: DEST←*one's-complement*(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.

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

Formal Description:

```
define ANDCT. p:qhsd≡  TOP[p;p;p]  dest  ⇐(¬ S1) A s2;
```

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
ANDCT.Q RTA,#3,#5                ; RTA=4
```

OR

Instruction: **OR . {Q,H,S,D}**
Class: TOP Logical (bit-wise) OR

Purpose: **DEST←S1vS2**

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

Formal Description:

define OR. *p:qhsd* ■ *TOP [p;p;p]* dest ⇐ S1 v s2;

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
OR.Q RTA,#3,#5          ;RTA=7
```

ORTC

Instruction: **ORTC . {Q,H,S,D}**
Class: TOP Logical **(bit-wise)** OR true/complement

Purpose: DEST←S1∧*one's-complement*(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.

Precision: S 1, S2, and DEST all have the precision specified **by the** modifier.

Formal Description:

define ORTC. *p: qhsd* ≡ TOP [*p*;*p*;*p*] dest ⇐ S1 v (¬ s2);

The following instruction illustrates **the** effect of all possible combinations of bits in **the** operands.

```
ORTC.Q RTA,#3,#5          ;RTA=773
```

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 RTA,WORD,MASK        ;RTA←WORD with non-selected bits set to one
INC RTA                   ;one bits propagate the carry
AND RTA,MASK              ;mask out non-selected bits
ANOTC WORD, MASK          ;mask out SELECTEO bits in WORD
OR WORD,RTA               ;merge the two results
```

ORCT

Instruction: **ORCT . {Q,H,S,D}**
Class: TOP Logical (bit-wise) OR complement/true

Purpose: DEST←*one's-complement*(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.

Precision: S 1, S2, and DEST all **have** the precision specified **by the** modifier.

Formal Description:

```
define ORCT, p:qhsd = TOP[p;p;p] dest ← (¬S1) v s2;
```

The following instruction illustrates **the** effect of all possible combinations of bits in the operands.

```
ORCT.Q RTA,#3,#5          ;RTA=775
```

NAND

Instruction: NAND . {Q,H,S,D}
Class: TOP Logical (bit-wise) NAND (NOT of AND)

Purpose: DEST←*one's-complement*(S1∧S2)

Precision: S 1, S2, **and** DEST all have the precision specified **by** the modifier.

Formal Description:

```
define NAND. p:qhsd e    T O P [p;p;p] d e s t ⇐ ¬ (S1 ∧ s2) ;
```

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
NAND.Q RTA,#3,#5          ; RTA-776
```

NOR

Instruction: **NOR . {Q,H,S,D}**
Class: TOP Logical (bit-wise) NOR (NOT of OR)

Purpose: DEST←*one's-complement*(S1∨S2)

Precision: S 1, S2, and DEST all have the precision specified by **the** modifier.

Formal Description:

```
define NOR. p:qhsd ε    TOP [p;p;p]  dest  ⇐ ¬(S1 v s2);
```

The following instruction illustrates the effect of all possible **combinations** of bits in the operands.

```
NOR.Q RTA,#3,#5          ;RTA=770
```

XOR

Instruction: **XOR . {Q,H,S,D}**
Class: TOP Logical (bit-wise) exclusive OR

Purpose: DEST←(S1∧*one's-complement*(S2)) v (*one's-complement*(S1)∧S2)

Precision: S 1, S2, and DEST all have the precision specified by the modifier.

Formal Description:

define XOR.*p*:*qhsd* ≡ *TOP*[*p*;*p*;*p*] d e s t ⇐ S1 ⊕ s2;

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
XOR. Q RTA, #3,#5          ;RTA=6
```

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 QUUX,ZTESCH
XOR ZTESCH, QUUX
XOR QUUX, ZTESCH
```

EQV

Instruction: **EQV .{Q,H,S,D}**
Class: TOP Logical (bit-wise) equivalence

Purpose: DEST←(S1∧S2) ∨ (*one's-complement*(S1)∧*one's-complement*(S2))

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

Formal Description:

```
define EQV. p:qhsd≡      T O P[p;p;p] d e s t ⇐ ¬ (S1 ⊕ s2) ;
```

The following instruction illustrates the effect of all possible combinations of bits in the operands.

```
EQV.Q RTA,#3,#5          ;RTA=771
```

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 QUUX, ZTESCH
EQV ZTESCH, QUUX
EQV QUUX,ZTESCH
```

5.7 Shift and Rotate

The shift and rotate instructions provide logical and arithmetic shifting of operands. Since all shift and rotate instructions are non-commutative, each instruction is also provided in its reverse form (e.g., SHF and SHFV).

Note that a left shift (rotate) by N is equivalent to a right shift (rotate) by -N for all the instructions in this section except for DSHF and DSHFV. The effect of these instructions is described individually.

SHF

Instruction: **SHF . {LF,RT} . {Q,H,S,D}**
Class: TOP Logical shift

Purpose: DEST←S I logically shifted {left,right} by S2. Bits shifted in are zero bits; bits shifted out are lost. Note that a left shift by S2 is identical to a right shift by -S2.

Precision: S2 is a single-word. DEST and S1 have the precision specified by the second modifier.

Formal Description:

```
define SHF.dir:lfrt.p:qhsd = TOP[p;p;S] dest ⇐ shift(S1, case dir of
                                                    LF: s2;
                                                    RT: -s2;
                                                   end);
```

The following shows the effect of a positive left-shift argument.

```
SHF.LF.Q RTA,#-1,#1      ;RTA=-2
```

SHFV

Instruction: **SHFV . {LF,RT} . {Q,H,S,D}**
Class: TOP Logical shift reverse

Purpose: **DEST←S2** logically shifted **{left,right}** by **S1**. Bits shifted in are zero bits; bits shifted out are lost. Note that a left shift by **S1** is identical to a right shift by -S 1.

Precision: S 1 is a single-word. DEST and S2 have the precision specified by the second modifier.

Formal Description:

```
define SHFV.dir:lfrt.p:qhsd■  TOP[p;S;p]  dest ⇐ shift(s2, case dir of
                                                    LF: S1;
                                                    RT: - S1;
                                                   end);
```

The following shows the effect of a negative left-shift argument.

```
SHFV.LF.Q RTA,#-1,#1     ;RTA=0
```

DSHF

Instruction: **DSHF.{LF,RT}.{Q,H,S}**
Class: TOP Logical double-width shift

Purpose: ⊂S 1 || NEXT(S 1)⊃ is logically shifted {left,right} by S2 positions. The {high-order,low-order] 9, 18, or 36 bits of the result (corresponding to Q, H, S respectively) are then stored in DEST. Note that ⊂S1|| NEXT(S 1)⊃ is not treated as a "long" operand, but as two separate operands (which is why the mnemonic is DSHF and **not** SHFL). This is useful for multi-word shifts of **any** of the three precisions allowed. Long right shifts **must** start at the right **end** of the multi-word vector, and **long** left shifts must start at the left end of the vector. Note that DSHF.RT by X is equivalent to DSHF.LF by (9-X), (18-X), (36-X).

Precision: ⊂S 1 || NEXT(S 1)⊃ is considered to **be** two {Q,H,S}-precision words (rather than one {H,S,D}-precision word) for alignment purposes.

The following illustrates the result of shifting a long operand. 1

```
DSHF.LF.Q RTA,#⊂123456⊃,#1        ;RTA=247
```

Suppose that a 30-word block of bits MARKERS is to be logically shifted in place three bits to the left. This can **be done as follows.**

```
        MOV RTB, #0                  ;RTB indexes MARKERS from left to right
LOOP:   DSHF.LF ⊂MARKERS⊃(RTB),#3 ;produce one result word
        ISKP.LSS RTB,#29.,LOOP       ; increment RTB and loop if < 29.
        SHF.LF MARKERS+29.,#3        ;do the 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 RTB, #29.                ;RTB indexes MARKERS from right to left
.LOOP:  DSHF.RT ⊂MARKERS⊃(RTB),#3 ;produce one result word
        DSKP.GTR RTB,#0,LOOP         ; decrement RTB and loop if > 0
        SHF. RT MARKERS, #3          ;do the last word in single precision
```

The **same block** of bits can **be** *arithmetical/y* shifted three bits to the right by using the same loop but changing the last SHF.RT instruction to SHFA.RT.

DSHFV

Instruction: **DSHFV . {LF,RT} . {Q,H,S}**
Class: TOP Logical double-width shift reverse

Purpose: ⊂S2||NEXT(S2)⊃ is logically shifted {left,right} by S 1 positions. The {high-order,low-order} 9, 18, or 36 bits of the result (corresponding to Q, H, S respectively) are then stored in DEST. Note that ⊂S2||NEXT(S2)⊃ is *not* treated as a "long" operand, but as two separate operands (which is why the mnemonic is DSHFV and not SHFLV). This is useful for multi-word shifts of any of the three precisions allowed. Long right shifts must start at the right end of the multi-word vector, and long left shifts must start at the left end of the vector. Note that DSHFV.RT by X is equivalent to DSHFV.LF by (9-X), (18-X), (36-X).

Precision: ⊂S2||NEXT(S2)⊃ is considered to be two {Q,H,S}-precision words (rather than one {H,S,D}-precision word) for alignment purposes.

Let X be a DW. Assume RTA contains the negative of the amount by which we wish to shift left. To store the shifted result in RTA the following instruction may be used.

```
DSHFV.RT RTA,#⊂44⊃(RTA),X
```

SHFA

Instruction: **SHFA . {LF,RT} . {Q,H,S,D}**
Class: TOP Shift arithmetically

Purpose: DEST←S1 arithmetically shifted {left,right} by S2. Shifts to the (true) left introduce zero 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. For division by a power of two, rounding towards zero, the QUO2 instruction should be used instead. Note that a left shift by S1 is equivalent ta a right shift by -S1.

Side Effects: INT_OVFL will be set if any bit that is to be shifted into the sign bit does not equal the original sign bit. This may occur when shifting left with S2>0 or by shifting right with S2<0. During untrapped integer-overflow SHFA stores the correct sign followed by the low-order bits of the correct result.

Precision: S2 is a single-word. DEST and Sl have the precision specified by the second modifier.

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
```

SHFAV

Instruction: **SHFAV . {LF,RT} . {Q,H,S,D}**
Class: TOP Shift arithmetically reverse

Purpose: DEST←S2 arithmetically shifted {left,right} by Sl. Shifts to the (true) left introduce zero 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. For division by a power of two, rounding towards zero, the QUO2V instruction should be used instead. Note that a left shift by S1 is equivalent to a right shift by -S1.

Side Effects: INT-OVFL will be set if any bit that is to be shifted into the sign bit does not equal the original sign bit. This may occur when shifting left with S1>0 or by shifting right with S1<0. During untrapped integer-overflow SHFA stores the correct sign followed by the low-order bits of the correct result.

Precision: S 1 is a single-word. DEST and S2 have the precision specified by the second modifier.

The following instruction sets INT-OVFL. 1

```
SHFAV.LF RTA,#7,#3      ;RTA=200
```

ROT

Instruction: **ROT.{LF,RT}.{Q,H,S,D}**
Class: TOP Logical rotate

Purpose: 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. Note that a left rotation by S2 is equivalent to a right rotation by -S2.

Precision: S2 is a single-word. DEST and S1 have the precision specified by the second modifier.

Formal Description:

define ROT. *dir: lfrt.p:qhsd*≡ *TOP* [*p*;*p*;S] *Rotate* (S1, *dir,* s2);

The following illustrates a right rotation by a positive amount.

```
ROT.RT.Q RTA,#1,#1        ;RTA=400
```

ROTV

Instruction: **ROTV . {LF,RT} . {Q,H,S,D}**
Class: TOP Logical rotate *reverse*

Purpose: DEST←S2 rotated {left,right} by S1. Rotation introduces bits shifted out of one end into the other end, so that no bits are lost. Note that a left rotation by S1 is equivalent to a right rotation by -S1.

Precision: S 1 is a single-word. DEST and S2 have the precision specified by the second modifier.

Formal Description:

define ROTV. *dir: lfrt.p:qhsd* ≡ *TOP*[*p*; *S*; *p*] *Rotate* (s2, *dir*, S1);

The following illustrates a left rotation by a negative amount.

```
ROTV.LF.Q RTA,#-1,#3      ;RTA=401
```

5.8 Skip and Jump

Skip and jump instructions allow control to be transferred to locations other than that of the next sequential instruction. Skip instructions are used for short-range transfers, while jumps are used to transfer control anywhere in the 30-bit-address space. In many cases, the skips or jumps occur only if a condition that is specified by a modifier to the instruction is true. Skips or jumps can occur on an *arithmetic condition* (ACOND) which can be any one of the following :

ACOND = {GTR,EQL,GEQ,LSS,NEQ,LEQ}

These correspond to the conditions $>, =, \geq, <, \neq, \leq$ respectively.

Skips may occur on *logical conditions* (LCOND) as well as arithmetic conditions for the SKP instruction. The LCONDs are:

LCOND = {NON,ALL,ANY,NAL}

These correspond to the logical conditions that relate two operands (say OP 1 and OP2) as shown in the table below. Here OP2 is considered to be a mask whose one-bits select bits of OP1 to be tested.

Modifier	Condition	Meaning
NON	(OP1 A OP2) = 0	If no masked bits are 1
ALL	(*one's-complement*(OP1 ∧ OP2)) = 0	If all masked bits are 1
ANY	(OP1 A OP2) ≠ 0	If any masked bit is 1
NAL	(*one's-complement*(OP1 ∧ OP2)) ≠ 0	If not all masked bits are 1

Table 5-5
LCOND modifier descriptions

By combining the ACONDs and the LCONDs, we get the arithmetic and logical conditions (ALCONDs) shown below:

ALCOND = {GTR,EQL,GEQ,LSS,NEQ,LEQ,NON,ALL,ANY,NAL}

All skip instructions are members of the skip instruction class (SOP). See section 4.1.3 for a discussion of this instruction class. The skip instructions are used to perform short jumps in the range -8 . . 7 single-words relative to the current PC (the first word of the instruction that is currently executing). The offset of the jump is specified by the four-bit SKP field of the opcode (OPCODE. SKP). Since OPCODE.SKP fully specifies the jump destination, both OP 1 and OP2 can be used in comparison operations.

All jump instructions are members of the jump instruction class (JOP). See section 4.1.4 for a discussion of this instruction class. The jump instructions are used to transfer control to a general memory location. The low twelve-bits of the instruction specify a JUMPDEST, that is, the location to which control will be transferred if the condition specified in the jump instruction is true. OP 1 specifies a general word that can be tested against the condition specified by the ACOND modifier.

SKP

Instruction: **SKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ,NON,ALL,ANY,NAL} . {Q,H,S,D}**
Class: SOP Skip on condition

Purpose: If OP1 ALCOND OP2 is true (where ALCOND∈{GTR, EQL, GEQ, LSS, NEQ, LEQ, NON, ALL, ANY, NAL)), control is transferred to the specified location that is within -8 . . 7 single-words of the current PC. If ALCOND is false, control is transferred to the next instruction. The number of single-words to skip is specified by OPCODE.SKP.

Precision: The precision of OP 1 and OP2 is specified by the second modifier.

Formal Description:

```
define SKP. rel:alcond. p:qhsd≡ SOP [p; p]   if rel(op1, op2) then Skip fi;
```

The following instructions compute the function "If RTA is Odd Then RTA←3*RTA+1 Fi; RTA←RTA/2;" repeatedly while RTA> 1. Note that FASM determines the SW offset automatically from the JUMPDEST operand.

```
THREEN:
        SKP.LEQ RTA,#1,DONE
        SKP.NON RTA,#1,RTAEVN   ;skip if RTA has an even integer
         MULT RTA, #3           ;multiply by three
         ADD RTA,#1             ;add one - result must be even,
RTAEVN:                         ; so fall into even case
        QUO2 RTA, #1            ;this is better than QUO RTA,#2
        JHPA THREEN
DONE:   ...
```

ISKP

Instruction: **ISKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ}**
Class: SOP Increment, then skip on condition

Purpose: OP 1←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 single-words of the current PC. If ACOND is false, control is transferred to the next instruction. The number of single-words to skip is specified by OPCODE.SKP.

Side Effects: INT_OVFL may be set by the incrementing operation.

Precision: OP1 and OP2 are both single-words.

Formal Description:

```
define ISKP. rel: acond ε   SOP[S,RW;S]Add(op1, 1) → sum, c, ovnext
                                Int_Overflow? next
                                (if rel (sum, op2) then Skip fi also
                                 op1 ⇐ sum also
                                 Carry ← c);
```

The following is a typical loop of the form, "For location I←M Thru N Do . ..". The inner part of the loop must not exceed 8 SWs when assembled.

```
        MOVI,M
LOOP:
        ...
        ISKP.LEQ I,N,LOOP
```

DSKP

Instruction: **DSKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ}**
Class: SOP Decrement, then skip on condition

Purpose: OP 1←OP l-l. 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 single-words of the current PC. If ACOND is false, control is transferred to the next instruction. The number of single-words to skip is specified by OPCODE.SKP.

Side Effects: INT_OVFL may be set by the decrementing operation.

Precision: OP 1 and OP2 are both single-words.

Formal Description:

```
define DSKP. rel: acond = SOP ES, RW; SI   Subtract(op1,1)→ dif, c, ovnext
                                           Int_Overflow? next
                                           (if rel(dif,op2) then Skip fi also
                                            op1 ⇐ dif also
                                            Carry ←c);
```

The following instructions search an array of N SWs 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 single-words follow this entry. In the following, I must be a register. Note that since the loop is one instruction long the SW skip offset is zero. The "-1" added to the base address TABLE compensates for the fact that the address calculation occurs before the decrernentation 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 I, ?<N+1>              ;N is an assembly literal symbol
LOOP:   DSKP.NEQ I,<TABLE-1>(I),LOOP
```

JMP

Instruction: JMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ}

Class: JOP — Jump on condition

Purpose: If OP1 ACOND NEXT(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.

Precision: OP 1 and NEXT(OP 1) are both single-words.

Formal Description:

define JMP.*rel*:*alcond* ≡ *JOP*[*p*,NR] **if** *rel*(op1, *Next* (op1)) **then** *Jump* **fi;**

The following loop searches down a chain of pointers for a specified tail pointer FOOPTR. 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,#cHEAD ↔ FOOPTR>      ;initial ize P and NEXT(P)
LOOP:   MOV P,(P)
        JMP. NEQ P, LOOP
```

JMPZ

Instruction: **JMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D}**
Class: JOP Jump on condition relative to zero

Purpose: If OP1 ACOND NEXT(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.

Precision: OP1 is a single-word.

Formal Description:

```
define JMPZ.rel:acond.p:qhsd = JOP[p] if rel(op1,0) then Jump fi;
```

The following jumps to A WAY iff RTA≤1.0.

```
JMPZ.LEQ #c-1.0>(RTA),AWAY
```

JMPA

Instruction: JMPA
Class: JOP Jump always

Purpose: Jump unconditionally to JUMPDEST. OD1 must be identically zero or a hard trap will occur.

Forma1 Description:

```
define JMPA E JOP[X,U] Jump;
```

The following instruction jumps to **the** RTA-th address stored in the table at JVECTS.

```
JMPA c@JVECTS(RTA)⊃
```

IJMP

Instruction: I JMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ}
Class: JOP Increment, then jump on condition

Purpose: OP I←OP I+ 1. CARRY is not affected. Then if OP I ACOND NEXT(OP 1) 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.

Side Effects: INT_OVFL may be set by the incrementing operation.

Precision: OP 1 and NEXT(OP I) are both single-words.

Formal Description:

```
define IJMP.rel:acond = JOP [p,NRW] Add(op1,1) → s u m , c, o v n e x t
                        Int_Overflow? next
                        (if rel (sum, Next (op1)) then Jump fi also
                         op I ⇐ sum also
                         C a r r y ←c);
```

The following is a typical loop of the form, "For location I←M Thru N Do . . .". The inner part of the loop may be any length when assembled.

```
        MOV.D.D I , [M↔N]          ;M,N a r e assembly l i t e r a l s
LOOP:
        ...
        I JMP.LEQ I,LOOP
```

IJMPZ

Instruction: **IJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ}**
Class: JOP **Increment, then jump on** condition relative to zero

Purpose: OP 1←OP1+1. CARRY is not affected. Then if OPI 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.

Side Effects: **INT_OVFL may** be set **by** the incrementing operation.

Precision: OP 1 is a single-word.

Formal Description:

```
define IJMPZ.rel:acond = JOP[p,RW]   Add(op1,1)→ sum, c, ov next
                                     Int_Overflow? next
                                     (if rel(sum,0) then Jump fi also op1 ⇐ sum also Carry ← c);
```

The following increments N **and** jumps to AWAY if N=0.

```
IJMPZ.EQL N,AWAY
```

IJMPA

Instruction: **IJMPA**
Class: JOP Increment and jump always

Purpose: OP 1←OP1+ 1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Side Effects: **INT_OVFL may be** set by the incrementing operation.

Precision: OP 1 is a single-word.

Formal Description:

```
define IJMPA = JOP [p,RW] Add(op1,1) → sum, c, ov next
                                    Int_Overflow? next
                                    (Jump also op 1 ⇐ sum also Carry ←c);
```

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

Instruction: **DJMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ}**
Class: JOP Decrement, then jump on condition

Purpose: OP1←OP1-1. CARRY is not affected. Then if OP1 ACOND NEXT(OP 1) 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.

Side Effects: INT_OVFL may be set by the decrementing operation.

Precision: OP 1 and NEXT(OP 1) are both single-words.

Formal Description:

```
define DJMP.rel: acond ■ -- JOP [p,NRW]   Subtract (op1,1)→ dif, c, ov next
                                           Int_Overflow? next
                                           (if rel(dif, Next (op1)) then Jump fi also
                                            op1⇐ dif also
                                            Carry ← c);
```

The following is a typical loop of the form, "For location I←M Step -1 Thru 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

Instruction: **DJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ}**
Class: JOP Decrement, then jump on condition relative to zero

Purpose: OP 1←OP 1-1. 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.

Side Effects: INT_OVFL may be set by the decrementing operation.

Precision: OP 1 is a single-word.

Formal Description:

```
define DJMPZ. rel: acond = JOP [p,RW]   Subtract (op1,1)→ dif , c, ov next
                                        Int_Overflow? next
                                        (if rel(dif, 0) then Jump fi also op1⇐ dif also Carry ← c);
```

The following decrements N and jumps to AWAY if N=0.

```
DJMPZ.EQL N,AWAY
```

DJMPA

Instruction: **DJMPA**
Class: JOP Decrement and jump always

Purpose: OP 1←OP I-1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Side Effects: INT_OVFL may be set by the decrementing operation.

Precision: OP 1 is a single-word.

Formal Description:

```
define DJMPA ≡  JOP[p,RW]  Subtract (op1,1)→ dif , c, ov next
                           Int_Overflow? next
                           (Jump also op1← dif also Carry ← c);
```

The following decrements N and jumps to AWAY.

```
DJMPA N,AWAY
```

BNDTRP

Instruction: **BNDTRP . {B,MIN,M1,0,1} . {Q,H,S,D}**
Class: XOP Bounds check and trap on failure

Purpose: Check if OP1 and OP2 satisfy the bounds condition that is specified by the first modifier. If the condition is *not* satisfied then a bounds trap will occur. The following conditions are associated with the first modifier:

Modifier	Meaning
B - [Both]	OP1 ≤ OP2 ≤ NEXT(OP1)
MIN - [MINimum]	MINNUM ≤ OP2 ≤ OP1
M 1 - [Minus 1]	-1 ≤ OP2 ≤ OP1
0 - [Zero]	0 ≤ OP2 ≤ OP1
1 - [One]	1 ≤ OP2 ≤ OP1

Table 5-6
BNDTRP modifiers and meanings

Precision: The precision of OP 1 and OP2 is specified by the second modifier.

The following two equivalent instructions both trap if |RTA|>1.0

```
BNDTRP.B [-1.0 ↔1.0],RTA
BNDTRP.0 #c2.0⊃,#c1.0⊃(RTA)
```

5.9 Routine Linkage

Routine linkage instructions include the instructions to jump to and and return from subroutines and coroutines. Instructions are also provided for returning from traps and interrupts (see Section 6). The subroutine linkage conventions for the S-l are described in a separate document.

The JSR instruction is used to jump to subroutines. OP1 and the PC of the next instruction to be executed (PC-NEXTJNSTR) are pushed into the *JSR* save area (JSR_SAVE_AREA) on the stack. It's format is shown in Figure 5-4. Control is then passed to the routine at the address specified by JUMPDEST. See Section 4.1.4 for a description of how JUMPDEST is computed. Return from a subroutine is accomplished using the RETSR instruction. The stack is decremented so that the old OP1 value that was previously saved in the stack and the return address are now popped off and saved in OP1 and PCNEXTJNSTR respectively.

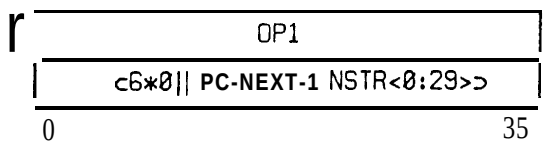

Figure 5-4
JSR Save Area Format

The JCR instruction is used to jump between coroutines. It allows easy transfer of control between two routines by using OP1, OP2 and NEXT(OP2) to transfer information. NEXT(OP2) contains the return address to the coroutine that is not currently executing. No locations on the stack are involved.

There are three return instructions that are used for returning from traps and interrupts. They restore different amounts of information including status words and the return PC. RET is used to return from instructions such as TRPSLF which do not save either PROC-STATUS or USER--STATUS in the save area. RETUS does a return and restores USER-STATUS. This is used for returning from soft-errors (see Section 6.1). RETFS does a return and restores full status, that is, both PROCSTATUS and USER-STATUS are loaded from the save area. Note that the return. address is the first single-word from the end (highest memory location) of all save areas. PROCSTATUS (if present) is the second single-word, while USER-STATUS (if present) is the third single-word from the end of the save area. The formats of the save areas for traps and interrupts are shown in Figures 6-3 to 6-7. Note that the RETFS restores the entire PROCSTATUS word from the save area rather than loading partial processor status (as described in section 2.5.1).

There are two instructions that are used to force the processor to execute trap sequences under program control: TRPSLF and TRPEXE. TRPSLF can be used by either the executive or the user to cause a trap to one of the TRPSLF_VECS that exist in the same address space as the

instruction executing the TRPSLF instruction. TRPEXE can be used by either the executive or the user to cause a trap to the executive. The vectors for TRPEXE start at location TRPEXEVECS in the executive's address space.

The TRPSLF and TRPEXE instructions both deliver parameters to their respective trap handlers by passing information in the form of two double-word *trap parameter operands* (TRP_PARM_OP{1,2}[0..1]. See Figure 6-6). The interpretation of these operands depends on the value of the *trap parameter descriptor single-word* (TRAP_PARM_DESC_SW) which is located in the trap vector for both TRPSLF and TRPEXE (see Figure 6-2).

The TRP_PARM_DESC_SW forms an extension to the opcode by describing ways in which the trap parameter operands can be interpreted. It is a single-word consisting of the four quarter-words labeled QW[0..3] respectively. QW[0] and QW[1] must be identically zero. QW[2] describes how OP2 of the trapping instruction will be passed into the trap routine in the double-word TRP_PARM_OP1[0:1]. QW[3] describes how OP1 of the trapping instruction will be passed into the trap routine in TRP_PARM_OP1[0:1]. QW[2] and QW[3] have identical format and interpretation, They are interpreted as TMODE-fields (as described below).

The tables below show how the trap parameter operands are interpreted based on the value of TMODE. Table 5-7 lists the primary uses for the different values of TMODE. Table 5-8 shows how the contents of TRP_PARM_OP{1,2}[0:1] are interpreted depending on the value of TMODE. This table also shows the cases that cause an error trap occurs when interpreting TMODE. The left or right arrows represent left or right justification with zero-filling respectively.

TMODE	Primary Use
0	Check an unused operand descriptor.
1	Deliver a PC-relative jump descriptor.
2	Deliver the entire operand descriptor.
3	Deliver a pointer operand (cannot be an immediate).
4	Deliver a quarter-word value operand.
5	Deliver a half-word value operand.
6	Deliver a single-word value operand.
7	Deliver a double-word value operand.

Table 5-7
TMODE Values and their Uses

TMODE	Trap Condition	TRP_PARM_OP{1,2}[0]	TRP_PARM_OP{1,2}[1]
<0	always	---	---
0	OD{1,2} ≠ 0	undefined	undefined
	never	→OD{1,2}	undefined
2	never	→OD{1,2}	extended word for OD{1,2}*
3	IMMED(OP{1,2})	ADDRESS(OP{1,2})**	undefined
4	never	QW ←OP{1,2}***	undefined
5	HW alignment	HW ←OP{1,2}***	undefined
6	SW alignment	SW OP{1,2}	undefined
7	SW alignment	OP{1,2}<0:35>	OP{1,2}<36:71>***
>7	always	---	---

* If TMODE=2, then the extended word for OD{1,2} is stored in TRP_PARM_OP{1,2}[1] if the extended-word exists, otherwise TRP_PARM_OP{1,2}[1] is undefined.

** If TMODE=3, TRPEXE stores ADDRESS(OP{1,2}) with P-bit- 1.

x x x If TMODE= 4 . . 7 , immediates are properly sign-extended and justified according to the **value** of OD.F.

Table 5-8
Interpretation of TMODE

The RET instruction is used for returning from TRPSLF instructions since it pops OP1 parameters off the stack **in** addition to the return PC. RETFS is used to return from TRPEXE instructions since it restores the status words in addition to popping the PC and the parameters.

JSR

Instruction: **JSR**
Class: JOP Jump to subroutine

Purpose: The return address and OP 1 are pushed onto the stack and SP is adjusted accordingly. The format of the JSR save area is shown in Figure 5-4. Control is then transferred to JUMPDEST. If this instruction would cause **SP>SL,** a hard trap will occur and the stack will not be affected. (The RETSR instruction is normally used to return from a subroutine called by JSR.)

Precision: A II operands involved are single-words.

Side Effects: **SP←SP+8**

The following pushes **ADDRESS(FOO)** and RTA on the stack before jumping to BAZ.

```
        JSR RTA,BAZ
FOO:    ...             ; return address
```

JCR

Instruction: **JCR**
Class: XOP Jump to coroutine

Purpose: OP1 and OP2 are exchanged. NEXT(OP2) is prefetched and stored temporarily. The PC_NEXT_INSTR of the routine that executed the JCR instruction is saved in NEXT(OP2). The value NEXT(OP2) that was prefetched is then loaded into PC and control passes to the coroutine.

Precision: A 11 operands involved are single-words.

Suppose that each of two coroutines has an associated stack. Let there be a double-word "save area" SAVE.AREA which contains the stack pointer and program counter for the currently inactive coroutine. Whichever coroutine is actually running uses register SP as its stack pointer, and of course uses PC **as** its program counter. Then the following instruction makes the current coroutine inactive, and -activates the other coroutine after setting up its stack pointer and saving the current one.

```
JCR SP, SAVE. AREA          ;call other coroutine
```

ALLOC

Instruction: **ALLOC.{1..32}**
Class: XOP Allocate stack locations

Purpose: This instruction is commonly used to save registers on the stack. It causes 1 . . 32 single-words starting at ADDRESS(OP1) to be moved into the memory locations starting at SP. OP2 is added to the value of SP, producing a new value for SP (OP2 is therefore a number of quarter-words, not a number of single-words). OP2 should be at least as large as four times the modifier, but this may not be checked for by the hardware. If this instruction would cause **SP>SL,** a hard trap will occur and the stack will not be affected. If the source and destination overlap, the result is undefined.

Side Effects: **SP←SP+OP2**

Precision: All operands involved are single-words.

The following saves all the registers and reserves an additional DW on the stack as well.

```
ALLOC.32 %0,?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 %0,?4*<32.+2>
```

RETSR

Instruction: **RETSR**
Class: XOP Return from subroutine

Purpose: Return from a subroutine that was invoked by the JSR instruction. The stack pointed to by OP2 (usually SP) is decremented by eight, removing the saved OP1 value and the return address. OP1 is then loaded with this old OP1 value, and control is transferred to the location specified by the return address (See Section 5.9 for a description of the JSR instruction and the JSR save area).

Side Effects: SP←ADDRESS(OP2)-8

Precision: A 11 operands involved are single-words.

Formal Description:

```
define RETSR = XOP ES; S, NR]  Check_Jump-Address (Next (op2)<6:35>) next
                               (Sp ← Address(op2) also
                                op1 ⇐ op2 also
                                pc-nxt-instr ← Next(op2)<6:33>);
```

The following code calls BAZ, which returns to FOO, saving and restoring RTA on the stack. Assume SP is the stack pointer.

```
        JSR RTA, BAZ
FOO:    ...             ; return here

BAZ:    ...             ; called routine
        RETSR RTA, (SP)
```

RET

Instruction: **RET**
Class: XOP Return and pop parameters

Purpose: Return from an exception without restoring registers. Note that OP 1=1 for a return from **TRPSLF. OP1+** 1 single-words (OP1 parameters + return address) are popped off the stack pointed to by **ADDRESS(OP2)** (usually SP), and the stack is adjusted. All popped words except the return address are thrown away and ignored. Control is then transferred to the location specified by the return address.

Side Effects: ADDRESS(OP2)←ADDRESS(OP2)-4-OP1*4

Precision: All operands involved are single-words.

Formal Description:

```
define RET = XOP IS, R; S, R] Check_Jump_Address(op2<6:35>) next
                              (Sp ← Address(op2) - shift(op1,2) also
                               pc-nxt-instr ←op2<6:33>);
```

The following returns from a previous JSR call, throwing **away** the operand previously pushed on the stack by the JSR.

```
RET #1,(SP)
```

RETUS

Instruction: **RETUS**
Class: **XOP** Return, restoring user status

Purpose: Return from an exception that requires **USER-STATUS** to be restored (e.g., soft traps). OP 1+2 single-words (OP 1 parameters + old **USER-STATUS** + return address) are popped off the stack pointed to by ADDRESS(OP2), and the SP is adjusted. **USER-STATUS** is loaded from the value in the stack. All other popped words except the return address are thrown away and ignored. Control is then transferred to the location specified by the return address.

Side Effects: SP←ADDRESS(OP2)-8-OP1*4

Precision: A ll operands involved are single-words.

The following returns from a soft trap (The soft-trap save area **is** shown in Figure 6-4).

```
RETUS #11, (SP)
```

RETFS

Instruction: **RETFS**
Class: XOP Return, restoring full status

Purpose: Return from an exception that requires' both USERSTATUS and PROCSTATUS to be restored (i.e., hard traps, TRPEXE and interrupts. See Section 6.6 for a description of the save areas associated with each of these). OP1 + 3 single-words (OP1 parameters + USER-STATUS + PROCSTATUS t return address) are popped off the stack, and the SP is adjusted. USER-STATUS is loaded from the value saved in the stack. The entire PROCSTATUS word is loaded from the value saved in the stack (as opposed to loading partial processor status; see Section 2.5.1 for a description of partial processor status). All other popped words except the return address are thrown away and ignored. Control is then transferred to the location specified by the return address.

Restrictions: Illegal in user mode.

Side Effects: SP←ADDRESS(OP2)-12-OP1*4

Precision: A ll operands involved are single-words.

The following returns from an interrupt.

```
RETFS #1,(SP)
```

TRPSLF

Instruction: **TRPSLF .{ 0 . .** 63 **}**
Class: XOP Trap to self

Purpose: Causes a trap to a routine in the current address space. The trap vectors start at location TRPSLF_VECS in the current address space. A particular vector in this block is selected by the modifier. The trap vector specifies a handler address and a TRP_PARM_DESC_SW. The save area contains two double-word trap operands, PC, and PC_NEXT_INSTR. The interpretation of the operands is based on the TMODE fields in TRP_PARM_DESC_SW. See Section 5.9 for a complete discussion of these fields and how they are interpreted.

The following causes a trap to the "number 0" trap routine in the current address space with operands X and Y.

```
TRPSLF, 0X, Y
```

TRPEXE

Instruction: **TRPEXE . { 0 .. 63 }**
Class: XOP Trap to executive

Purpose: Causes **a trap** to a routine in the executive's address space. The trap vectors start at location **TRPEXE_VECS in** the executive's address space. A particular vector in this block is selected by the modifier. The trap vector specifies a handler address, **a TRP-PARMDESCSW, USERSTATUS** and PROCSTATUS. The save area contains two double-word trap operands, PC, the old USER-STATUS and PROCSTATUS, and PCNEXTJNSTR. The interpretation of the operands is based on the **TMODE** fields in TRP-PARMDESCSW. See Section 5.9 for a complete discussion of the uses of TRPEXE.

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
```

5.10 Stack

A stack is specified by any two consecutive single-words in memory (or in registers). The S-l interprets these locations as a *stack-painter* and a *stack-limit*. The meaning of these terms differs slightly whether we are talking about *upward-growing stacks* or *downward-growing stacks.* The interpretation of which of these two single-words is the stack-pointer and which is the stack-limit depends on whether we are talking about upward-growing stacks or downward-growing stacks. In the description of stacks that follows, note that an upward-growing stack and a downward-growing stack can exist together in memory at the same time. In this case, the same register is used for the SP of the upward-growing stack as is used for the SL of the downward-growing stack (and vice-versa!).

Upward-growing stacks grow towards higher memory locations. Instructions that operate on upward-growing stacks use the "UP" modifier with the stack instruction. For upward-growing stacks, OP is the stack-pointer and NEXT(OP) **is** the stack-limit. The stack-pointer points to the *next free location* on the stack. Thus, a push onto an upward-growing stack involves saving the value in the location specified by the stack-pointer and then incrementing the stack pointer. The stack-limit for an upward-growing stack is the location *immediately following* the stack-pointer (i.e., stack-limit=NEXT(stack-pointer)). It points to the first location beyond the end of the stack.

Downward-growing stacks grow towards lower memory locations. Instructions that operate on downward-growing stacks use the "DN" modifier with the stack instruction. For downward-growing stacks, OP is the stack-limit and NEXT(OP) is the stack-pointer. The stack-pointer points to the *top item* on the stack. Thus, a push onto a downward-growing stack involves incrementing the stack pointer and then saving the operand in this location. The stack-limit for an upward-growing stack is the location immediately preceding the stack-pointer. It points to the last stack location into which information can be stored.

The SP_ID field of USER-STATUS specifies a *particular* upward-growing stack for implicit use by certain instructions such as JSR and **ALLOC;** the SP and SL for this stack must be in registers. By contrast, the instructions in this section can operate on any arbitrary stack specified by an explicit operand,

ADJSP

Instruction: **ADJSP .{UP,DN}**
Class: XOP Ad just (arbitrary) stack pointer

Purpose: Adjust the size of an {upward-growing, downward-growing] stack. OP2 is the a single-word two's-complement number which is (added to, subtracted from] OP1 for ADJSP.{UP,DN}. Thus, ADJSP with a positive OP2 makes **a** stack larger while AD JSP with **a** negative OP2 makes a stack smaller.

Side Effects: If OP1+OP2>NEXT(OP1) for ADJSP.UP or NEXT(OP1)-OP2<OP **1** for ADJSP.DN, a hard trap will occur.

Precision: Both OP 1 and OP2 are single-words.

The following throws **away** the **top** 4 stack elements. Let SPL be the address of **a** stack pointer/limit DW.

```
ADJSP. UP SPL, #-4
```

PUSH

Instruction: **PUSH . {UP,DN} . {Q,H,S,D}**
Class: XOP Push onto (arbitrary) stack

Purpose: Push OP2 with precision specified by the second modifier onto an upward-growing or downward-growing stack.

Side Effects: If OP 1+{1,2,4,8}>NEXT(OP 1) for PUSHUP or NEXT(OP1)-{1,2,4,8}<OP1 for PUSH.DN, a hard trap will occur.

Precision: Both OP1 and OP2 are single-words.

The following **pushes** RTA **on a** stack. Let SPL **be the** address of a stack pointer/limit DW.

```
PUSH.UP SPL,RTA
```

POP

Instruction: **POP .{UP,DN} . {Q,H,S,D}**
Class: **XOP** Pop from (arbitrary) stack

Purpose: Pop OP2 with precision specified by the second modifier off of an upward-growing or downward-growing stack.

Precision: Both OP 1 and OP2 are single-words.

The following pops the top **value on a** stack into RTA. Let SPL be the address of **a** stack pointer/limit DW.

```
POP.UP  SPL,RTA
```

5.11 Byte

The **byte** data types (single-word and double-word) are described in Section 3.5. Byte pointers and byte selectors are described in Section 3.6. Byte instructions access bytes via byte **pointers.**

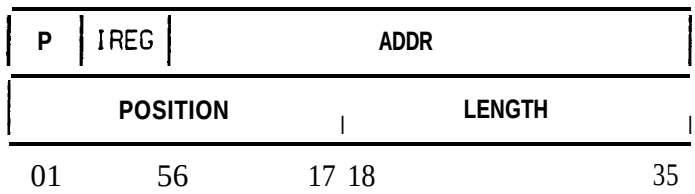

Figure 5-5
Byte Pointer

The instruction modifier {S,D} specifies the byte precision that the instruction works with (S=single-word **byte, D=double-word byte).** Let MBL be the *maximum byte length* for for a given precision **byte.** Single-word **bytes have** MBL=36. Double-word **bytes have** MBL=72. Any **byte** instruction will hard-trap if POSITION+LENGTH> MBL. Furthermore, the IAP **must** point to the beginning of a single-word or the instructions will hard-trap. This restriction **on the** IAP and the rule **concerning** MBL implies **that** single-word bytes may not cross single-word boundaries.

There are three immediate instructions which use **only** a **byte** selector (a **<position,,length> single-word)** to **access an** *immediate byte.*

LBYT

Instruction: **LBYT . {S,D}**
Class: XOP

Load (unsigned) byte

Purpose: OP2 is the (source) byte pointer. OP1 is the destination word which receives the *zero-extended* byte. POSITION+LENGTH>MBL causes a hard trap.

Precision: OP1 has the same precision as the modifier. OP2 is a byte pointer. OP2 points to a byte with **a** precision specified by the modifier.

The following sets RTA to the exponent field of the single-word floating-point number X.

```
LBYT RTA, [X+1,,11
```

LIBYT

Instruction: **LIBYT . {S,D}**
Class: TOP Load immediate (unsigned) **byte**

Purpose: S2 is the (source) byte selector. **S1** contains the (source) immediate byte. DEST receives the zero-extended byte.

Precision: S I and DEST **have** the same precision as the modifier. S2 is a byte selector. The **byte** contained in **S1** has the **same** precision as the modifier.

The **following** sets RTA to the exponent field of the single-word floating-point **number** X.

```
LIBYT RTA,X,#c1,,11>
```

LSBYT

Instruction: **LSBYT . {S,D}**
Class: XOP Load signed byte

Purpose: OP2 is the (source) byte pointer. OP1 is the destination word which receives the *sign-extended* byte. POSITION+LENGTH>MBL causes **a** hard trap.

Precision: OP1 has the **same** precision **as the** modifier . OP2 is **a** byte-pointer. OP2 points to **a byte** with **a precision specified by the modifier.**

The following sets RTA to the signed **value** of **the sign and exponent** fields of the single-word floating-point **number X.**

```
LSBYT RTA, [X+ 121
```

LISBYT

Instruction: **LISBYT.{S,D}**
Class: TOP Load immediate signed byte

Purpose: S2 is the (source) byte selector. S1 contains the (source) immediate byte. DEST receives the *sign-extended* byte.

Precision: S 1 and DEST have the same precision as the modifier. S2 is a byte selector. The byte contained in S 1 has the same precision as the modifier.

The following sets RTA to the signed value of the sign and exponent fields of the single-word floating-point number X. Notice that a short constant can be used, because the position of the byte is zero.

```
LISBYT RTA,X,#12
```

DBYT

Instruction: **DBYT . {S,D}**
Class: XOP Deposit byte

Purpose: OP2 contains, as its low-order bits, the byte to be stored. OP1 is the byte pointer that locates the byte to be replaced.

Precision: OP1 is a **byte** pointer. It points to a byte with the **same** precision as the modifer. OP2 has the **same** precision as the modifier.

The following sets the mantissa of the single-word floating-point number X to **the** twenty-six low order bits of RTA.

```
DBYT IX ↔ 12,,321,RTA
```

DIBYT

Instruction: **DIBYT . {S,D}**
Class: TOP Deposit immediate byte

Purpose: DEST is the destination word for the immediate byte. S1 contains, as its low order bits, the byte to be stored. S2 is the byte selector that controls the placement of the byte in DEST.

Precision: S 1 and DEST have the same precision as the modifier. S2 is a byte selector.

The following sets the exponent field of the single-word floating-point number in RTA to zero.

```
DIBYT RTA,#0,#c1,,11⊃
```

ADJBP

Instruction: **ADJBP . {S,D}**
Class: TOP

Adjust byte pointer

Purpose: Sl is the source byte pointer. S2 specifies the number of bytes to adjust S 1 by. DEST receives S 1 adjusted by the number of bytes specified by S2. In more detail, if S1.LENGTH=0 then S 1 is copied into DEST. Otherwise, DEST becomes S 1 adjusted forward or backwards by S2. If S2 is positive, the byte pointer is advanced. If S2 is negative, the byte pointer is backed up. S2=0 causes Sl to be copied into DEST. The adjustment assumes that single-word bytes are contained in single-words and double-word bytes are contained in double-words (i.e., POSITION+LENGTH≤MBL). The adjustment will not cause DEST.ADDR to overflow into DEST.IREG. Instead, the adjustment is done modulo 2^{30} {no hard trap occurs on wrap-around).

Precision: Sl and DEST are byte pointers and the bytes they specify have precision equal to the modifier- S2 is a single-word.

The following advances the byte pointer at BP by one byte.

```
ADJBP BP, #1
```

Suppose that TABLE is a vector of NBYTES four-bit bytes, packed nine per single-word. Suppose that a purported index into this table is in RTB. This code checks the purported index for validity and then produces the desired **byte** in RTA, or zero if the index was invalid. It produces a flag indicating whether the index is valid, and then selects one of two **byte** pointers to adjust. If the index is valid, a byte pointer to the beginning of the table is adjusted to point to the desired byte; if not, a byte pointer to a zero-length **byte** is produced. Loading a byte using a zero-length byte pointer always produces a zero. Note the "↑3" in the ADJBP instruction: it causes the indexing by RTA to be double-word indexing, because byte pointers are two words long.

```
        BNDSF.O RTA,#⊂NBYTES-1⊃,RTB      ;RTA←-1 if index okay, else 0
        ADJBP RTA,⊂BPTRS+10⊃(RTA)↑3,RTB;get ptr to desired byte, or nul l ptr
        LBYT RTA,RTA                     ; load byte into RTA
        ...

BPTRS:  TABLE↔0,,4               ;byte pointer to beginning of TABLE
        TABLE↔0,,0               ;zero-length byte pointer
```

5.12 Bit

Bit instructions operate on the boolean data type. These instructions are concerned with individual bits and their ordering. BITRV and BITRVV reverse the order of the low-order bits of a word. BITEX and BITEXV extract bits from a word, according to a mask, and then *squeeze* them to the right of the destination. This is useful for extracting a set of flags in order to do an N-way branch on them. BITCNT counts the number of one-bits in a word. This was designed for counting the number of elements in a PASCAL set. **BITFST** gives the position of the first (left-most) one bit in a word. This is useful for computing the index of the first element of a PASCAL set.

BITRV

Instruction: **BITRV.{Q,H,S,D}**
Class: TOP Bit reverse

Purpose: Reverse the order of the S2 low-order bits of Sl, and zero-extend the result into **DEST.**

Precision: S 1 and DEST have the same precision as the modifier. S2 is a single-word.

Formal Description:

```
define BITRV. p: qhsd ≡   TOP[p;p;S]   if (s2 < 0) v (s2 > Bits(p) )
                                         then Hard-Error
                                         else dest ⇐ Reverse_Bits(S1,s2)
                                       fi;
```

The following reverses all nine bits of its operand.

```
BITRV.Q RTA,#c123ↄ,#11 ;RTA=624
```

BITRVV

Instruction: **BITRVV . {Q,H,S,D}**
Class: TOP Bit reverse reverse

Purpose: Reverse the order of the S1 low-order bits of S2, and zero-extend the result into DEST.

Precision: S2 and DEST have the same precision as the modifier. S 1 is a single-word.

Formal Description:

```
define BITRVV.p:qhsd = TOP [p;p;S]   if (S1 < 0) v (S1 > Bits (p))
                                        then Hard-Error
                                        else dest ⇐ Reverse_Bits (s2, S 1)
                                     fi;
```

The following reverses all nine bits in the operand.

```
BITRVV RTA,#11,#c624>   ;RTA=123
```

BITEX

Instruction: **BITEX . {Q,H,S,D}**
Class: TOP

Bit extract

Purpose: Extract the bits of S1 selected by the one-bits of S2. Squeeze these selected bits to the right and zero-extended into DEST.

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

Formal Description:

```
define BITEX, p:qhsd =    TOP[p;p;p]  dest ⇐ Extract-Bits (S1,s2);
```

The following extracts alternate bits from the operand. 1

```
BITEX.QRTA,#c765ↄ,#c525ↄ        ;RTA=37
```

This code does an eight-way dispatch based on CARRY, INT_Z_DIV_MODE, and FLAGS<0> in USER-STATUS.

```
        RUS RTA                         ;read USER-STATUS into RTA
        BITEX RTA,#c010000,,400010ↄ     ;select bits
        JMPA c@DISPTABLEↄ(RTA)↑2        ;dispatch through table of IAPs
DISPTABLE:
        NDNEDFTHEM          ;to this address if no bits were set
        FLAG                ;to this address if only FLAG<0> set
        ZDIV                ;and so on...
        ZDIVFLAG
        CARRY
        CARRYFLAG
        CARRYZDIV
        CARRYZDIVFLAG
```

BITEXV

Instruction: BITEXV **. {Q,H,S,D}**
Class: TOP Bit extract reverse

Purpose: Extract **the bits** of S2 selected by **the one-bits** of **S1. Squeeze these** selected bits to the right and zero-extended into DEST.

Precision: S 1, **S2,** and DEST all **have the** precision specified **by the** modifier.

Formal Description:

define BITEXV. *p*:*qhsd* ≡ *TOP*[*p*;*p*;*p*] **dest** ← *Extract-Bits* (s2, S1);

The following extracts a group of seven bits from the operand.

```
BITEXV.Q RTA,#c765ↄ,#c525ↄ        ; RTA-127
```

BITCNT

Instruction: **BITCNT . {Q,H,S,D}**
Class: XOP Bit count

Purpose: OP1←*number of one bits in* OP2

Precision: OP 1 is a single-word. OP2 has the same precision as the modifier.

Formal Description:

define BITCNT.*p*:*qhsd* ≡ *XOP* [S;*p*] op1 ← *Number_of_1_Bits* (op2);

The following sets RTA (flag-style) if RTA has odd parity.

```
        BITCNT RTA,RTA
        CMPSF.ALL RTA,#1
```

The parity of an arbitrarily long block of bits can be obtained by using the XOR instruction to condense the block. (The XOR operation essentially causes pairs of one-bits to cancel.) If TABLE is a block of N single-words (N>2), this code sets RTA (flag-style) if TABLE has odd parity.

```
        XOR RTA,<TABLE+N-1>,<TABLE+N-2>;RTA gets XOR of two words
        MOV RTB,#<N-3>                  ;RTB counts all other words
LOOP:   XOR RTA,<TABLE>(RTB)            ;XOR in next uord
        DSKP.GEQ RTB,#0,LOOP            ;loop until all words done
        BITCNT RTA,RTB                  ;count result as before
        CMPSF.ALL RTA,#1
```

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 single-word integer, this code jumps to TWOPOWER if HUNOZ is an exact power of two.

```
        BITCNT 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. Alternatively, a trick involving the NEG instruction can be used instead.

BITFST

Instruction: **BITFST . {Q,H,S,D}**
Class: XOP

Bit number of first one bit

Purpose: If OP2=0 then OP1←-1 *else* OP1←*bit number of the leftmost one bit in* OP2

Precision: OP 1 is a single-word. OP2 has the same precision as the modifier.

Formal Description:

define BITFST.*p*:*qhsd* = *XOP*[S;*p*] op1 ← *Number_of_First_1_Bit*(op2);

The following sets RTA to floor(log2(RTA)) with RTA assumed to be a non-zero unsigned single-word integer.

```
BITFST RTA,RTA
SUBV RTA,#c43>
```

Suppose that location MASK contains a non-zero single-word. This piece of code constructs a byte pointer in (double-word) RTA to the smallest byte containing all the one-bits in WUNOZ.

```
BITFST RTA,HUNOZ          ;number of leading zero bits
BITRV RTA1,HUNOZ,#c36.>   ;reverse HUNOZ into RTA1
BITFST RTA1               ;number of trailing zero bits
ADD RTA1,RTA              ;number of surrounding zero bits
SUBV RTA1,#c36.>          ;length of smallest containing byte
MOV.H.D RTA1,RTA          ;put position in high halfword of RTA1
MOVADR RTA,HUNOZ          ;make IAP to HUNOZ in RTA
```

5.13 Block

Blocks are discussed in Sections 3.7. The instructions in this section are used for comparing, moving, and initializing blocks. Block I/O instructions are described in Section 5.17.

STRCMP is used to compare two blocks (or strings). BLKINI initializes a block to a given scalar value. BLKMOV copies one block to another location. BLKID does a BLKMOV, but transfer a block from and INSTRUCTION page to a DATA page. This allows instructions to be accessed as data. BLKDI transfers from a DATA page to an INSTRUCTION page, allowing data to be executed as instructions. See Section 2.3.2 for a discussion of INSTRUCTION and DATA pages.

STRCMP

Instruction: **STRCMP.{RTA,RTB}**
Class: XOP String compare

Purpose: Consider the two blocks OP1 and OP2 to be strings of quarter-word characters. The blocks have the same length. {RTA,RTB} contains the block length in quarter-words. Signed comparison is used, and each quarter-word character is compared separately. The result of the comparison is computed as shown in the following table and is stored back 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 bit 0 being set does not affect indexing.

Caution: This instruction **may cause** a non-zero **value** to be stored in INSTRUCTION-STATE.

Condition	Result
STRING1 = STRING2	0
STRING1 > STRING2	n
STRING1 < STRING2	$-2^{35}+n$ (i.e MINNUM+n)

where n **is** the position of the first character to differ

Table 5-9
STRCMP Results

-Precision: OP1 and OP2 are blocks. The elements of the blocks are quarter-words. RTA **and RTB are single words.**

The following sets RTA to the result of comparing the eighty-character blocks at X and Y.

```
        MOV RTA,?120            ;120 octal = 80 decimal
        STRCMP.RTA X,Y
```

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" ail 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 RTA,XLENGTH,YLENGTH          ;set RTA to minimum real length
        INC RTB,RTA                      ; save one greater in RTB for unequal case
        STRCMP.RTA X,Y                   ;do comparison
        JMPZ.NEQ RTA,DONE                ;difference found
          SKP.EQL XLENGTH,YLENGTH,DONE   ;done if strings are equal length
          MOV RTA,RTB                    ;RTB is index of "imaginary" character
          SKP.LEQ XLENGTH,YLENGTH,DONE   ;set high-order bit if necessary
            OR RTA,#c400000,,0>          ;or DIBYT RTA,#1,#1 to save a word!
DONE:   ...                              ;RTA contains result
```

BLKMOV

Instruction: **BLKMOV . {RTA,RTB}**
Class: X O P Block **move**

Purpose: OP2 is the source block. OP1 is **the** destination block. {RTA,RTB} specifies which register contains **the** quarter-word transfer length.

The semantics of the BLKMOV instruction are such that if the source and destination blocks overlap, no word in the source block is overwritten until after it has been transferred to **the** destination block.

Caution: This instruction **may cause** a non-zero value to **be** stored in INSTRUCTION-STATE.

Precision: OP 1 and OP2 are blocks. The elements of the block have quarter-word precision. RTA and RTB are single-words.

The following **moves all** registers into **an** area starting at RECS. The original contents of RTA must be saved temporarily in SAVRTA since RTA is used to contain **the** quarter-word transfer **length.**

```
SLR, 4 SAVRTA, ?4*40        ;save RTA and load with transfer length
BLKMOV, RTA REGS, %0        ;do block transfer
MOV REGS+4*RTA, SAVRTA      ; f ix up saved RTA
```

BLKINI

Instruction: **BLKINI . {RTA,RTB} . {Q,H,S,D}**
Class: XOP Block initialize

Purpose: OP2 is the scalar initialization value. OP1 is the block to be initialized. {RTA,RTB} specifies the register containing the number of *quarter-words* to be initialized.

Caution: This instruction may cause a non-zero value to be stored in INSTRUCTION-STATE.

Precision: OP1 is a block. OP2 has the same precision as the second modifier. The elements of the block also have the same precision as the second modifier. A hard trap will occur if the contents of {RTA,RTB} is not a multiple of the block-element precision. RTA and RTB are single-words.

The following zeros registers 8 through 31.

```
MOV RTA, ?4*30          ;set RTA to number of QWs
BLKINI.RTA %8,#0        ;initialize block
```

BLKID

Instruction: **BLKID . {RTA,RTB}**
Class: XOP Block transfer instructions to data

Purpose: OP2 is the source block. OP1 is the destination block. {RTA,RTB} specifies which register contains the quarter-word transfer length. The source block must be on a page(s) marked with **INSTRUCTION4** The destination block must be on **a page(s)** marked with DATA= 1.

Caution: This instruction may cause a non-zero value to be stored in **INSTRUCTION-STATE.**

Precision: OP1 and OP2 are blocks. The elements of the block have quarter-word precision. **RTA** and RTB **are** single-words.

The following transfers a single word instruction at INST into RTA.

```
NOV RTA,?4        ;load RTA uith QW transfer length
BLKIO.RTA, INST   ;load RTA uith instruction
```

BLKDI

Instruction: BLKDI . **{RTA,RTB}**
Class: XOP Block transfer data **to** instructions

Purpose: OP2 is the source block. **OP1** is the destination block. {RTA,RTB} specifies which register contains the quarter-word transfer length, The source block must be on a page(s) marked with DATA-l. The destination block **must** be on a page(s) marked with INSTRUCTION4

Caution: This instruction may cause a non-zero value **to** be stored in INSTRUCTION-STATE.

Precision: **OP1** and **OP2** are blocks. The elements of the block have quarter-word precision. RTA and RTB are single-words.

The following transfers a DW **value** in RTA to a two word instruction **at** INST.

```
MOV RTB, ?10          ;set RTB to QW transfer length
BLKDI.RTB INST,RTA    ;move RTA to instruction space
```

5.14 Status

Status instructions are used to manipulate the USER-STATUS and PROC_STATUS words. Instructions exist for reading, writing, and jumping based on *logical conditions* (LCONDs). The LCONDs are described in Section 5.6. See Section 2.5 for a description of the status words.

RUS

Instruction: **RUS**
Class: XOP

Read user **status**

Purpose: OP 1←USER_STATUS. OP2 is unused.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets RTA to USER-STATUS. Note **that** FASM supplies **the** zero operand.

```
RUS RTA
```

JUS

Instruction: **JUS . {NON,ALL,ANY,NAL}**
Class: JOP Jump on selected user status bits

Purpose: If OP 1 LCOND USER-STATUS (where LCOND∈{NON,ALL,ANY,NAL}) is true, control is transferred to the location specified by JUMPDEST.

Precision: All operands concerned are single-words.

Let ERRORS be a mask for several bits in USER-STATUS. The following jumps to ZIP if any of these bits are set.

```
JUS  ERROR&ZIP
```

JUSCLR

Instruction: JUSCLR . {NON,ALL,ANY,NAL}
Class: JOP Jump on selected user status bits and clear

Purpose: TEMP←USER_STATUS. USER-STATUS is then loaded according to USER_SATUS←USER_STATUS∧*one's-complement*(OP1). If OP1 LCOND TEMP (where LCOND∈{NON,ALL,ANY,NAL}) is true, control is transferred to the location specified by JUMPDEST. Note that a hard trap will occur if clearing the specified bits would produce an illegal value for USER-STATUS.

Precision: All operands concerned are single-words.

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 ZD IV, YDW
```

WUSJMP

Instruction: WUSJMP
Class: JOP Write user status and jump

Purpose: USER_STATUS←OP1. Control is then transferred to the location specified by JUMPDEST. Note that a hard trap will occur if an illegal value of USER-STATUS is specified.

Precision: All operands concerned are single-words.

The following sets the USER-STATUS to NEWUS and jumps to AWAY.

```
WUSJMP NEWUS,AWAY
```

SETUS

Instruction: **SETUS**
Class: XOP Set specified user status bits

Purpose: USER_STATUS←USER_STATUS∨OP1. OP2 is unused. Note that a hard trap will occur if an illegal value of USER-STATUS is specified.

Precision: OP1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets the low order bit in USER-STATUS.

```
SETUS #1
```

CLRUS

Instruction: **CLRUS**
Class: XOP Clear specified user status bits

Purpose: USER_STATUS←USER_STATUS∧*one's-complement*(OP 1). OP2 is unused. Note that a hard trap will occur if an illegal value of USER-STATUS is specified. The JUSCLR instruction can clear specified user status bits and simultaneously test them.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following clears the low order bit in USER-STATUS.

```
CLRUS #1
```

RSPID

Instruction: **RSPID**
Class: XOP Read SPJD

Purpose: OP 1←USER_STATUS.SP_ID. OP2 **is unused.**

Precision: OP1 is a single-word. OP2 is **unused** (OD2 must equal **zero).**

The following loads the top stack **element** into RTA, without **first** knowing which register is the stack pointer (as long as it is **not** RTA!).

```
RSPID RTA              ;RTA←stack register number
MOV RTA,@0(RTA)↑2      ;RTA←top of stack
```

WSPID

Instruction: **WSPID**
Class: XOP Write SPJD

Purpose: USER_STATUS.SP_ID←OP1. If OP1>31 or OP1<0, the result is undefined. A hard trap will occur if OP1=3 or OP1=31 (these are illegal values for SP_ID). OP2 is unused.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets the stack pointer/limit to the last two registers.

```
WSPID #36          ; SP=%36, SL=%37
```

RRNDMD

Instruction: **RRNDMD**
Class: XOP Read rounding mode

Purpose: OP 1←USER_STATUS.RND_MODE. OP2 is unused. See Section 5.3.1 for a description of rounding modes.

Precision: OP I is a single-word. OP2 is unused (OD2 must equal zero).

The following jumps to FLOOR if floor rounding is specified by USER-STATUS.

```
RRNDMD RTA
JMPZ.EQL RTA,FLOOR
```

WRNDMD

Instruction: WRNDMD
Class: XOP Write rounding mode

Purpose: USER_STATUS.RND_MODE←OP1. If OP1>31 or OP 1<0, the result is undefined. OP2 is unused. See Section 5.3.1 for a description of rounding modes.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets the USER-STATUS to specify floor rounding.

```
WRNDMD #0
```

RPS

Instruction: **RPS**
Class: XOP Read processor status

Purpose: OP 1←**PROC_STATUS**. OP2 is unused.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets RTA to **PROC_STATUS**.

```
RPS RTA
```

WFSJMP

Instruction: WFSJMP
Class: JOP

Write **full status** and jump

Purpose: USER_STATUS←OP 1. PROC_STATUS←NEXT(OP 1). Note that NEXT(OP 1) is loaded directly into PROCSTATUS without interpreting the PREV/CRNT_FILE or PREV/CRNT_MODE fields in the special way **that** is done when loading partial processor **status.** (See Section 2.5.1 for a discussion of processor status.) Note that a hard trap will occur if an illegal **value of** PROCSTATUS is specified.

Restrictions: Illegal in user mode.

Precision: A 11 operands concerned are single-words.

The following sets PROCSTATUS to NEWPST and jumps to BRAZIL. 1

```
WFSJMP NEWPST,BRAZIL
```

RCFILE

Instruction: **RCFILE**
Class: XOP Read **CRNT_FILE**

Purpose: OP 1←**PROC_STATUS.CRNT_FILE**. OP2 is unused.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word, OP2 is unused (OD2 must equal zero).

The following sets RTA to the current file number.

```
RCFILE RTA
```

WCFILE

Instruction: WCFILE
Class: XOP Write CRNT_FILE

Purpose: PROC_STATUS.CRNT_FILE←OP 1. If OP 1> 15 or OP 1<0, the result is undefined. OP2 is unused.

Restrictions: Illegal in user mode.

Precision: OP1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets the current file number to the value in RTA.

```
WCFILE RTA
```

RPFILE

Instruction: **RPFILE**
Class: XOP

Read **PREV_FILE**

Purpose: OP 1←PROC_STATUS.PREV_FILE. OP2 is **unused.**

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 **is unused (OD2 must equal zero).**

The following loads RTA with the previous file number.

```
RPFILE RTA
```

WPFILE

Instruction: **WPFILE**
Class: XOP Write **PREV_FILE**

Purpose: **PROC_STATUS.PREV_FILE←OP** 1. If **OP1>15** or **OP1<0,** the result is undefined. OP2 is unused.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must **equal** zero).

The following sets the previous file number to the value in RTA.

```
WPFILE RTA
```

RPID

Instruction: RPID
Class: XOP Read processor identification number

Purpose: OP1←PROC_ID. OP2 is unused.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 is unused (OD2 must equal zero).

The following sets RTA to the processor ID number.

```
RPID RTA
```

5.15 Cache and Map

Each S-l processor has two private caches to reduce memory access times for those sections of memory that are frequently accessed. One cache is for instructions. The other is for data. The instruction cache retains only locations from pages marked with INSTRUCTIONS=1, the data cache retains locations from pages marked with DATA=1. (See Section 2.3.2 for details on access modes.) Instruction words may not, in general, be accessed as data (except as immediate operands). Special instructions are provided for converting instructions to data and data to instructions. (See BLKID, and BLKDI in Section 5.13 for details.)

Each cache uses physical *addresses* to tag entries, allowing the software to switch virtual addresses spaces without sweeping the cache. This eliminates the problem of clogging the cache with multiple copies of shared read-only information.

For purposes of communication or synchronization, it may be necessary to insure that certain variables are not present in the cache of a specific processor. Access modes serve this purpose and are described in Section 2.3.2. In addition, special instructions are provided to sweep the caches (SWPIC and SWPDC). Sweeps may either update main memory, invalidate the cache residents, or both.

No instructions are provided which, when executed on processor PA, cause the cache of processor P_B to be swept ($A \neq B$). This necessary function will be accomplished by directing a special interrupt from PA to P_B which causes P_B to sweep its own cache.

Each processor also has two page map caches. These contain, for the most recently used pages, the complete translation from virtual page addresses to physical page addresses. See Section 2.3 for a discussion of the virtual-to-physical translation. One map is for the addresses of instructions and the other is for the addresses of data. Special sweep commands are provided for the maps (SWPIM, SWPDM).

Two other commands are discussed in this section: WEPJMP and WUPJMP. These write into the executive/user segment pointer/limit registers (see Section 2.3).

SWPIC

Instruction: **SWPIC . {RTA,RTB} . {V,P}**
Class: XOP Sweep instruction cache

Purpose: Sweep the instruction cache by {Virtual,Physical} addresses, *killing* residents. To kill means to remove cache residents without updating memory. Updating is not provided for the instruction cache since residents in the instruction cache cannot be modified. OP1 is the block to be swept. {RTA,RTB} contains the number of quarter-words to be swept (which must be a multiple of four (4) or a hard trap will occur).

The address sequence generated by the instruction may be interpreted by the hardware as either virtual or physical addresses, depending on the modifier (V=virtual,P=physical). Physical-address sweeps are legal only in executive mode to prevent the user from degrading system performance by sweeping addresses which not in its address space. Virtual-address sweeps are legal in both user and executive mode.

In the case of physical-address sweeps, the microcode may, for efficiency reasons, choose to sweep the entire cache, if a very large sweep range is specified. No sweep-range optimization is performed for virtual-address sweeps.

Restrictions: Illegal in user mode.

Caution: This instruction may cause a non-zero value to be stored in INSTRUCTION-STATE.

Precision: OP 1 is a block. OP2 is unused (OD2 must equal zero). RTA and RTB are single-words.

The following sweeps all instructions from START up to but not including the following instructions.

```
MOV RTA, <. -START>      ;set RTA the number of intervening QWs
SWPIC.RTA.V START        ; sweep cache
```

SWPDC

Instruction: SWPDC . {RTA,RTB} . {V,P} . {U,UK}
Class: XOP Sweep data cache

Purpose: Sweep the data cache by {Virtual, Physical) addresses, (updating, updating and killing); residents. To kill means to remove cache residents without updating memory. No instruction is provided for killing data cache residents without updating. OP 1 is the block to be swept. {RTA,RTB} is the number of quarter-words to be swept (which be a multiple of four (4) or a hard trap will occur)

The address sequence generated by the instruction may be interpreted by the hardware as either virtual or physical addresses, depending on the modifier (V=virtual,P=physical). Physical-address sweeps are legal only in executive mode to prevent the user from degrading system performance by sweeping addresses which not in its address space. Virtual-address sweeps are legal in both user and executive mode.

In the case of physical-address sweeps, the microcode may, for efficiency reasons, choose to sweep the entire cache, if a very large sweep range is specified. No sweep-range optimization is performed for virtual-address sweeps.

Restrictions: Illegal in user mode.

Caution: This instruction may cause a non-zero value to be stored in INSTRUCTION-STATE.

Precision: OP 1 is a block. OP2 is unused (OD2 must equal zero). RTA and RTB are single-words.

The following updates the registers, without removing them from the data cache (i.e., not killing them).

```
MOV RTA,?200          ;set RTA to number of QWs
SWPDC.RTA.V.U %0      ;sweep cache
```

SWPIM

Instruction: SWPIM . {E,U}
Class: XOP

Sweep instructions page map

Purpose: Sweep the instruction page map, killing (executive, user}-space residents. SWPIM is used for eliminating residents of the instruction page map. It does not update main memory since page map residents cannot be modified. OP1 is interpreted as a virtual address, and the translation entry for the page containing that virtual address is removed from the page map. OP2 is unused. Since SWPIM operates on **only** one page map resident at a time, it is fast and **not** interruptable.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 is unused (OD2 **must equal** zero).

The following kills **the page map** entry for the **next lower** addressed instruction page in **the** users address space.

```
SWPIM.U .-4000
```

SWPDM

Instruction: SWPDM . {E,U}
Class: XOP

Sweep data page map

Purpose: Sweep the data page map, killing (executive, user)-space residents. SWPDM is used for eliminating residents of the instruction page map. It does not update main memory since page map residents cannot be modified. OP1 is interpreted as a virtual address, and the translation entry for the page containing that virtual address is removed from the page map. OP2 is unused. Since SWPDM operates on only one page map resident at a time, it is fast and not interruptable.

Restrictions: Illegal in user mode.

Precision: OP1 is a single-word. OP2 is unused (OD2 must equal zero).

The following kills the-page map entry for the data page containing the virtual address specified in RTA.

```
SWPDM. U RTA
```

WUPJMP

Instruction: **WUPJMP**
Class: JOP Write user segment table pointer and jump

Purpose: USER_STP←OP 1. USER_STL←NEXT(OP 1). PC←JUMPDEST. A hard trap will occur if either OP 1 or NEXT(OP1) contains an address that is not a multiple of four. This instruction also kills all *user* residents of the instruction and data page maps.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. NEXT(OP1) is a single word.

The following sets the user segment table to the six SWs pointed to by RTA and jumps to NEXT.

```
MOVPHY- RTA,(RTA)
ADD RTA1,RTA,#6
WUPJMP RTA,NEXT
```

WEPJMP

Instruction: **WEPJMP**
Class: JOP Write executive segment table pointer and jump

Purpose: EXEC_STP←OP 1. EXEC_STL←NEXT(OP I). PC←JUMPDEST. A hard trap will occur if either OP 1 or NEXT(OP1) contains an address that is not a multiple of four. This instruction also kills all *executive* residents of the instruction and data page maps. Notice that the jump destination is computed in the *old* executive context, but the location actually transferred to will be within the *new* executive context.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. NEXT(OP 1) is a single word.

The following sets the executive segment table to the six SWs pointed to by RTA and jumps to NEXT.

```
MOVPHY RTA, (RTA)
ADD RTA1, RTA, #6
WEPJMP RTA, NEXT
```

5.16 Interrupt

Interrupts occur during **the** first stage of the instruction-execution sequence (see Section 5.1). When **an** interrupt **has been** accepted, control is transferred to **an** interrupt handler whose address is contained in **the** *interrupt-vector* associated with the particular interrupt **that** occurred. The interrupt-vector format is **shown** in Figure 5-6. The occurrence of an interrupt also causes information to be put on **the** stack in **an** interrupt save area (INTUPT_SAVE_AREA). The format of this save area is **shown in** Figure 5-7. The concepts of save areas and vectors are discussed in Section 6. The *interrupt-parameter* is used to pass information about the interrupt to the interrupt handler. The way in which interrupt requests are handled is discussed in the following paragraphs.

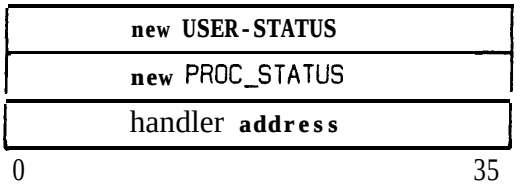

Figure 5-6
Interrupt Vector Format

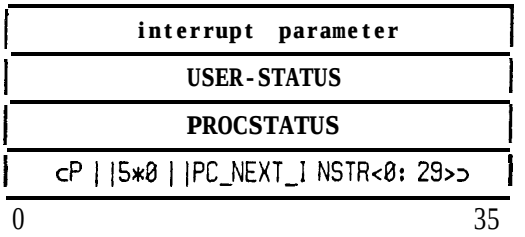

Figure 5-7
Interrupt Save Area Format

The interrupt architecture of **the S-1 allows** for eight **levels** of *priority*. The priority of **the** processor is specified by PROC_STATUS.PRIO<0:2>. The priority of any interrupts that are pending and that are enabled is specified **by** the eight-bit register INTUPT_AT_LVL<0:7>. INTUPT_AT_LVL[*i*]=1 means that one or more interrupts are pending and have been enabled **at level** i.

Associated with each priority **level** *i* (and thus with INTUPT_AT_LVL<*i*>) are two 36-bit registers INTUPT_PEND[*i*] and INTUPT_ENB[*i*]. The *interrupt-pending* registers INTUPT_PEND[0..7] **can each** accept interrupt requests from up to thirty-two devices in bits 0.. 31. Bits 32 . . 35 are unused. If device *j* with priority *i* requests an interrupt, INTUPT_PEND[*i*]<*j*> is set equal to one. The second register at each priority **level** is the *interrupt-enable* register INTUPT_ENB[0 . . '7]. INTUPT_ENB[*i*] provides interrupt-enable bits for the thirty-two devices **that are handled by** INTUPT_PEND[*i*]. As with INTUPT_PEND,

INTUPT_ENB<32:35> is unused.

If INTUPT_PEND[*i*]<*j*> and INTUPT_ENB[*i*]<*j*> are both equal to one for any combination of *i* and *j*, INTUPT_AT_LVL[*i*] will be set to one. Zero is the highest priority and seven the lowest. If there exists a priority *i*, such that INTUPT_AT_LVL[*i*]= 1 and PROC_STATUS.PRIO>*i*, **the processor will be** interrupted. If more than one bit of INTUPT_AT_LVL is set, the device with the highest priority (smallest magnitude) will **be the one** that interrupts the processor. Within a given interrupt level i, bit zero **has** the highest priority and bit thirty-one the lowest. Note that devices with priority=7 cannot interrupt **the** processor **because** PROC_STATUS.PRIO can never be greater than seven. Note also that if PROC_STATUS.PRIO=0, the processor cannot be interrupted at ail.

Each interrupting device has a unique interrupt *vector* (INTUPT_VEC) and a unique bit at priority i in INTUPT_PEND[*i*] associated with it. **When** a device interrupt occurs the appropriate bit of INTUPT-PEND is **set** and the interrupt-parameter **is** stored in a calculated position of INTUPT_PARM[0:255], **a** RAM located in the S-l processor. (The calculation is to create an INTUPT-VECNUM, described below.) When an interrupt from a device has been accepted (as described above), control is transferred to the address specified by the handier address in the interrupt vector. The INTUPT_PEND[*i*]<*j*> bit that caused the interrupt is cleared. New USER_STATUS and PROC-STATUS words are also loaded from the interrupt vector. The old USER--STATUS and PROC-STATUS words are saved in the interrupt *save* area (INTUPT-SAVE-AREA). The interrupt-parameter, which contains information about the cause of the interrupt, is **also** saved in INTUPT-SAVE-AREA. The format of INTUPT-SAVE-AREA is shown in Figure *5-7.*

Instructions are provided to read, write, set and clear INTUPT_ENB and INTUPT-PEND. There are also instructions to read and and write an interrupt-parameter. Ail interrupt instructions are legal in both executive and user mode.

Two terms that are used in the following instruction descriptions are INTUPT_LVL_NUM and INTUPT_ VEC_NUM. INTUPT_LVL_NUM is a 3-bit *interrupt level-number* (ILN), right-justified in a single-word field of zeros (i.e., ⊂33*0 || ILN<0:2>⊃). It is used to specify **a** priority level. INTUPT-VECNUM is a S-bit level-number (ILN) concatenated with a 5-bit *interrupt bit-number* (IBN) within **the** level, all right-justified in a single-word (i.e., ⊂28*0 || ILN<0:2> || IBN<0:4>⊃). It uniquely specifies **a** particular interrupt vector **number.** (Note that **the** INTUPT-VECNUM is also the location of the interrupt-parameter in INTUPT_PARM.)

RIEN

Instruction: RIEN
Class: XOP

Read interrupt enable

Purpose: OP2 is an INTUPT_LVL_NUM. OP1 gets the contents of the interrupt-enable register associated with priority level OP2 (INTUPT_ENB[OP2]).

Restrictions: Illegal in user mode.

Precision: OP 1 and OP2 are both single-words.

The following loads RTA with the enable bits for the highest priority level.

```
RIEN RTA,#0
```

WIEN

Instruction: **WIEN**
Class: XOP Write interrupt enable

Purpose: OP1 is an INTUPT_LVL_NUM. The interrupt-enable register associated with priority level OP 1 (INTUPT_ENB[OP1]) is set to OP2. If OP2<32:35> ≠ 0, then a hard trap will occur.

Restrictions: Illegal in user mode.

Precision: OP 1 and OP2 are both single-words.

The following enables ail interrupts at the second-highest priority level. 1

```
WIEN #1,#c-20>
```

SIEN

Instruction: **SIEN**
Class: XOP

Set specified bits in interrupt enable

Purpose: OP1 is an INTUPT_LVL_NUM. The interrupt-enable bits (for priority level OP 1) corresponding to the one bits of OP2 are set to one (i.e., INTUPT_ENB[OP1]←OP2 v INTUPT_ENB[OP1]).

Precision: OP 1 and OP2 are both single-words.

The following enables for interrupt by the third-highest priority device at the third-highest priority level.

```
SIEN #2,#c100000,,0↵
```

CIEN

Instruction: **CIEN**
Class: XOP

Clear specified bits in interrupt enable

Purpose: OP 1 is an INTUPT_LVL_NUM. Clear the interrupt-enable bits (for priority level OP 1) corresponding to the one bits of OP2 (i.e., INTUPT_ENB[OP1]←one's-complement(OP2) A INTUPT_ENB[OP1]).

Precision: OP 1 and OP2 are both single-words.

The following disables interrupts by the fourth-highest priority device at the fourth-highest priority level.

```
CIEN #3,#c40000,,0ↄ
```

RIPND

Instruction: **RIPND**
Class: *XOP* Read interrupts pending

Purpose: OP2 is an INTUPT_LVL_NUM. OP1 gets the contents of the interrupt-pending register associated with priority level OP2 (INTUPT_PEND[OP2]).

Precision: OP1 and OP2 are both single-words.

The following sets RTA to the pending interrupts at the fourth-lowest priority level.

```
RIPND RTA,#4
```

WIPND

Instruction: **WIPND**
Class; XOP Write interrupts pending

Purpose: OP 1 is an INTUPT_LVL_NUM. **The interrupt-pending** register associated with priority level OP1 (INTUPT_PEND[OP1]) **is** set to *OP2.* If OP2<32:35> ≠ 0, **then** a hard trap **will occur.**

Precision: OP 1 and OP2 are both single-words.

The following sets interrupts pending for ail devices at the third-lowest priority level.

```
WIPND #5,#c-20>
```

SIPND

Instruction: **SIPND**
Class: XOP

Set specified interrupt-pending bits

Purpose: OP1 is an INTUPT_LVL_NUM. The interrupt-pending bits (for priority level OP1) corresponding to the one bits of OP2 are set to one (i.e., INTUPT_PEND[OP1]←OP2 v INTUPT_PEND[OP1]).

Precision: OP1 and OP2 are both single-words.

The following sets an interrupt pending for the second-lowest priority device at the second-lowest priority level.

```
SIPND #6,#c40>
```

CIPND

Instruction: **CIPND**
Class: XOP Clear specified interrupt-pending bits

Purpose: OP1 is an INTUPT_LVL_NUM. Clear the interrupt-pending bits (for priority level OP1) corresponding to the one bits of OP2 (i.e., INTUPT_PEND[OP1]←one's-complement(OP2) ∧ INTUPT_PEND[OP1]).

Precision: OP1 and OP2 are **both single-words.**

The following clears any interrupt pending for the lowest priority device at **the** lowest priority level.

```
CIPND #7,#<20>
```

RIPAR

Instruction: **RIPAR**
Class: XOP

Read interrupt parameter

Purpose: OP2 is an INTUPT_VEC_NUM. OP1 gets the contents of INTUPT_PARM[OP2].

Precision: OP1 and OP2 are both single-words.

The following sets RTA to the interrupt parameter for vector 1.

```
RIPAR RTA,#1
```

WIPAR

Instruction: **WIPAR**
Class: XOP

Write interrupt parameter

Purpose: OP 1 is **an** INTUPT_VEC_NUM.INTUPT_PARM[OP1] is set to OP2.

Precision: OP 1 **and** OP2 are **both** single-words.

The following sets the interrupt parameter for vector 1 to RTA.

```
WIPAR #1,RTA
```

5.17 Input/Output

The S-l performs I/O via I/O buffers. The number of I/O buffers is implementation dependent (with upper bound 2^9). The Mark II contains eight *I/O buffers* (IOBUF[0:7]). Each of the eight IOBUFs contains 2K single-words. Each IOBUF is connected to exactly one *I/O* Processor (IOP) through a simple interface (IOBUF_IFACE) in the IOP. One IOP may be connected to multiple IOBUFs. Devices on the IOP's internal bus (IOP-BUS) address the IOBUF either as 32-bit words or as pairs of 16-bit words. These 32-bit words are right-justified in the 36-bit memory. The extra four bits allow the S-l processor to use the buffers as auxiliary storage. The IOBUF_IFACE can be configured by the IOP so that the addresses of the IOBUF can start at **any** (aligned) IOP-BUS address.

The IOP and devices **on** the IOP-BUS can read **and** write locations in the IOBUF as normal IOP-BUS locations (including &bit, 16-bit, and 32-bit writes). The S-l processor can read and write IOBUF locations in a single cycle as 36-bit single-words. A synchronization mechanism is provided to prevent simultaneous access. One set of translation hardware is located between the eight IOBUFs **and the main** data path of the S-l processor. This hardware is able to do four different types of translations in each direction.

IOBUF to Processor	Processor to IOBUF	Name
Bit stream	Bit stream	B
8 bits right-justified in QW	QW<1:8> in 8 bits	a
16 bits right-justified in HW	HW<2:17> in 16 bits	H
32 bits right-justified in SW	SW<4:35> in 32 bits	S

QW=quarter-word, HW=half-word, SW=single-word.

Table 5-10
Processor/IOBUF Translations

Certain areas within **each** IOBUF are, by convention, dedicated to IOP/S-1 control **communication.** All device interrupts are forwarded through **an** IOP to the S-l processor. Interrupts are described fully in Section 5.16. When a device interrupt occurs, **the** IOP writes control information into the control section of the IOBUF (including the INTUPT-PEND register number, the INTUPT-PEND bit number, the interrupt-parameter). The IOP then interrupts the S- 1 processor. The S-l processor immediately processes **the** interrupt and interprets the control information in the IOBUF. It should be noted that before the IOP writes **the** control area of IOBUF, it busy-waits until the previous interrupt has been serviced **by the** S-l processor.

Similarly, when the S-l processor needs to interrupt the IOP, it sets up the contents of another portion of the control area of the appropriate IOBUF and **executes an** instruction which

causes the IOP to interrupt and interpret the IOBUF control area. The S-l processor **also** does a busy-wait to avoid conflicts.

There are instructions to fill and empty an IOBUF, and to interrupt an IOP. All I/O instructions are legal in either executive or user mode.

When an operand is to be interpreted as a IOBUF address, the following procedure is used. The virtual address which results from the operand address calculation must reside on an I/O page (see Section 2.3.2). The standard virtual-to-physical address transformation takes place (see Section 2.3). The resulting physical address is not interpreted as a physical address in memory, but rather as **an** IOBUF *physical address* (IOBUF-PHYADDR). IOBUF-PHYADDR has the following format: ⊂7*0 || IOBUF_NUM<0:8> || ADDR_IN_IOBUF<0:17>⊃. IOBUF_NUM refers to the number of the IOBUF to **be** accessed. **(On** the Mark IIa IOBUF_NUM must be in the range 0 .. 7.) ADDR_IN_IOBUF specifies the 32-bit-word address within the selected IOBUF. If IOBUFNUM **is** larger than the maximum available, or if ADDRJNJOBUF is not a valid 32-bit-word address within an IOBUF, or if the first seven bits of IOBUF-PHYADDR are not zero, or if the virtual address specified was not on an I/O page then a hard trap will occur.

This virtual-to-physical transformation allows the executive to maintain control over the I/O buffers, even though the I/O instructions are legal in user mode. It is up to the executive to set up the transformation to a valid IOBUF address and to indicate that the virtual page is a valid I/O page.

BLKIOR

Instruction: **BLKIOR . {RTA,RTB} . {B,Q,H,S}**
Class: XOP Block I/O read and translate

Purpose: Transfer from an IOBUF to main memory. OP1 is the destination memory block. {RTA,RTB} contains the quarter-word block length. OP2 is the source IOBUF block. {B,Q,H,S} specifies the type of translation between the IOBUF and the processor.

Caution: This instruction may cause a non-zero value to be stored in INSTRUCTION-STATE.

Precision: OP1 is a block. OP2 is an IOBUF block. RTA and RTB are single-words.

Assume BUFFER is a legitimate IOBUF address. To read eighty characters from the IOBUF (starting at BUFFER) to a block in memory starting at IMAGE the following instruction sequence could be used.

```
MOV RTA,?120                    ;set RTA to eighty QWs
BLKIOR.RTA.Q IMAGE,BUFFER       ;do read
```

BLKIOW

Instruction: BLKIOW . {RTA,RTB} . {B,Q,H,S}
Class: XOP Block I/O write and translate

Purpose: Transfer from main memory to an IOBUF. OP1 is the destination IOBUF block. {RTA,RTB} contains the quarter-word **block** length. OP2 is the source memory block. {B,Q,H,S} specifies the type of translation between the processor and the IOBUF.

Caution: This instruction **may cause a** non-zero value to **be** stored in INSTRUCTION-STATE.

Precision: OP 1 is an IOBUF block. OP2 is a block. RTA and RTB are single-words.

Assume BUFFER is a legitimate IOBUF address. To transfer the two characters "S1" into the IOBUF starting at BUFFER the following instruction **sequence** could be used.

```
MOV  RTA,#2                          ;set RTA to two QWs
BLKIOW.RTA.Q BUFFER,#c"S1",,0        ;do  write
```

INTIOP

Instruction: **INTIOP**
Class: XOP Interrupt I/O processor

Purpose: OP1 is an IOBUF address. The IOP connected to the IOBUF containing OP1 is interrupted. OP2 is unused.

Precision: OP 1 is a single-word (and must transform to a valid IOBUF_PHY_ADR). OP2 is unused (and hence OD2 must be zero).

Assume BUFFER is a legitimate IOBUF address. The following instruction will interrupt the I/O Processor containing BUFFER.

```
INTIOP BUFFER
```

5.18 Performance Evaluation

The S-l has several double-word counters which can be configured to count different events. These counters are all be readable in user mode, but they are be writable only in executive mode. Each counter has enable bits associated with it, accessible only in executive mode. Counter zero is always enabled, *by convention,* to count real-time cycles.

RCTR

Instruction: **RCTR**
Class: XOP

Read counter

Purpose: OP2 is a counter number. OP1 gets the contents of the counter specified by OP2.

Precision: OP1 is a double-word. OP2 is a single-word.

The following sets RTA (DW) to the current real-time cycle count.

```
RCTR RTA, #0
```

WCTR

Instruction: **WCTR**
Class: XOP Write counter

Purpose: **OP** 1 is a counter number. Write OP2 into the counter specified by OP **1.**

Restrictions: Illegal in user mode.

Precision: OP1 is a single-word. OP2 is a double-word.

The following zeros the real-time cycle counter.

```
WCTR #0,#0
```

RECTR

Instruction: **RECTR**
Class: XOP

Read enable bits for counter

Purpose: OP2 is a counter number. OP1 gets the contents of the enabling register for the counter specified by OP2.

Restrictions: Illegal in user mode.

Precision: OP 1 is a double-word. **OP2** is a single-word.

The following reads the enabling bits for counter COUNT into **RTA.**

```
RECTR RTA, COUNT
```

WECTR

Instruction: WECTR
Class: XOP Write enable bits for counter

Purpose: OP1 is a counter number. Write OP2 into the enabling register for the counter specified by OP1.

Restrictions: Illegal in user mode.

Precision: OP 1 is a single-word. OP2 is a double-word.

The following writes ENABLE into the enabling register for counter COUNT.

```
WECTR  COUNT,ENABLE
```

5.19 Miscellaneous

The instructions in this section fit no general category.

NOP

Instruction: **NOP**
Class: XOP

No operation

Purpose: NOP may have operands, but it performs no operation and stores no result. It always transfers control to the next Instruction. The operand addressing calculations are carried through; while the operands themselves are not referenced, an invalid addressing mode will cause a hard trap.

Precision: OP 1 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)>(SP)↑2
```

JPATCH

Instruction: **JPATCH**
Class: HOP Jump to patch

Purpose: JPATCH is an unconditional jump instruction which uses ⊂OD1||OD2⊃ as a signed 24-bit offset from the PC to form the jump address. It is intended for use by a debugger, to allow a single-word instruction to be replaced by a jump to a patch area. The use of JPATCH **in** ordinary user eode is discouraged; for most purposes **JMPA should** be used instead.

Precision: OP 1 and OP2 may be any precision since they are not fetched.

This instruction occupies only one instruction word.

```
JPATCH PATCH.AREA
```

XCT

Instruction: **XCT**
Class: XOP Execute

Purpose: Execute the instruction OP1. If that instruction requires extended-words, then NEXT(OP1) and NEXT(NEXT(OP1)) are used as necessary. During execution of the instruction OP 1, PC means the PC of the XCT instruction, *not* the address of OP 1. Similarly, PC_NEXT_INSTR means the PC of the instruction following the XCT. PC is used in all indexing off Register 3 during the interpretation of the executed instruction. PC and PC_NEXT_INSTR are stored on the stack as specified when executing a context-saving instruction (e.g., TRPSLF or instruction which traps due to an error). Chaining XCT instructions is legal; in this case PC and PC-NXTJNSTR always refer to those of the first XCT in the chain. OP2 of an XCT is unused. If OP1 of an XCT instruction is an immediate constant (either long, short, or indexed) then a hard trap will occur. If an enabled interrupt occurs during the execution of an XCT chain, the interrupt will be serviced, and the XCT chain will be restarted upon return. OP 1 (and the next two single-words following OP1) of an XCT must be located on a page marked with DATA=1. As with all instructions, the two single-words following the XCT instruction itself must be on a page marked with INSTRUCTIONS=1.

The XCT instruction must have its operand in the current address space. The instruction being executed by XCT may access the previous address space with the same effect as if that instruction were executed in-line.

XCT is very slow.

Precision: OP1 is a single-word. OP2 is unused (OD2 must equal 0).

Let SP be the stack pointer. Assume an entire instruction has been pushed on the stack, followed by the negative of the number of extended words that the instruction used. The following executes the stacked instruction.

```
XCT c-2(SP)↓(-1(SP))↑2
```

RMW

Instruction: RMW
Class: TOP Read/modify/write

Purpose: In one memory cycle (and hence indivisibly with respect to other processors in a multiprocessor system), DEST←S2 and then S2←S 1.

Precision: S 1, S2, and DEST are all single-words.

The following illustrates the use of RMW to implement a test-and-set lock for interprocessor communication, The lock is a single-word flag which is -1 if some processor has seized the lock and 0 if the lock is free.

```
SEIZE:  RMW RTA,#-1,LOCK        ;attempt to seize lock
        JMPZ.NEQ RTA,SEIZE      ;busy-wait if someone else has it
        ...                     ;do ... if lock was zero (now I have it)
FREE:   MOV LOCK,#0             ;release the lock
```

WAIT

Instruction: **WAIT**
Class: XOP **Wait for interrupt**

Purpose: Cause the processor **to wait for an interrupt.**

Restrictions: Illegal in user mode.

Precision: OP 1 and OP2 are unused; **hence** OD 1 and OD2 **must** be zero.

The following instruction waits for an interrupt.

```
WAIT
```

HALT

Instruction: HALT
Class: JOP

Halt this processor

Purpose: Halt the processor. Execution continues at JUMPDEST when the halted processor continues. HALT only halts the processor that executes it. OP1 is unused.

Restrictions: Illegal in user mode.

Precision: OP1 is unused (OD1 must be zero).

The first instruction continues at CONT; the second halts immediately upon continuation.

```
HALT CONT
HALT .
```

6 Traps and Interrupts

Traps and interrupts provide a convenient means of handling exceptional conditions that arise during program execution. They make use of *trap vectors* and *interrupt vectors* to direct control to exception handling routines. Each type of trap (as well as interrupts) has a Mock of vectors associated with it. These *vector blocks* are located at fixed addresses in memory. (See Figure 6-l.) The trap vector associated with each particular trap (interrupt) is located at a fixed offset from the beginning of its vector block. See Section 6.5 for the formats of the different types of trap vectors.

A trap (interrupt) causes a new PC to be loaded from the *handler address* that is specified in the trap vector. The low order 30-bits of the handler address specify the address of the routine that will service the exception (the high-order bits are ignored). Other information such as status words may also be loaded from the vector associated with the particular trap (interrupt). These values are loaded *after* the previous state of the processor has been saved on the stack. The group of words that is stored on the stack is called a save *area.*

The save area associated with a trap (interrupt) may contain information that is used by the routine that will handle the trap (interrupt). Information that is put in the save area typically includes the PC of the next instruction to be executed, status words, and information needed to determine the cause of the trap (interrupt). The formats of the various different types of save areas are shown in Figures 6-3, 6-4, and 6-5.

6.1 Soft Traps

A *soft trap* can occur as the result of certain types of instruction execution errors (e.g., integer-overflow). It causes control to be transferred to the handler address that is specified in SFTERRVEC. Soft-trap vectors are located in the same address space in which the soft trap occurred (i.e. user traps to soft-trap vectors in the user's address space and the executive traps to soft-trap vectors in the executive's address space. See Figure 6-l). They start at address SFTERRVECS and occupy 400_8 single-words giving a maximum of 85 vectors (three words per vector). The format of SFTERR_VEC is shown in Figure 6-2.

Soft traps cause a number of words to be saved in the soft-trap save area. These are shown in Figure 6-4. USER-STATUS is saved in a temporary location, and a new value is loaded from the soft-trap vector. When all values shown have been stored on the stack, control is transferred to the handler specified by the handler address in the soft-trap vector.

The RETUS instruction is used to return from soft traps. It is described in detail in Section 5.9 along with the return instructions.

6.2 Hard Traps

A *hard trap* **can occur as** the result of certain types of illegal operations (e.g., attempting to write a read-only page of memory). It **causes** control to be transferred to the handler address that is specified in HRDERR_VEC. **The** hard-trap vectors start a& location HRDERR_VECS and occupy 1000_8 single-words (thus, maximum number of vectors is 170). All hard-trap vectors are located in the executive's address space. They are shown in Figure 6-1.

During the processing of a hard trap, the old PROCSTATUS and USER-STATUS are first saved in temporary locations. New PROCSTATUS and USER-STATUS are then loaded from the trap vector. Note that the new PROCSTATUS defines a **new** stack and thus the location of the save area. The remainder of the information that is put into save areas depends **on** the type of hard trap. There are three types of hard traps: *nested hard traps, fatal hard traps,* and *recoverable hard traps.*

Nested hard traps **are due** to hard errors that occur within a hard trap or interrupt initiation. They save the address of **the** hard-trap vector from which the nested hard trap occured in NESTED-HARD-SAVE-AREA. Fatal hard traps are hard traps from which recovery is not normal. Information about the trap is saved in FATAL-HARD-SAVE-AREA. Recoverable hard traps are hard traps from which recovery is the normal case. Information needed to effect recovery is saved in RECOV_HARD_SAVE_AREA. The formats for the save areas of **the above** mentioned types of hard traps are shown in Figure 6-3.

The RETFS instruction is used to return from hard traps. It **is** described in detail later **on in** this section.

6.3 Trace-Traps

Trace-trapping occurs before instructions when trace-trapping is enabled. It is useful for debugging purposes, and for performance evaluation. The trace-trap feature uses two bits in PROCSTATUS (TRACE_PEND and TRACEENB) to ensure that the proper number of trace traps occur, and that they occur at the right times. After interrupts are processed during the first stage of the instruction-execution sequence, the TRACE-PEND bit is sampled and reset. If TRACE_PEND=1, then **a** trace-trap will occur immediately. If TRACE-PEND=O, then the instruction-execution **sequence will** proceed normally. The details of **the** trace-trap mechanism are described in Section 5.1.

6.4 Interrupts

Interrupts are similar to traps in the sense that **they have** vectors and save areas associated with them. The *interrupt vectors* are located after **the** trap vectors in **the user's** address space as shown in Figure 6-l. The main description of **the** interrupt architecture is discussed in the Section 5.16 along with the descriptions of the interrupt instructions.

6.5 Vector Locations and Formats

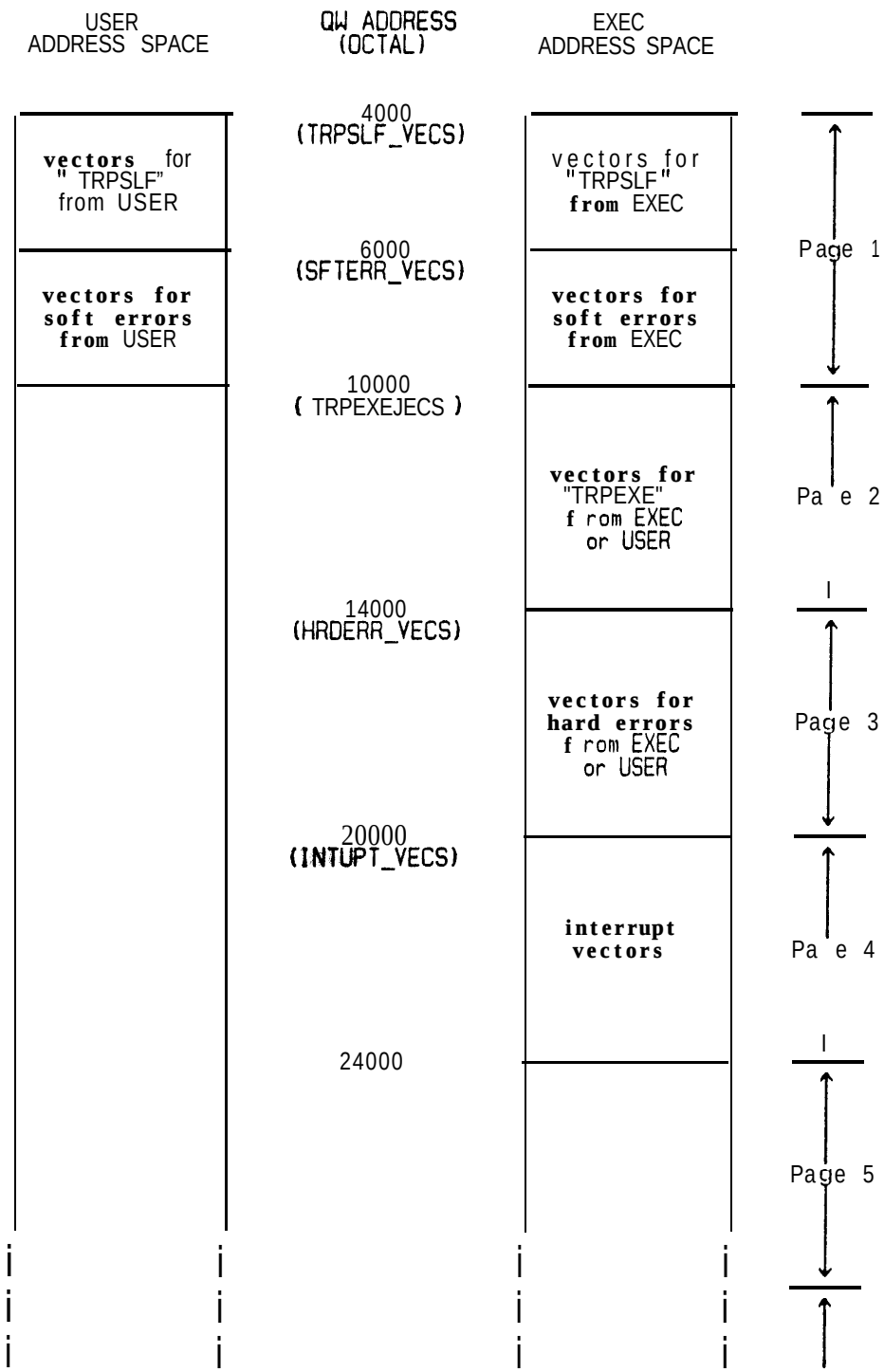

Figure 6-l
Trap and Interrupt Vector Locations

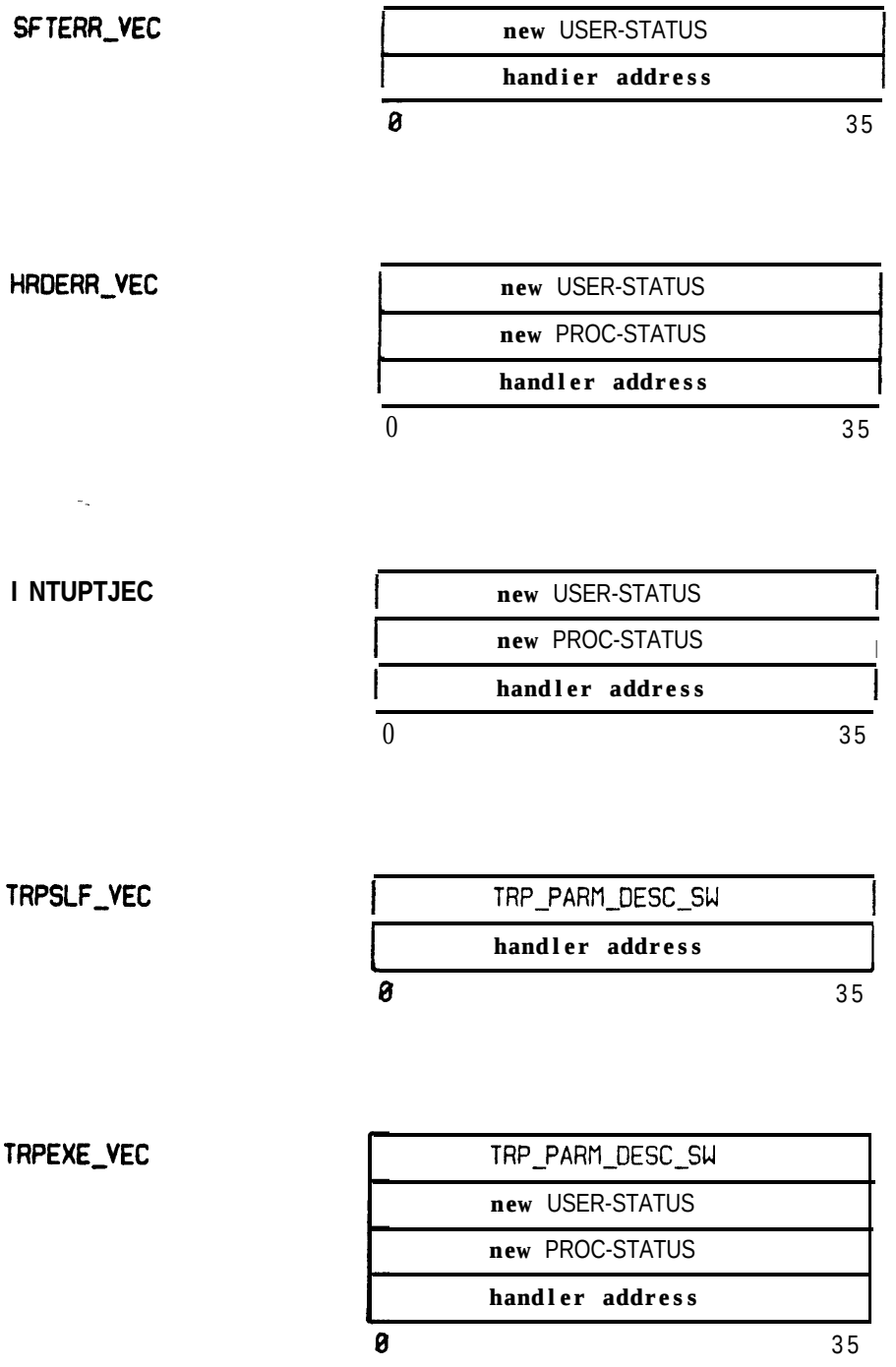

Figure 6-2
Trap and Interrupt Vector Formats

6.6 Save Area Formats

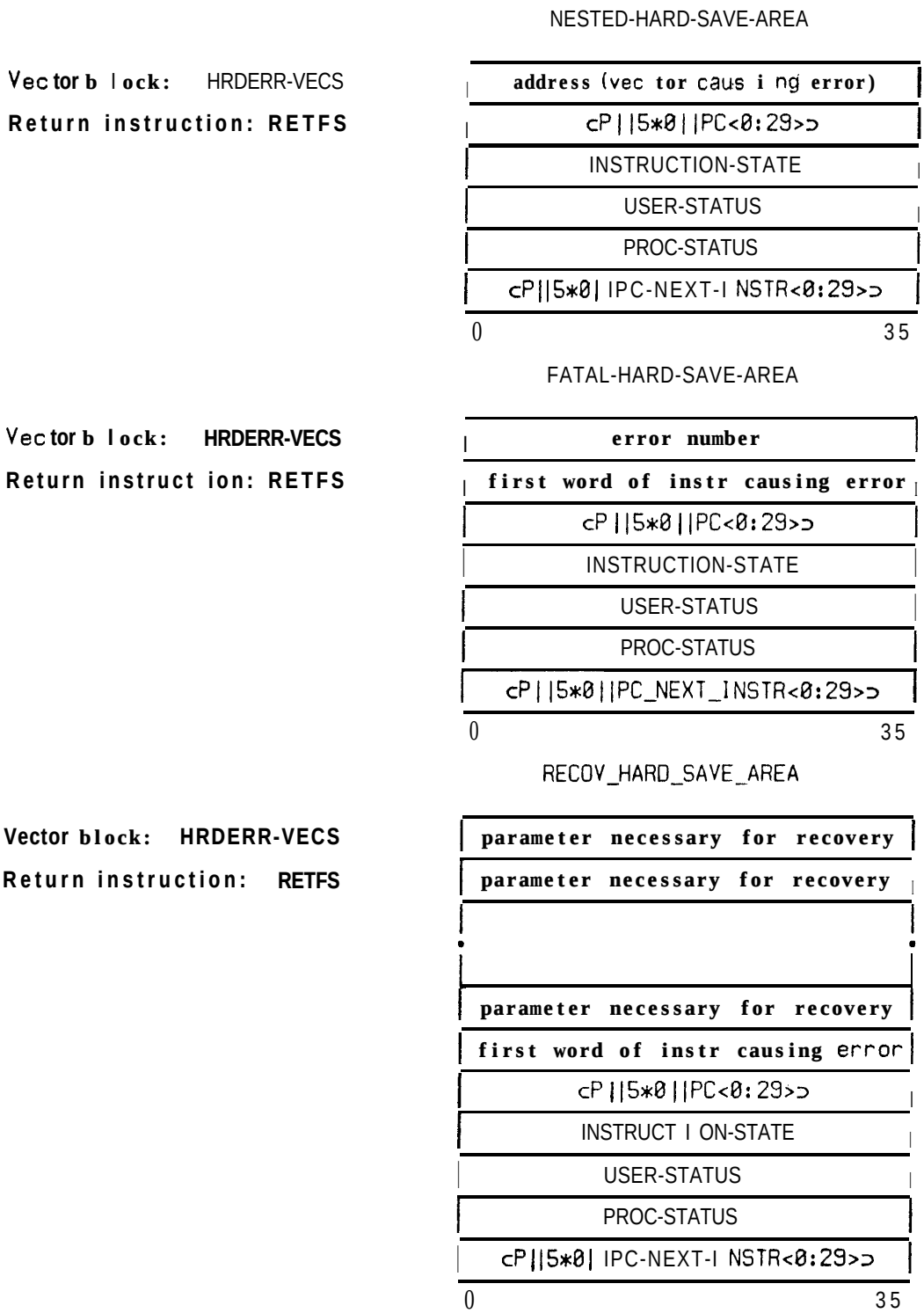

Figure 6-3
Hard-Trap Save Area Formats

Vector block : SFTERR_VECS

Return instruction; RETUS

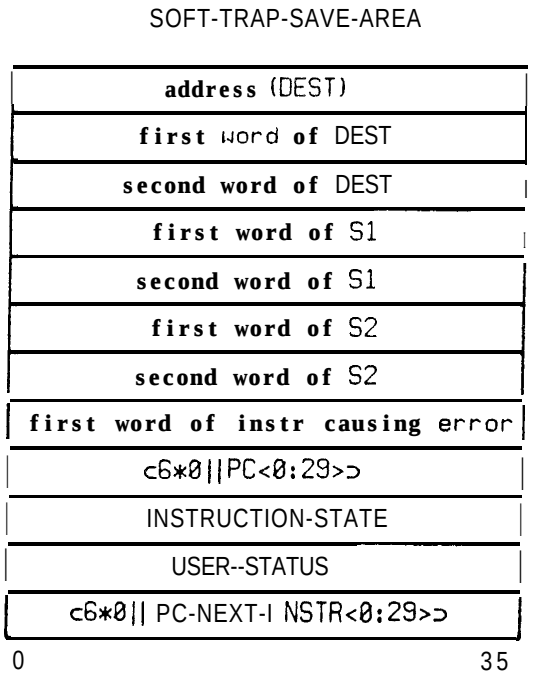

Figure 6-4
Soft-Trap Save Area Format

Vector block: INTUPT_VECS

Return instruction: RETFS

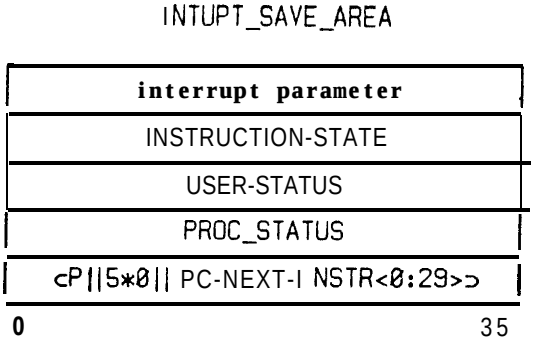

Figure 6-5
Interrupt Save Area Format

Vector block: TRPSLF_VECS
Return inetruction: RET

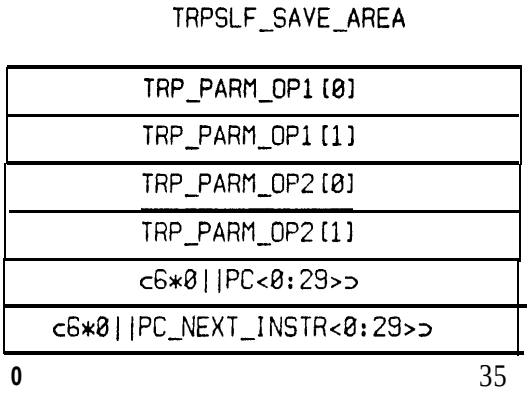

Figure 6-6
TRPSLF Save Area Format

Vector block: TRPEXE_VECS
Return instruction: RETFS

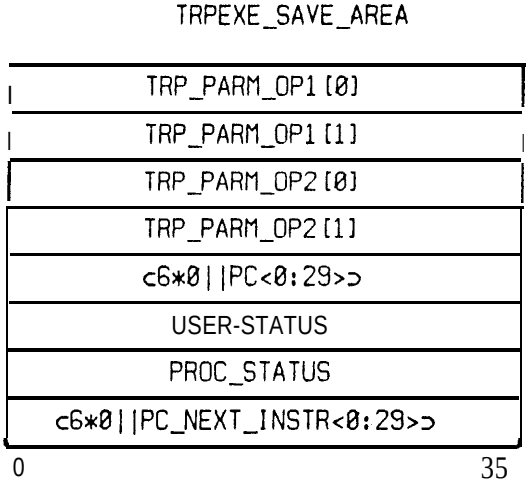

Figure 6-7
TRPEXE Save Area Format

Vector Name	Error Condition
TRACE-VEC	Trace-trap. due to TRACE_PEND=1
STK_OVFL_VEC	SP > SL
PG_FAULT_VEC	Page fault for a page not in memory
ADDRESS_MD_VEC	Illegal access mode (VA.ACCESS is illegal)
USER_P_VEC	User attempt to access previous context with P-bit=1
EXEC_ONLY_VEC	User attempted to execute a privileged instruction

Table 6-1
Recoverable Hard-Trap Vector Descriptions

Error Number	Description
1	Error during soft trap
2	Address not aligned
3	register-boundary error
4	P-bit used twice, operand of XCT, or jump dest
5	Trap descriptor out of range
6	Illegal instruction
7	Illegal F-field
8	Non-zero unused OD-field
9	Register number out of bounds
10	Short-operand addressing mode 2
11	Unused
12	Jump to the registers
13	Immediate as destination, ADDRESS(), jump destination, NEXT(), or XCT
14	Illegal byte pointer
15	Illegal block alignment
16	I/O buffer physical address is out of range

Table 6-2
Fatal Hard-Trap Error Numbers

Vector Name	Error Condition
FLT_OVFL_VEC	Integer-overflow and INT_OVFL_ENB= 1
FLT_UNFL_VEC	Floating-overflow and FLT_OVFL_ENB= 1
FLT_NAN_VEC	Floating-underflow and FLT_UNFL_ENB= 1
INT_OVFL_VEC	Zero-divide and INT_Z_DIV_MODE=0
INT_Z_DIV_VEC	Bounds check error
BND_CHK_VEC	

Table 6-3
Soft-Trap Vector Descriptions

7 Acknowledgments

We wish to acknowledge crucial support for this work which has been received from the Department of the Navy via Office of Naval Research Order Numbers N00014-76-F-0023, N00014-77-F-0023, and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U. S. Department of Energy under Contract No. W-7405-Eng-48), from the Computations Group of the Stanford Linear Accelerator Center (supported by the U. S. Department of Energy under Contract No. EY-76-C-03-0515) and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation).

The S-l architecture was originally designed by Thomas McWilliams and L. Curtis Widdoes, and has been extensively revised and enhanced by Jeff Rubin, Guy Steele, and Mike Farmwald. Four of the architecture designers (Thomas McWilliams, L. Curtis Widdoes, Guy Steele, and Mike Farmwald) wish to gratefully acknowledge the support of their graduate studies which has been extended by the Fannie and John Hertz Foundation.

The architecture designers also appreciate the help they have received from very many individuals who have familiarized themselves with this work and have offered their thoughtful comments on it; among these who deserve special thanks are Forest Baskett, Gordon Bell, Sam Fuller, Alan Kotok, John McCarthy, John Reiser and Lowell Wood. Forest Baskett was especially helpful in guiding the Project in its early stages.

The authors (Brent Hailpern and Bruce Hitson) wish to thank Jeff Rubin and L. Curtis Widdoes for their vital support during the writing of this manual. Without their extensive knowledge of the architecture, this manual would not have been possible. The authors also would like to acknowledge the continual guidance from Gio Wiederhold. Special thanks go to Guy Steele, who wrote the S-l Formal Description; Marc LeBrun, who provided the examples used in the manual; Jeff Rubin, who wrote the original FASM section; and Don Woods, who illuminated many otherwise dark corners of the POX text processing language. Brent Hailpern wishes to gratefully acknowledge the support of his graduate studies which has been provided by the National Science Foundation.

We also wish to particularly thank the many other members of the S-l Project tear-n, including Lowell Wood, Mike Farmwald, Hon Wah Chin, Bill Bryson, Craig Everhart, Andy Hertzfeld, Peter Schwarz, Erik Gilbert, David Wall, Beverly Kedzierski, Marsha Berger, Ramez El-Masri, Mohammad Olumi, Peter Nye, Javad Khakbaz, Rick McWilliams, Harlan Lau, Joe Skupnjak, Polle Zellweger, Peter Kessler, Phil Gerring, Hal Schectman, Manchor Ko, Hal Deering, Amy Lansky, Arthur Keller, Dick Sites, and Dan Perkins; without these people this architecture work would have been to no avail.

8 Appendix: Instruction Summary

```
DATE: 17DEC78 2207 BTH;

"
MODS :

QHSD = Q, H, S, D;
QHS = Q, H, S;
HSD = H, S, D;
HS = H, S;
SD = S, D;

BQHS = B, Q, H, S;
BQ = B, Q;

ACOND = GTR, EQL, GEQ, LSS, NEQ, LEQ;
LCOND = NON, ALL, NAL, ANY;
ALCOND = GTR, EQL, GEQ, LSS, NEQ, LEQ, NON, ALL, NAL, ANY;

RND = FL, CL, DM, HP, US;
'LFRT = LF, RT;
UPDN = UP, DN;
VP = V, P;
EU = E, U;
UUK = U, UK;
BND = B, MIN, MI, 0, 1;
RTARTB = RTA, RTB;

MQLEN = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 64, 128;
NOT031 = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31;
N1TO32 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
  19; 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32;
N2TO32 = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32;
NOT063 = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
  18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
  54, 55, 56, 57, 58, 59, 60, 61, 62, 63;

END;

CLASSES :
```

"SIGNED INTEGER"

```
ADD       .QHSD :TOP;
ADDC      .QHSD :TOP;

SUB       .QHSD :TOP;
SUBV      .QHSD :TOP;
SUBC      .QHSD :TOP;
SUBCV     .QHSD :TOP;

MULT      .QHSD :TOP;
MULTL     .QHS :TOP;

QUO       .QHSD :TOP;
QUOV      .QHSD :TOP;
QUOL      .QHS :TOP;
QUOLV     .QHS :TOP;

QUO2      .QHSD :TOP;
QUO2V     .QHSD :TOP;
QUO2L     .QHS :TOP;
QUO2 LV   .QHS :TOP;

REM       .QHSD :TOP;
REMV      .QHSD :TOP;
REML      .QHS :TOP;
REMLV     .QHS :TOP;

MOD       .QHSD :TOP;
MODV      .QHSD :TOP;
MODL      .QHS :TOP;
MODLV     .QHS :TOP;

DIV       .QHSD :TOP;
DIVV      .QHSD :TOP;
DIVL      .QHS :TOP;
DIVLV     .QHS :TOP;

INC       .QHSD :XOP;
DEC       .QHSD :XOP;

TRANS     .QHSD .QHSD :XOP;

NEG       .QHSD :XOP;
ABS       .QHSD :XOP;
```

```
MIN     .QHSD :TOP;
MAX     .QHSD :TOP;
```

"UNSIGNED INTEGER"

```
UMULT    .QHSD :TOP;
UMULTL   .QHS  :TOP;

UDIV     .QHSD :TOP;
UDIVL    .QHS  :TOP;
```

"FLOATING POINT"

```
FADD     .HSD:TOP;

FSUB     .HSD:TOP;
FSUBV    .HSD:TOP;

FMULT    .HSD:TOP;
FMULTL   .HS :TOP;

FDIV     .HSD :TOP;
FDIVV    .HSD:TOP;
FDIVL    .HS:TOP;
FDIVLV   .HS:TOP;

FSC      .HSD:TOP;
FSCV     .HSD:TOP;

FIX      .RND.QHSD.HSD:XOP;
FLOAT    .HSD.QHSD:XOP;
FTRANS   .HSD .HSD :XOP;

FNEG     .HSD:XOP;
FABS     .HSD :XOP;

FMIN     .HSD:TOP;
FMAX     .HSD:TOP;
```

"MOVE"

```
MOV      .QHSD .QHSD :XOP;
MOVMQ    .MQLEN :XOP;
MOVMS    .N2TO32 :XOP;

EXCH     .QHSD :XOP;

SLR      .N0TO31 :XOP;
SLRADR   .N0TO31 :XOP;

MOVADR :XOP;
MOVPHY  :XOP;
```

"FLAG"

```
CMPSF    .ACOND .QHSD :TOP;
BNDSF    . B N D .QHSD:TOP;
```

"BOOLEAN"

```
NOT      .QHSD :XOP;

AND      .QHSD :TOP;
ANDTC    .QHSD :TOP;
ANDCT    .QHSD :TOP;
OR       .QHSD :TOP;
ORTC     .QHSD :TOP;
ORCT     .QHSD :TOP;
NAND     ,QHSD :TOP;
NOR      .QHSD :TOP;
XOR      .QHSD :TOP;
EQV      .QHSD :TOP;
```

"SHIFT AND ROTATE"

```
SHF      .LFRT.QHSD:TOP;
SHFV     .LFRT .QHSD :TOP;

DSHF     .LFRT.QHS:TOP;
DSHFV    .LFRT.QHS:TOP;

SHFA     .LFRT .QHSD :TOP;
SHFAV    .LFRT .QHSD :TOP;

ROT      .LFRT .QHSD :TOP;
ROTV     .LFRT.QHSD:TOP;
```

```
"SKIP AND JUMP"

SKP       .ALCOND .QHSD:SOP;

ISKP      .ACOND :SOP;
DSKP      .ACOND :SOP;

JMP       .ACOND :JOP;
JMPZ      .ACOND .QHSD :JOP;
JMPA      :JOP;

IJMP      .ACOND :JOP;
IJMPZ     .ACOND :JOP;
IJMPA     :JOP;

DJMP      .ACOND :JOP;
DJMPZ     .ACOND :JOP;
DJMPA     :JOP;  --

BNDTRP    .BND .QHSD:XOP;

JPATCH    :HOP;
```

"ROUTINE LINKAGE"

```
JSR       : JOP;
JCR       :XOP;

ALLOC     .N1TO32 :XOP;

RETSR     :XOP;
RET       :XOP;
RETUS     :XOP;
RETFS     :XOP;

TRPSLF    .N0TO63 :XOP;
TRPEXE    .N0TO63 :XOP;
```

"STACK"

```
ADJSP    .UPDN :XOP;

PUSH     .UPDN .QHSD :XOP;
POP      .UPDN .QHSD :XOP;
```

"BYTE"

```
LBYT     .SD :XOP;
LIBYT    .SD :TOP;

LSBYT    .SD :XOP;
LISBYT   .SD :TOP;

DBYT     .SD :XOP;
DIBYT    .SD :TOP;

ADJBP    .SD :TOP;
```

"BIT"

```
BITRV    .QHSD :TOP;
BITRVV   .QHSD :TOP;

BITEX    .QHSD :TOP;
BITEXV   .QHSD :TOP;

BITCNT   .QHSD :XOP;
BITFST   .QHSD :XOP;
```

```
"BLOCK"

STRCMP   .RTARTB :XOP;

BLKMO'J  .RTARTB :XOP;
BLKINI   .RTARTB .QHSD :XOP;

BLKID    .RTARTB :XOP;
BLKDI    .RTARTB :XOP;
```

```
"STATUS REGISTER"

RUS      :XOP;
JUSCLR   .LCOND :JOP;
JUS      .LCOND :JOP;
WUSJMP   :JOP;
SETUS    :XOP;
CLRUS    :XOP;
RSPID    :XOP;
WSPID    :XOP;
RRNDMD   :XOP;
WRNDMD   :XOP;

RPS      :XOP;
WFSJMP : JOP;
RCFILE :XOP;
WCFILE :XOP;
RPFILE   :XOP;
WPFILE   :XOP;

RPID     :XOP;
```

"CACHE AND MAP"

```
SWPIC    .RTARTB .VP :XOP;
SWPDC    .RTARTB .VP .UUK:XOP;

SWPIM    .EU :XOP;
SWPDM    .EU :XOP;

WUPJMP :JOP;
WEPJMP   : JOP;
```

"INTERRUPT"

```
RIEN     :XOP;
WIEN     :XOP;
SIEN     :XOP;
CIEN     :XOP;

RIPND    :XOP;
WIPND    :XOP;
SIPND    :XOP;
CIPND    :XOP;

RIPAR    :XOP;
WIPAR    :XOP;
```

"INPUT/OUTPUT"

BLKIOR .RTARTB .BQHS :XOP;
BLKIOW **.RTARTB .BQHS:XOP;**

INTIOP **:XOP;**

"PERFORMANCE EVALUATION"

```
RCTR    :XOP;
WCTR    :XOP;

RECTR   :XOP;
WECTR   :XOP;
```

"MISCELLANEOUS"

```
NOP     :XOP;
XCT     :XOP;
RMW     :TOP;
WAIT    :XOP;
HALT    : JOP;
```

```
"HOKEY FOR SIMULATOR AND I/O MEMORY"

SETSTK   :XOP;
SETSYM   :XOP;
JMPCC    :XOP;
TIMER    :XOP;
INCHRW   :XOP;
INCHRS   :XOP;
OUTCHR   :XOP;
INTFE    :XOP;
BRIOM    .BQ:XOP;
BWIOM    .BQ:XOP;
RIOM     :XOP;
WIOM     :XOP;
JCOMNZ   :JOP;
```

```
" " " "END"
END;
```

9 Appendix: S-1 Formal Description

```
define acond ≡ (GTR, EQL, GEQ, LSS, NEQ, LEQ);
```

```
define Add (Addend, Augend) → Sum, Cout, Overflow next Continuation ≡
    Add-With-Carry (Addend, Augend, 0) → Sum, Cout, Overflow next
    Continuation;
```

```
define Add-With-Carry (Addend, Augend, Cin) → Sum, Cout, Overflow next Continuation ≡
    let x ≡ Addend, y ≡ Augend
    then let z
            ≡ c0:x⊃ + c0:y⊃ + zero-extend Kin, width(x) + 1)
        then let Sum ≡ low (width (x), z),
                Cout ≡ z<0>,
                Overflow ≡ (x<0> ≡ y<0>) ∧ (x<0> ≠ z<1>)
            then Continuation;
```

```
define alcond ≡ (GTR, EQL, GEQ, LSS, NEQ, LEQ, NON, ALL, NAL, ANY);
```

```
define ALL (Arg1, Arg2) ≡ ((¬ Arg1) ∧ Arg2) ≡ 0;
```

```
define ANY (Arg1, Arg2) ≡ (Arg1 ∧ Arg2) ≠ 0;
```

```
define bcnum ≡ (MIN, M1, 0, 1);
```

```
define Bits (p) ≡
    case p of                                   (* number of bits for given precision *)
      Q: 9;
      H: 18;
```

```
        s : 36;
        D: 72;
      end;
```

```
define Block_Memory_Address_Is_a_Register ≡ Use-Shadow;        (* terms missing *)
```

```
define bqhs ≡ {B, Q, H, S} I
```

```
define Bytes(p) ≡
      case p of                                    (* number of bytes for given precision *)
        Q: 1;
        H: 2;
        s: 4;
        D: 8;
      end;
```

```
define Dest ⇐ Value ≡ M[address(Dest)] 1 ← Value;
```

```
define Dfetch(M[Address]) → Word next Continuation ≡      (* ought to hack memory faults *)
      let Word = M[Address]
      then Continuation;                                   (* ought to use data cache *)
```

```
define' EQL (Arg1, Arg2) ≡ Argl = Arg2;
```

```
define    eu ≡ {E, U};
```

```
define Extract-Bits (Field, Mask) ≡
      let x = Field,
          Y = Mark,
```

```
        z ← zero-extend (0, width (Field))
    then while y ≠ 0
            do if y<0> then z ← shift (z,1) v unsigned (x<0>) fi next
               (x ← shift (x,1) also y ← shift (y,1))
            od next
        z;
```

```
define GEQ(Arg1, Arg2) ≡ Arg1 ≥ Arg2;
```

```
define GTR(Arg1, Arg2) ≡ Arg1 > Arg2;
```

```
define    hs ≡ {H, S};
```

```
define    hsd ≡ {H, S, D};
```

```
define lfetch(M[Address])→ Word next Continuation ≡        (* ought to hack memory faults *)
    let Word ← M[Address]
    then Continuation;                                     (* ought to use instruction cache *)
```

```
define Int_Overflow?[also] next3 Continuation ≡
    if OV
      then if Int_Ovfl_Enb then Overflow-Trap else Int_Ovfl ← 1 next Continuation fi
      else Continuation
    fi;
```

```
define Jump ≡ pc-nxt-instr ← jump-address;        (* see Calculate-Jump-Target *)
```

```
define lcond ≡ {NON, ALL, NAL, ANY};
```

```
define LEQ(Arg1, Arg2) ≡ Arg1 ≤ Arg2;
```

```
define lfrt ≡ {LF, RT};
```

```
define Long(p) ≡
        case p of                                   (* long version of a precision *)
          Q: H;
          H: S;
          S: D;
        end;
```

```
define LSS(Arg1, Arg2) ≡ Arg1 < Arg2;
```

```
define M[Address] ≡
        let address<0:27> = Address
        then if Memory-Address_Is_a_Register (address)
                then R[address<23:27>]
                else physical-memory [address]
             fi;
```

```
define Memory-Address_Is_a_Register (Address) ≡
        (Address<0:22> = 0) ∧ (¬ Block_Memory_Address_Is_a_Register);
```

```
define NAL(Arg1, Arg2) ≡ ((¬ Arg1) ∧ Arg2) ≠ 0;
```

```
define NEQ(Arg1, Arg2) ≡ Arg1 ≠ Arg2;
```

```
define NON(Arg1, Arg2) ≡ (Arg1 ∧ Arg2) = 0 ;
```

```
define Number-of_First_1_Bit( Field) ≡
       let x = <0: Field>,
           n<0:35> = - 1
       then if x ≠ 0
              then repeat n ← n + 1 also x ← shift (x,1) until x<0> taeper
            fi next n;
```

```
define Number_of_1_Bits (Field) ∈
       let x = Field, n<0:35> = 0
       then while x ≠ 0 do if x<0> then n ← n + 1 fi next x ← shift (x,1) od next n:
```

```
define qhs ≡ {Q, H, S} ;
```

```
define qhsd ∈ {Q, H, S, D};
```

```
define Reverse-Bits (Field, Count) ≡
       let x = Field,
           y = zero-extend (0, width (Field) ),
           n<0:35> = Count
       then while n > 0
                 do n ← n - 1 also
                    (y ← shift (y,1) v unsigned(low(1,x)) next x ← shift (x,-1))
                 od next
            y;
```

```
define rnd ∈ {FL, CL, DM, HP, US};
```

```
define Skip ≡ pc-nxt-instr ← program-counter + signed (opcode.SKP);
```

```
define Subtract (Minuend, Subtrahend) → Difference, Cout, Overflow next Continuation ≡
    Subtract--With-Carry (Minuend, Subtrahend, 1) → Difference, Cout, Overflow next
    Continuation ;
```

```
define Subtract_With_Carry (Minuend, Subtrahend, Cin) → Difference, Cout, Overflow
                        next Continuation ≡
    let x = Minuend, y = Subtrahend
    then let z
            = c0:x⊃ + c0:¬ y⊃ + zero-extend Kin, width (x) + 1)
        then let Difference = low (width (x), z),
                Cout = z<0>,
                Overflow = (x<0> ≠ y<0>) A (y<0> = z<1>)
            then Continuation;
```

```
define updn ≡ {UP, DN} ;
```

```
define vp ≡ {V, P} ;
```

```
define uuk ≡ {U, UK} ;
```

10 Appendix: The S-1 Assembler (FASM)

10.1 Preliminaries

10.1.1 Instruction and Data Spaces

It is assumed that the user is familiar with the S-l architecture and in particular understands about page table access bits. These are the bits that control what kind of access can be made by the processor to its pages. The output from FASM specifies certain page table access bits for the various output segments. In more detail, an output segment is either an instruction segment or a data segment, corresponding to the page table access bits INSTRUCTIONS and DATA. During an assembly, FASM maintains a number of spaces, each of which is either an instruction space or a data space. Just how many of these spaces there are and how they are mapped into the output segments is described in Section 10.3.

10.1.2 Passes

FASM makes three passes *over* the input file. This is necessary to do a good (but not perfect) job on optimizing the use of PR type jumps. During the first pass, FASM assumes that all jumps will NOT be in PR mode. This causes labels to be set to the maximum possible value that they might attain. During the second pass, FASM attempts to use PR type jumps for jumps in I space, when the jump destination is in I space only and not external. By the end of the second pass all of the labels have been set to their final correct values. During the third pass, the code is actually assembled and output.

10.1.3 Character Set

The character set understood by FASM is the superset of ASCII used at the Stanford Artificial Intelligence Lab. Certain important characters are used by FASM that are not present in standard ASCII. FASM does, however, allow substitutes for these characters from standard ASCII. The following table lists the allowable substitutions:

Stanford ASCII	ASCII	Stanford ASCII	ASCII
⊂ and ⊃	[and]	← (left arrow)	_
↔	?	↑ (up arrow)	^ (caret)
∧ (“and” sign)	&	≠ (not-equal)	#
v (“or” sign)	!	¬ (not-sign)	'

Table 10-1
FASM Character Set

10.2 FASM Formats

10.2.1 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. A single term with no operators is a legal expression. 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.

When an expression is encountered, 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.

10.2.1.1 Operators

The following are the valid operators along with their precedences:

```
+ - * / ∂ % ¬ ! v & ^ # ≠ ↑
1 1 2 2 5 5 5 3 3 3 3 3 3 4
```

The first four are the usual arithmetic operators of addition, subtraction, multiplication and division. Plus is ignored as a unary operator. Minus may also be used as a unary operator. ∂ is equivalent to - as a unary operator, % is a unary operator which forces the entire expression to have the register attribute. The next four operators are Boolean. They are logical negation, inclusive or (either ! or v), logical and (either & or ^) and exclusive or (either # or ≠). The last operator is the logical shift operator. A↑B has the value of A left-shifted B bits. A logical right shift is performed if B is negative. 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; they are used to indicate index registers.) 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.

10.2.1.2 Terms

A term in an expression may be a number, a symbol, a literal, a text constant or a value-returning pseudo-op.

10.2.1.2.1 Numbers

A string of digits is interpreted as a number in the current radix unless it ends in a decimal point in which case it is assumed to be a decimal number. The radix is initially base 8 (octal) and may be changed with the RADIX pseudo-op. A 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.

10.2.1.2.2 Symbols

A symbol is a one to twelve character name made up from letters, numbers, and the characters . and $. (A symbol may actually contain more than twelve characters, but all characters after the twelfth are ignored.) 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. 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.*

Just the single character . is a symbol whose value is the current location counter. It is either an IVAL or a DVAL, depending upon which space is currently being assembled into. The symbols RTA and RTB have been predefined to have the values %4 and %6 respectively. Register values are in the range $0..37_8$. 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.)

When a symbol with the register attribute appears in an expression, then the expression is a register expression and itself 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.

10.2.1.2.3 Literals

A literal is any set of assembler statements enclosed in [] (called *square brackets).* A literal directs the assembler to assemble the statements appearing insided the square brackets and store them at some location other than where the current location counter points. The value of the literal

for use in an expression is the address where the first single-word 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 have assembled some statements already.

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- 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.

10.2.1.2.4 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 quarter-word. 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.

10.2.1.2.5 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 for a description of the values they return.

10.2.2 Statements

10.2.2.1 Statement Terminators

How a statement is terminated will depend upon the exact type of statement. In general, a statement is terminated with a line-feed, a ↔, or' a semicolon that begins a comment that terminates at the next line-feed. Some statements, like symbol definitions, can also be terminated with a space or a tab.

10.2.2.2 Symbol Definition

Symbols may be defined to have specific values 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 t. 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 typeout. 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 most any separator, including space and tab. Therefore it is possible to have more than one assignment statement per line, or have an assignment statement on the same line with other statements.

10.2.2.3 S-l Instructions

An S-l instruction is a statement that can cause the assembly of one, two or three single-words. It is made up of an opcode with modifiers followed by a list of operands.

10.2.2.3.1 Operands

An operand may be in any one of the following formats: (in the following |.....| may be used in place of ⊂.....⊃).

expression

This may be a register expression or not. If so, it is assembled as register direct, otherwise as an absolute address. If the operand is a hop, skip, or jump destination, then the difference between the expression and the location counter (.) is used as the signed displacement, if possible.

?expression

This assembles as either a short or long constant depending upon the value of expression. It is dangerous to use an as yet undefined symbol in the expression, as the assembler might decide to switch from one length to another, which would confuse the rest of the assembly. If the expression is in the range -3‘2 . . 31 (decimal) the assembler will generate a short constant. If not, it will generate a long, sign-extended constant. A data word (see below) may not appear in the expression unless it is enclosed in brokets.

#expression

This assembles as a short constant. It doesn’t matter if the expression has a register value or not. It is an error if the expression cannot be expressed as a short constant.

expression(register **expression)**

This is a short index. The expression inside the parentheses must have a register value. If the internal assembler switch BADRSI is off (the default state), the expression before the parentheses is assumed to be a number of single-words and must be in the range -32 . . 31 (decimal). If the switch is on, it is assumed to be a number of quarter-words and must be divisible by 4. The result of division by 4 must be in the range -32 . . 31 (decimal). If the expression before the parentheses is omitted, zero is assumed.

#⊂expression⊃

This is a format-l long constant (right justified with zero fill).

#⊂expression ↔ 0⊃

This is a format-3 long constant (left justified with zero fill). The spaces around the ↔ are optional.

```
#c!S↔expression>
```

This is a format-2 long constant (right justified, sign extended).

```
#cexpression⊃(register expression)
```

This is an indexed constant. The first expression is the constant and the second expression is the index register (which may be zero but may not be omitted).

```
c!P@ expl (exp2)⊃(exp3(exp4))↑exp5
```

This is the general form of an extended word. The !P and @ are optional and cause the P bit and I bit respectively to be set in the extended word. If exp2 is present, the extended word is in variable-based format (V-bit=1); otherwise it is in fixed-based format. Expl is the base or signed displacement and is considered a quarter-word address (note that in short indexing, the corresponding expression may be a signed single-word value). Everything after the ⊃ is optional. If nothing is there, a short operand (SO) of short constant 0 is generated. If something is there, the outer set of parentheses must be present. These are mnemonic, indicating that the SO that's inside the parentheses is fetched. The SO inside the parentheses may be either a short index (which requires the use of another set of parentheses as described above) or register direct (in which case no other parentheses are used) which must evaluate to a register value. Finally, if the ↑exp5 is present (which it may be even if (exp3(exp4)) is omitted), the value of exp5 is used as the S field of the extended word.

```
!expression
```

This format forces the operand to have the value of the low 12 bits of expression. No extra word will be assembled for an extended word in the case that the value has the 4000_8 bit on. It is possible with this format to generate illegal instructions. It is meant for hand or program patching of code.

'Here are some examples:

```
c!P Table(R3) ⊃(-3(SP))↑2
c Table ⊃(R5) or c Table(R5)⊃
c@ Table⊃((SP))
```

10.2.2.3.2 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. So, for example,

```
SLR.8
```

is different from

```
SLR.10
```

It is also possible to use an already defined symbol as a modifier. For example, if A has been defined by A←%5 then SLR.A assembles the same way as SLR.5 does. Note that an expression may NOT be used in place of a modifier. For example, SLR.4+1 is not permitted in place of SLR.5. 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 M 1 is a legal modifier for BNDTRP and takes precedence over the lookup of the symbol M 1.

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.

10.2.2.3.3 Instruction Types

There are five basic S-l instruction types, SOPs, JOPs, XOPs, TOPs, and HOPs. For the assembler, they differ as to the number and interpretation of operands.

‘An SOP is a two operand instruction with a skip destination. The skip destination is just like a third operand, and should evaluate to the quarter-word address of the instruction that is to be skipped to. Both of the operands must be present. 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 of the location of the SOP.

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 OPI 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 (it *even* takes a whole extra pass through the source file just for this). For example,

```
IJMPZ.NEQ %2,LOOP
```

assembles a jump to location LOOP.

An XOP is a two operand instruction, one of which must be specified. If exactly one 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 (7.0). For example,

```
INC COUNT
```

assembles the same as

```
INC COUNT,COUNT.
```

A TOP is a three operand instruction, where one of the operands is restricted. There are 4 possible combinations for the operands, involving use of RTA and RTB. If only two operands are given, then T=00 is used (DEST=S1=OP1). If the first operand's value is RTA, then T=10 is used (DEST=RTA,S1=OP1). If it is RTB, T=11 is used (DEST=RTB, S1=OP1). If the second operand's value is RTA, then T=01 is used (DEST=OP1,S1=RTA). Any other format is illegal. For example,

```
ADD RTA,FOO,BAR
```

assembles a T=10 TOP.

An HOP is a one operand instruction. It takes a jump destination like a JOP and assembles it as a pc relative single-word offset directly into the OD1 and OD2 fields. No extended words are ever used. This instruction type is specifically for the JPATCH instruction, which can jump to PC-<2^{24}>*4 through PC+<2^{24}-1>*4. Note that this is not the full virtual addressing range of the S-1. This instruction, therefore, is not recommended for branching. Use JMPA instead, which can jump to any location in the address space. JPATCH is provided so that a debugger can "patch" an instruction location and clobber only one single-word.

10.2.2.4 Data Words

A data word is much like a short index in that it can specify indexing. For example,

```
-1
30,,7                ;A single-uord uith 30 in its left half-uord
                     ; and 7 in its right half-word
@-4(SP)
(TT)[   1
        2
        4   ]
```

are all data words. If indexing is used, then the value in the register field is assembled into bits <1:5> and the value of the expression surrounding the index is assembled into bits <6:35>. If indexing is not used, then the value is stored in the entire word, bits <0:35>. If an @ is present, the sign bit of the word is turned on. This is the P-bit in an indirect word. The word "surrounding" is used because of the following effect:

```
-1(TT)[A
       B
       C 3
```

will assemble with TT in the index field and with the address of the literal -1 in the address field. This is useful if TT for example ranges from 1 to 3.

Data words may be used anyplace where an instruction might have been used. They may be used in long constants and in literals. They are legal inside any broketed expression.

10.3 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 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 DVALs 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 DPACE 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 DPACE 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 or 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.

10.4 The Location Counter

The location counter is a symbol internal to the assembler that has the value of the quarter-word address where the next word will be assembled. It has either the IVAL or DVAL attribute depending upon the use of the IPAGE, DPAGE, ISPACE, DSPACE and XSPACE pseudo-ops. Initially it has the IVAL attribute and for an absolute assembly, it has initial value 10000_8. For a relocatable assembly it has initial value 0. The symbol . may be used to reference the location counter. It cannot be defined with an assignment statement or used as a label.

10.5 Pseudo-ops

The following is a list of all the pseudo-ops in alphabetical order. Wherever the construct ⊗ text ⊗ is used, the ⊗ represents the first non-blank, non-tab character appearing after the pseudo-op and text is all of the characters between the matching pair of these characters.

.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.

.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 FOO.BAR[P,PN]>
```

.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.

.LENGTH ⊗ t e x t ⊗

Has the value of the length of the string *text*. A CRLF counts as one character.

.QUOTE ⊗ t e x t ⊗

Legal only inside a macro definition. It allows the assembler to see *text* without scanning it for a DEFINE or a TERMIN.

. SWITCH swname1,swval1,swname2,swval2, ...

Sets internal assembler switch "swname1,2,..." to the value in the expression "swval1,2,...". The currently existing switches are:

BADRSI	If set, all **short** indexes are assumed to be quarter-word addresses and **must be** divisible by four. Otherwise a short index is considered a single-word index.

ABSOLUTE

Forces the assembly to be absolute.

ASCII ⊗ t e x t ⊗

Assembles text as ASCII characters into consecutive quarter-words, padding the last used single-word 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 ⊗ t e x t ⊗

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 ⊗ t e x t ⊗

Same as ASCII except that it guarantees that at least one null character appears at the end of the string.

ASCIZV ⊗ t e x t ⊗

Is the same as ASCIIV except it does ASCIZ.

AUXPRX ⊗ t e x t ⊗

The *text* is output to the auxiliary file. An error message is generated if no auxiliary file is open.

AUXPRV ⊗ t e x t ⊗

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 express i on

Adds expressions4 to the location counter. That is, the expression is the number of single-words 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 a no-op 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 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, < conditionaliy assembled text > rest of program
IFN2, < condi tionai iy assembled text > rest of program
IF3, < condi tionai iy assembled text > rest of program

IFN3,< conditional iy 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 expression,< conditionally assembled text > rest of program
IFN expression,< conditional iy assembled text > rest of program
IFL expression,< conditionally assembled text > rest of program
IFG expression,< conditional ly assembled text > rest of program
IFLE expression,< conditionally **assembled text > rest of program**
IFGE expression,< 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, c
        SKP. NEQ A, "<"          ;> MATCHING BROKET
        JMPA FOUNOLESS
>;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>,<>,< . . . > . . .
```

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

Is a no-op 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.

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 express i on

Sets the location counter to the specified quarter-word 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 ⊗ t e x t ⊗

Prints *text* on the console.

RADIX expressi on

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.

10.6 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-IO 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 other assmeblers, however. Macro expansion in FASM is performed at the "read-next-character" level whereas in other assemblers it is done at symbol lookup time during expression evaluation. Due to this difference, in FASM, macro expansion 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.

10.6.1 Macro Definition

Macros are defined using the DEFINE pseudo-op which has the following format:

```
DEFINE macroname argument list
  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.

10.6.1.1 The Argument List

Basically, the argument 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 arguments is in the range 0. . 64 . The macro argument list is terminated by either a ; or a CRLF.

Each macro argument has certain attributes associated with it. In FASM these attributes are *balancedness, gensymmedness,* and *parenthesizedness*. From now on, it shall be said that an argument is or is not *balanced, is* or is not *gensymmed,* and that certain pairs of parentheses can or cannot *parenthesize* an argument. If an argument isn't balanced or gensymmed then it is said to be *normal.*

Argument attributes are specified by enclosing a string of characters in double quotes

preceding an argument in the argument list. The attributes specified by that string are "sticky", that is, they apply to all following arguments until the next such string is specified. The characters B and G may appear in the string to indicate that the argument is to be balanced or gensymmed respectively. There are four parenthesis pairs, namely: (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 argument. 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 O! appears, then there will be no concatenation character. Only the last statement made about the concatenation character will apply.

At the beginning of the argument list, the attributes have the following defaults: ! is the concatenation character, arguments are neither balanced nor gensymmed, and any pair of parentheses may be used to parenthesize an argument. Whenever an attribute string is encountered, the previous set of attributes are forgotten and the new one applies to future arguments 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, "[0!" B
```

With these definitions, MAC has no arguments and has ! for the concatenation character. MAC1 has three normal arguments, A, B and C with ! for the concatenation character. MAC2 has two normal arguments A and B, a gensymmed argument C and uses ' as the concatenation character. MAC3 has a balanced argument A, for which () and [] can be used as parentheses and a normal argument B for which [] can be used as parentheses. MAC3 has no concatenation character.

10.6.1.2 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 parameter 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 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. .QUOTE is like an ASCIZ statement, 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
TERN1 N
```

will define MAC's body to be

```
....                    ;how to put a DEFINE in a comment
```

10.6.2 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.

10.6.2.1 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 argument is not assigned a string, 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, semicolon, or the argument terminator for the last argument. 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.

10.6.2.2 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 ail 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 p'arenthesized argument.

If the first character is a comma, then the argument is the null string.

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 ail characters from the first character until, but not including, a comma or macro call terminator. If the terminator is a comma, it is thrown out; a macro call terminator, however, will be kept to terminate the macro call.

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 closed parenthesis will terminate the argument and the call. If it is a parenthesized call, the closed 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.

10.6.2.3 Special Processing in Macro Argurnents

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 from 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 a pair of ' (backquote) characters surround an expression, 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 backquotes 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 backquoted expression. As in the case of \ this can occur in non-parenthesized macro arguments or in expression evaluation. The ' characters themselves are thrown out.

Following are some examples of the use of these features:

```
X←←1    FOO 'X':    JMPA F001
```

will assemble as

```
F001:    JMPA F001
```

If FOO was a macro name, it would have been expanded in the previous example. This could be inhibited with:

```
\FOO\ 'X':          JMPA F001
```

Next consider:

```
DEFINE MAC
X←←X+1
X!TERMIN

FOO 'MAC':
```

will define the label F002 while incrementing X to be 2. The next time FOO'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
BAR←←BAR+1
A*BAR
TERMIN

MAC 'BAR'
```

will call MAC with the current value of BAR. Without the backquotes, the string BAR would be passed to the macro and used where “a” appears which is after BAR is incremented.

11 Appendix: S-1 Formal Description Syntax

11.1 The S-l Architecture Notation

The S-i Architecture Notation is a LISP-like language. It has a modified LISP syntax. There is an interpreter/debugger which executes procedures in the language, and a pretty-printer which takes the LISP-like code and produces a file which is a version of the code rendered in an ALGOL- or PASCAL-like syntax. This is the format that appears under the heading "Formal Description" with each instruction description, and in other places as well. In this description we shall exhibit the LISP-like and PASCAL-like notations side-by-side.

The basic data objects in the language are *numbers* and *bit fields.* A *number* is simply a signed integer. A *bit field* is an object with definite *width* (the number of bits), *contents* (values for each of the bits), and *alignment,* which is a number for the leftmost bit, following bits have successively higher integer indices. (Internally, bit fields are represented as S-lists of integers (content width alignment). For many purposes, one can think of an integer as a bit-field in two's-complement form with half-infinite width, sign-extended to the left.)

An integer can be notated in the ordinary decimal notation, with an optional sign. It can also be notated in octal by preceding it with a "*".

Examples: 12 + 14 -10 *7777 e-43

A bit field can be notated in the "LISP" syntax by writing <j:k>n, where j, k, and n are ail numbers. This specifies a field k-j+1 bits wide, aligned so that the leftmost bit is bit number j, and whose contents are the low k-j+1 bits of the two's-complement representation of n. In the "PASCAL" syntax this is written as n<j:k>.

There are also one-dimensional arrays of bit fields, called *memories.* These cannot be constructed dynamically, but must be pre-declared (this is discussed later).

11.2 Symbols

Non-numeric tokens, or ***symbols***, occur in four distinct varieties: *constant **symbols**, substitution variables, identifiers,* and ***keywords***. They are distinguished by their spelling, and in the "PASCAL" syntax also by the use of special fonts:

Type	"LISP" syntax	'*PASCAL" syntax
constant	all upper case	all upper case, gothic font
substitution	capitalized, or leading %	*italic font, usually capitalized*
identifier	*all lower* case	all lower case, gothic font
keyword	leading $	boldface

Table 1 l-l
Symbol Types and Fonts

Actually, only the first two characters of the symbol are examined in performing this classification. The letters A-Z and digits O-9 are considered to be capitals, and all other characters, even special characters such as "-" and "*", are considered to be lower-case. A "capitalized" symbol is one whose first character is upper-case and whose second is lower-case.

When a "LISP'-syntax symbol is rendered into "PASCAL' syntax, a leading $ or % is elided (because the font carries the necessary information). Also, any "-" characters are changed to "_" characters ("-" is the standard LISP "break" character, while "_" is the standard "break' character for PASCAL-like languages.)

Examples of PASCAL syntax:

constant	Q H S D LF RT MODE
substitution variable	*Address Extended-Word p foo Extended-word* ***
identifier	program-counter od x n
keyword	if while case

Examples of LISP syntax:

constant	Q H S D LF RT MODE
substitution variable	Address Extended-Word %p %foo Extended-word ***
identifier	program-counter od x n
keyword	$if $while $case

*** Note: 'Extended-Word" and "Extended-word" are two ***different*** substitution variables. The first is the preferred form.

Constant symbols are used in much the same way as scalar data type elements ark in

PASCAL: to provide constant values for control purposes with a manifestly meaningful name. Substitution variables are similar to Algol-style call-by-name parameters, and will be discussed below. Identifiers are ordinary call-by-value variables; their values may be numbers, bit fields, *or* constant symbols; they are also used as names for memories. Keywords are used to identify certain control constructs, and as noise words.

In presenting the "PASCAL" syntax here, we will use font changes in lieu of the capitalization and leading "$" conventions. Thus, we will write:

"if okay then *Operation else Error* **fi"**

with "if", "then", "else", and "fi" in boldface; "okay" in gothic letters; and "Operation" and "Error" in italics to mean

"$if okay $then Operation $else Error $fi"

in the LISP syntax.

11.3 Forms

In the "LISP" syntax, as in real LISP, nearly all forms except numbers and symbols are written as a list of forms enclosed in parentheses. Such a form may mean one of three things:

(1) If the first element is an identifier, then it is *a procedure call* or *function call.* The identifier is the name of the function, and the other elements of the list are the arguments, which are evaluated before the function is called.

Examples: (shift x n) (+ y z) (> a b)

(2) If the first element is a keyword, then it is a *special form* , a control construct of some kind.

Example: ($while x $do y)

The keyword "$while" signifies a special form, The keyword "$do" is a (required) noise word.

(3) If the first element is a substitution variable (or a constant symbol) with a global macro definition (which has not been shadowed by a local definition -- never done in practice!), then it is a macro call,

Example: (Calculate-Operand 2 $next Operation) The symbol "Calculate-Operand" signifies a macro call, with the parameters "2" and "Operation", and the noise word "$next".

(If the first element is a substitution variable with some local binding, or a global binding which is

not a macro definition, then its definition is substituted in and the three-way classification is tried again. See the description of macros below.)

11.4 Primitive Functions and Other Identifiers

The language provides a number of identifiers with function definitions which are useful for manipulating bit fields. Recall that the arguments to all functions are fully evaluated before invoking the function on the result. Some global identifiers are also predefined with useful bit-field values. (In the descriptions that follow, Greek letters are meta-variables which range over forms. The "LISP" syntax is shown on the left, and the "PASCAL" syntax to the right. If the syntax is common to both, as in the case of symbols, they are shown centered.)

(+ α β)	addition	α + β
(- α β)	subtraction	a - β
(A α β)	logical and	a A β
(v a β)	logical or	α v β
(⊗ a β)	logical xor	α ⊗ β

$0 $false

These identifiers initially have as value a one-bit field containing a 0.

$1 $true

These identifiers initially have as value a one-bit field containing a 1.

N.B. $0 and $1 are usually used with the bit-field concatenation construct -- see below.

Table 11-2
Arithmetic and Logical Functions

These arithmetic and logical operators will accept either integers or bit fields. If both are integers, then an integer results. If one is an integer and the other a bit field, then the integer is first converted by two's-complement truncation to a bit field of the same width as the other -argument. If both are bit fields, they must be the same width, or an error will result; the value is a bit field of the same width, aligned so that the high bit is bit number zero. In no case is overflow detected.

(¬ α) logical not ¬ α

If a is an integer, the result is an integer. If α is a bit field, the result is a bit field of the same width, aligned so that the high bit is bit number zero.

N.B. There is no unary minus. However, if one writes "(- 0 α)", then the pretty-printer will render it as "-α" rather than as "0 -α".

(< α β)	signed less than	α < β
(> α β)	signed greater than	α > β
(≤ α β)	signed less than or equal	α ≤ β
(≥ α β)	signed greater than or equal	a ≥ β

If either argument is a bit field, it is first converted to an integer by considering it as a signed two's-complement representation. If both arguments are bit fields, they must be the same width (for error-checking purposes). The two integers are compared, and the result is a bit field exactly one bit wide, whose content is 1 if the specified relation holds, and 0 otherwise.

(= α β) equal α = β
(≠ α β) not equal α ≠ β

These operators compare their arguments for equality. The arguments may be two integers, two bit fields of the same width, an integer and a bit field (in which case the latter is sign-extended, or the former is truncated -- the two interpretations are equivalent), or two symbolic constants.

(signed a) sign extension signed(a)

If the argument is a integer, that integer is returned. If it is a bit field, an integer produced by sign-extending the bit field "to infinity" is returned.

(unsigned a) unsigned interpretation unsigned(a)

If the argument is a integer, it must be non- negative (otherwise an error occurs), and is returned. If it is a bit field, an integer produced by zero-extending the bit field "to infinity" is returned.

(sign-extend a β) sign extension sign-extend(a, β)

The argument β must evaluate to a non-negative integer, or to one of the symbolic constants Q, H, S, D, or A (which mean 9, 18, 36, 72, and 30, respectively). The argument a must evaluate to a bit field whose width is no greater than β. A field β wide containing the same signed value as a is returned, aligned so that the left bit is bit 0.

(zero-extend a β) zero extension zero_extend(α, β)

The argument β must evaluate to a non-negative integer, or to one of the symbolic constants Q, H, S, D, or A (which mean 9, 18, 36, 72, and 30, respectively). The argument a must evaluate to a bit field whose width is no greater than β. A field β wide containing the same unsigned value as a is returned, aligned so that the left bit is bit 0.

(low α β) extract low bits low(α, β)

The argument β should produce a bit field, and a should produce an integer, bit field, or one of Q, H, S, D, or A. The unsigned value of a specifies how many bits should be extracted from the low end of β (the width of β must be no less than specified by α). The result is aligned so that the leftmost bit is bit 0.

(high α β) extract high bits high(a, β)

The argument β should produce a bit field, and a should produce an integer, bit field, or one of Q, H, S, D, or A. The unsigned value of a specifies how many bits should be

extracted from the high end of β (the width of β must be no less than specified by a). The result is aligned so that the leftmost bit is bit 0.

(shift a β) shift field shift(α, β)

The argument a should produce a bit field, and should produce an integer or bit field. A field is returned which is as wide as the field a, and which has the *same alignment*, with contents equal to those of a shifted a distance β, where positive β is to the left, and the shift loses bits without any overflow detection and shifts in zero bits.

(realign a β ϵ) realign field realign(α, β, ϵ)

The arguments a and β must produce numbers (not bit fields), and ϵ must produce a bit field, in in which case the width of ϵ should be β-α+1; or an integer, in which case the integer is truncated without overflow checking to a signed field of that width. The result is a copy of ϵ realigned so that the leftmost bit is bit number a.

N.B. This is almost never used explicitly by the programmer, but is used implicitly by control constructs which bind identifiers, such as $let (q.v.). It is also used by the construction <α:β>n -- see below.

(extract-bits a β ϵ) extract subfield ϵ<α:β>

<α:β>ϵ

The arguments a and β must produce numbers (not bit fields), and ϵ must produce a bit field. A bit field is returned of width β-α+ 1, whose contents are those of bits α through β of ϵ. The result is aligned so that the leftmost bit is bit 0 (not bit a!). It is permitted to abbreviate the field specifier "<j:j>" to simply "<j>", thus selecting a single bit. Through a bit of clever programming, the "LISP" syntax has an alternative form <α:β>ϵ, which is similar to the "PASCAL' form, except for putting the operator up front, as with most LISP constructs. This syntax enforces a rule that a and β, the forms themselves, must be explicit numbers, and not any old numeric- valued expression. This is to force the programmer to use the operators low, high, and shift when variable fields are involved. Another twist is that if in <α:β>ϵ, the form ϵ is explicitly a number, then that expression is parsed as (realign a β ϵ) rather than as (extract-bits α β ϵ) -- see above.

(word α β) β[α]
[α]β

The form a must produce a bit field or a non-negative number; the form β must produce an array (memory). The unsigned value of a must lie within the declared range of subscripts for **the** array. The bit field selected from the array β by a is the result. Through a bit of clever programming, the "LISP" syntax has an alternative form [α]β, which is similar to **the** "PASCAL" form, except for putting **the** operator up front, as with most LISP constructs. A special twist of the [...] syntax is that if the form β is explicitly a substitution variable, then the form [α]β is not parsed into (word a β), but into (β[a]), which is a macro call. In this way one can make a macro call look like **an** ordinary array reference.

(concatenate al a2 . . . aj aj aj . . . aj . . . an) concatenate bit fields
⊂α1||a2|I...||k∗αj|I...||αn⊃

By special arrangement, concatenate can take any number of arguments. The bit fields are concatenated together in order, leftmost argument being leftmost in the result field. The width of the result is the sum of the widths of all the arguments. The result is aligned- so that the leftmost bit is bit 0. The "LISP" syntax **has an** alternative notation identical to that of the "PASCAL" syntax. The arguments are enclosed in "⊂⊃" and seperated by "||". If before any argument the phrase "k∗" appears, where k is an explicit number, it is as if the argument had been written **that many** times. This is often **used** in **con** junction with $0 and $1.

Example: ⊂6∗0|| program-counter ||2∗0⊃

11.5 Special Forms

(In the descriptions that follow, Greek letters are meta-variables which range over forms. The "LISP" syntax is shown on the left, and the "PASCAL" syntax to the right.)

```
($if α β ϵ)                    if α then β else ϵ fi
```

The form α must evaluate to a bit field exactly one bit wide. (Such bit fields are typically the result of predicate operators such as "<".) If that bit is a 1, then β is evaluated, and otherwise ϵ is evaluated.

Example:

```
($if (> s 1 s2) s1 s2)          if s1 > s2 then s1 else s2 fi
```

```
($case a                        case a of
    (set1 β1)                     set1: β1;
    (set2 β2)                     set2: β2;
    ...                           ...
    (setn βn))                    setn: βn;
                                end
```

The form a is evaluated, and the resulting value should occur in one of the sets. If it is found in setj, then βj is evaluated,

A "set" may be any one of the following:

[a] a symbolic constant or an integer

[b] (integer $to integer) integer..integer

[c] (x 1 x2 . . . xn) x 1, x2, ... xn

where each xj is a set of type [a] or [b]

Example:

```
($case reg                      case reg of
    ((0 1 2 (4 $to 31)) Foo)      0,1,2,4..31: Foo;
    ((3) Bar))                    3: Bar;
                                end
```

```
Wet ((v 1 = a1)                 let v1 = a1,
    . (v2 = α2)                     v2 = a2,
     ...                            ...
     (vn = an))                     vn = an
    β)                          then β
```

The forms a1, ... an are all evaluated; then their values are all *simultaneously* assigned to the identifier specifications v 1, ... vn, which constitute new local variable bindings. Finally, the form β is evaluated in this new environment. An identifier specification aj can be just an identifier, or it can be of the form (extract-bits α β ϵ), where α and β are integers and ϵ is an identifier. In the latter case, the result of (realign a β aj) is what is assigned to the identifier ϵ. This allows the precise width and alignment of the newly-bound identifier to be specified.

Example:

($let ((x = Field)	let x = Field,
(<0:35>n = 0))	n<0:35> = 0
Continuation)	*then Continuation*

($while α $do β) **while a do β od**

The form a is evaluated, and should result in a one-bit field. If this field contains a 1, then β is evaluted, the result thrown away, and the process is iterated. If ever evaluating a produces a 0, then a '*garbage" result is returned, which is illegal to use for any operation on bit fields.

($repeat *a* $until β) **repeat a until β taeper**

The form a is evaluated, and the result is thrown away. Then the form β is evaluated, which should produce a one-bit field. If this field contains a 0, then the process is iterated. If ever evaluating β produces a I, then a "garbage" result is returned, which is illegal to use for any operation on bit fields.

($do-forever a) **do forever *a* od**

The form a-is evaluated and the result thrown away for **an** indefinitely large number of iterations.

($prefix x a) **prefix(x,α)**

x must explicitly be **an** identifier. **a must** also be an identifier, possibly after resolution of substitution variables. The effect is as if a single identifier had been written in place of the #prefix-form, whose name is that of x, followed by a **"-"** ("LISP" syntax) or a **"_"** ("PASCAL" syntax), **followed** by that of a.

Example: ($prefix address Op) is the same as address-opl, **assuming** that the substitution variable Op has the substitution value "opl".

($next a β ϵ . . . n) a **next β next ϵ next . . . next π**

The forms a, β, ϵ, ... π are evaluated in order. The results of all but the last are thrown away. The result of the last form is the result of the **$next-form.**

($also *a* β ϵ . . . π) *a* **also β also ϵ also . . . also π**

The forms a, β, ϵ, ... π are evaluated in an *arbitrary* order. No defined result is produced.

(← α β) α ← β

This is the assignment statement. It is very complicated because of the variety of forms permitted **on** the left-hand side:

identifier
identifier< j:k>
array[n]
array[n]< j:k>
$let . . . $then a
#if π **$then** al **$else** a2 **$fi**

where *a, al, a2* are themselves forms permissible on the left-hand side of "←". (The last two cases are useful when a macro is used to compute which identifier is to be assigned to.)

11.6 Global Register and Memory Declarations

Local identifiers can be declared using the let **statement.** Globally available identifiers and arrays can be declared using the "toplevel" register and memory statements. (There is no way to locally declare an array.)

The general form of a **register declaration** is:

```
($register < j:k>identifier)        register identifier<j: k>;
```

This defines a globally available register whose width is k-j+1 and whose leftmost bit is bit number j. The bit range limits j and k muse be integers.

Examples:

```
($register <0:35>user-status)       register user_status<0:35>
@register <0:27>program-counter)    register program_counter<0:27>
```

The general form of a $memory declaration is:

```
($memory <j:k>[m:n]identifier)      memory identifier Em: nl <j:k>;
```

This defines a globally available array of bit fields. Each bit field is k-j+1 bits wide, with the leftmost bit being bit number j. There are n-m+1 such bit fields in the array, numbered from m through n.

Examples:

```
($memory <0:35>[0:511Iregister-file)        memory register-file [0:5 11 ]<0:35>
($memory <0:35>[0:4095]physical-memory)     memory physical-memory [0: 40953 <0:35>
```

11.7 Macros and Substitution Variables

A macro may be thought of as a procedure which takes all of its arguments "by name", in the Algol sense (however, as we shall see, the rules for scoping variables are different from those of Algol). It may also be thought of as a piece of text to be used in place of a call on that macro, with specified arguments substituted into specified places in the macro definition.

A macro definition is a "top-level" declaration, such as the **register** and **memory** declarations. It has the general form:

(≡ <prototype> <body>) **define** <prototype> ≡ <body>;

Whenever an instance of the prototype (a macro call) is seen as a form of the language, a copy of the body may be substituted for it, possibly with alterations determined by matching the macro call against the formal prototype in the definition.

The simplest type of macro has *no* parameters. It is merely an abbreviated name for a piece of text which is evaluated whenever the name of the macro is encountered. The name must be a substitution variable; the body may be any valid form. For example, with this definition:

(≡ Jump (← pc-nxt-instr jump-address))
define Jump ≡ pc_nxt_instr ⇐ jump-address;

then writing "Jump" as a form would be entirely equivalent to writing "(← pc-nxt-instr jump-address)" (LISP syntax) or "pc,nxt,instr ⇐ jump-address" (PASCAL syntax).

In the more general case, a macro prototype may be an arbitrarily complicated list structure, provided the first element of the outermost list level is a substitution variable (which is the name of the macro). A call on this macro must be a similar list structure, with the first element of the outermost list level being the name of the macro. To **substitute the macro body for the call, one** *matches* the call against the formal prototype. Wherever a substitution variable occurs in the formal prototype, the corresponding expression in the call is matched to it. If an identifier or keyword occurs in the formal prototype, that same identifier or keyword *must* appear in the macro call, as a "noise word". When the match has been completed, then the body may be used, with the provision that any occurrences of the matched-against substitution variable parameters in the body be replaced by the matching **expressions** in the call.

For example, consider:

(≡ (Memory-Address-Is-a-Register Address)
(∧ (= <0:22> Address 0) (¬ Block-Memory-Address-Is-a-Register)))

define *Memory-Address-Is-a-Register (Address)* ≡
(Address<0:22> = 0) ∧ ¬ Block_Memory_Address_Is_a_Register;

Then if *one* were to write:

(Memory-Address-Is-a-Register address-op 1)
Memory-Address-Is-a-Register (*address_op1*)

it would be exactly the same as writing:

(A (= <0:22>address-op1 0) (¬ Block-Memory-Address-Is-a-Register)))
(*address_op1*<0: 22> = 0) A ¬ *Block_Memory_Address_Is_a_Register*;

because the parameter "add ress-op 1" is substituted for occurrences of "Address" in the body of the macro definition.

There are three extra features which can be used in the formal prototype to control the matching.

[1] If the form
($vel a b c . . . z) a|b|c I...|z
appears in the-formal prototype, then the match succeeds if the corresponding part of the call is **an** identifier or keyword which is in the list a, b, c, ..., z.

[2] If the form
(ϵ x y) xϵy
appears in the formal prototype, where "ϵ" is actually the character epsilon, **then** the form y must be (or resolve via substitutions to) a form
($set al a2 . . . an) {a 1 ,a2,...,an}
Then the match succeeds only if the corresponding part of the call is in the set (whose elements may be constant symbols, integers, or integer ranges "(m $to n)" ("LISP,' syntax) or "m..n" ("PASCAL' syntax), as with case sets). If this is true, then that same part of the call is matched against x (which is normally a substitution variable).

[3] If the form
(= x α) x=α
appears in the formal prototype, it is just as if x itself had been written (where x must be a substitution variable), *except* that if the macro call has too few elements at the list level containing the = construction, so that no part corresponds to x, then the match still succeeds, with x corresponding to α. In this way a serves as a "default value" for x.

Macro calls in the "LISP" syntax all look pretty much alike, according to the **above rules.** To permit some syntactic variety in the "PASCAL" syntax, special cases of the "LISP" syntax are defined to pretty-print in special ways.

The *standard syntax* for macro calls (and prototypes) is used when no keywords occur at the top list level of the call, and one of the special formats described below is not involved. In this case the macro name is printed, followed by a left parenthesis, followed by all the arguments separated by commas, followed by a right parenthesis:

(Reverse-Bits Field Count) *Reverse- Bits(Field, Count)*

If no parameters are present, then just the name of the macro is printed, without parentheses. Thus the two "LISP" forms "Jump" and "(Jump)" are both rendered in the 'PASCAL" syntax as simply "Jump". It is recommended that the second "LISP" form be avoided.

If keywords (boldface) are present in the call, then the rule is to first write the macro name and all parameters up to the first keyword as a standard call; then print the keywords and other following parameters in order, using a comma as a separator in case two non-keyword arguments are adjacent:

(Add s 1 s2 $→ sum c ov $next <more>)
Add (s1, s2) → **sum, c, ov** next More

If a macro call has exactly four elements, and the second and fourth are "[" and "]", then the call (a [β]) is pretty-printed in the form "α[β]". (Recall that in the "LISP" syntax the expression "[β]α" is parsed as (a [β]) iff a is explicitly a substitution variable.)

Example:

```
(≡ [Number]Index-reg
   ($if (= Number 3)
       ⊂6*$0||Program-Counter||2*$0⊃
       <0:35>[Number]Register))

(≡ (Index-reg /[ Number /I)
   ($if (= Number 3)
       (concatenate $0 $0 $0 $0 $0 $0 Program-Counter $0 $0)
       (extract-bits 0 35 (Register /[ Number /])))))

define ≡Index-reg[Number]
        if Number = 3
           then ⊂6*0|| Program-Counter ||2*0⊃
           else R [Number]<0:35>
        fi;
```

'Here we have, in the middle expression, expanded out all the funny syntactic forms into regular "LISP-like" syntax to show explicitly the interpretation involved. (The character "/" is used to "quote" the following character so that it will be interpreted as a letter rather than a special syntactic character.)

As a special case, it is permitted to use a constant symbol as the name of a macro. This is usually, but not always, used in conjunction with one of the following special formats.

If the second element of the call in the "LISP" syntax is "$i", then *instruction* macro format is used. The name and the arguments after the "$i" are printed in order, separated by ".".

(MOV $i S D) MOV.S.D

If the second element of the call in the "LISP" syntax is "⊗", then *selector* macro format is used. There must be only one argument after the "⊗"; it is printed, then a ".", and then the name of the macro.

(MODE ⊗ od l) od l.MODE

If the name of the macro is XOP, JOP, TOP, *or* SOP, then a very funny format is used.

11.8 Comments

Comments may be inserted in the "LISP" syntax in the usual way: a comment begins with a semicolon, and is terminated by the end of the line. Such comments are rendered into the "PASCAL" syntax in one of two ways. If the comment begins with more than one semicolon (usually three and a space are used), then the form "$comment<the comment>;" is used. Such comments are normally used outside of other forms. If the comment begins with only one semicolon, then the form "(* <the comment> *)" is used; the comment is right-justified (thrown against the right-hand margin). Such comments can be put in most reasonable places within a form.

The comment is set in the font used for identifiers and constant symbols. However, if a "%" is within the comment, it is thrown away, and succeeding characters up until the next punctuation character (space tab , ; . ! ? " ') are set in the font used for substitution variables.

11.9 Standard Programming Techniques

A special technique which the language was designed to exploit involves the use of *continuations.* A continuation is a piece of code which is normally to be executed, if another piece of code (in a macro body) executes "successfully". In case of failure, however, the continuation is to be ignored, and some alternative action taken.

Suppose, for example, that we want to access a register operand. Normally we want to get back the contents of the register. If there is an error, for example trying to fetch a double-word beginning at register 31, then we want to abort the operation entirely. Now if we merely wrote:

... *let* op = *Access-Register-Operand* (*Num,Prec*) then <more>

then there is no simple way in the macro Access_Register_Operand to abort the operation <more> in case of an accessing error. The solution is to make the piece of code <more> explicitly available to the macro, so that it can decide whether or not it should be executed:

... *Access_Register_Operand* (*Num,Prec*) → op next <more>

We then write the definition as follows:

```
define Access-Register-Operand (Num,Prec) → Result next Continuation =
        if (Prec = D) ∧ (Num = 31)
            then Alignment-Error
    else let Result = case Prec of
                        Q: R[Num]<0:9>;
                        H: R[Num]<0:17>;
                        S: R[Num];
                        D: ⊂R[Num] || R[Num + 1]⊃;
                      end
        then Continuation
fi;
```

Now there are several interesting things to note here. One is that if the macro decides that the precision is "D" and the register number is 31, then the continuation is never executed at all, but rather the macro Alignment-Error (which presumably involves the code for taking an error trap). In any case, whatever it was that was going to be done when the register operand had been accessed is completely aborted. Another thing is that the text "Continuation" is *substituted* wholesale into the body of the macro definition. This means that the identifier matched to the substitution variable "Result" will be locally bound in the $let statement, and then will be visible to the code text in '*Continuation". Thus substitution variables do *not* behave like Aigol call-by-name parameters. Writing

Access-Register-Operand (op1.MODE,S) → value next n ← value + 1

is exactly like writing

```
if (S = D) A (op1.MODE = 31)
   then Alignment-Error
   else let value = case S of
                    Q: R[op1.MODE]<0:9>;
                       H: R[op1.MODE]<0:17>;
                       S: R[op1.MODE];
                       D: cR[op1.MODE]||R[op1.MODE + 1]⊃;
                end
        then n ← value + 1
   fi
```

whence it is clear that the binding of "value" is available to the continuation "n ← value + 1".

Another thing continuations are good for is "returning more than one value". Suppose we want to add two bit fields and a carry-in bit and get not only the sum but also carry-out and overflow bits. This is difficult to do using the functional notation "Add(s1,s2,cin)" without using obscure side effects. Using the notion of a continuation we write the definition:

```
define Add ( Addend,Augend,Cin ) → Sum, Cout, Overflow next Continuation ≡
          let x = Addend, y = Augend
     then let z = c0:x⊃ + c0:y⊃ + unsigned (Cin)
               then let Sum = low (width (x), z),
                    Cout = z<0>,
                           Overflow = (x<0> = y<0>) A (x<0> ≠ z<1>)
                      then Continuation;
```

Then if we write the call

```
Add (s1,s2,cin) → sum, cod, ov next <more>
```

this is exactly the same as writing

```
let x = s1, y = s2
then let z = c0||x⊃ + c0||y⊃ + unsigned (cin)
          then let sum = low (width (x), z),
                       cout = z<0>,
                       ov = (x<0> = y<0>) A (x<0> ≠ z<1>)
                  then <more>
```

Thus the identifiers sum, cout, and ov are all available to the continuation <more>. (So are the identifiers x, y, and z! The identifiers x and y are used in case the evaluation of Addend and Augend involve side effects. The identifier z is used to save time and to make the code more readable. However, because the Algol "copy rule" is purposely *not* used, in order to allow just such

"identifier conflicts" when desired, one must be careful not also to allow undesirable conflicts to occur. This requires care on the part of the programmer.)

Note that by convention the keyword **"$→"** is used to precede variables to be bound by the macro in a **$let** statement for the benefit of the continuation. This is meant to remind the reader of the assignment arrow **"←"**.

Also by convention, the keyword "next" or 'also" is used to precede to continuation to a macro. These keywords, when not appearing as part of a macro call, are used to denote language constructs that enforce or avoid ordering of execution. By convention these keywords are used in macro calls to indicate the same ordering or lack of ordering. Sometimes a macro may need to be called in one place using "next" and in another place using "also". This is the reason the **"vel"** construct is provided: one may write the macro prototype (for example):

(Overflow? (**$vel $also** #next) Continuation)	["LISP"]
Overflow? **also|** next *Continuation*	**["PASCAL"]**

The general rule (**in** the "PASCAL' syntax) is that if several "statements" appear separated by next, then they are executed in order, barring any errors; and if they are separated by also, then they may be permuted into any other order among themselves before being executed; but then if any statement is a macro call it may receive the remainder as a continuation. (This is only an intuitive, not a precise, description. In particular, it doesn't deal with the possibility of permuting the statements so that a macro call is last. The intended interpretation is that it receive a "null continuation". The interpreter for the language, running on a serial machine, in fact executes also in exactly the same way as it executes next. In this context the distinction is thus only a teleological one, a commentary on the code.)

12 Index