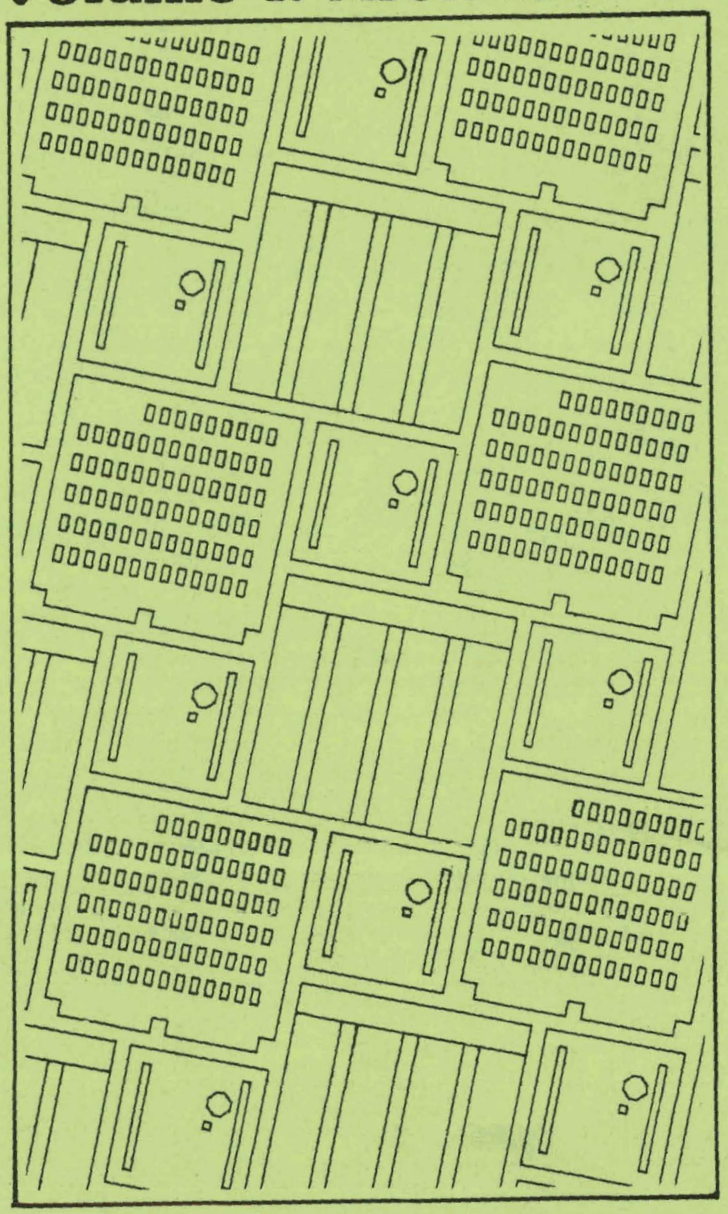
UCID-18619

Volume I: Architecture



1979 Annual Report
The S-1 Project

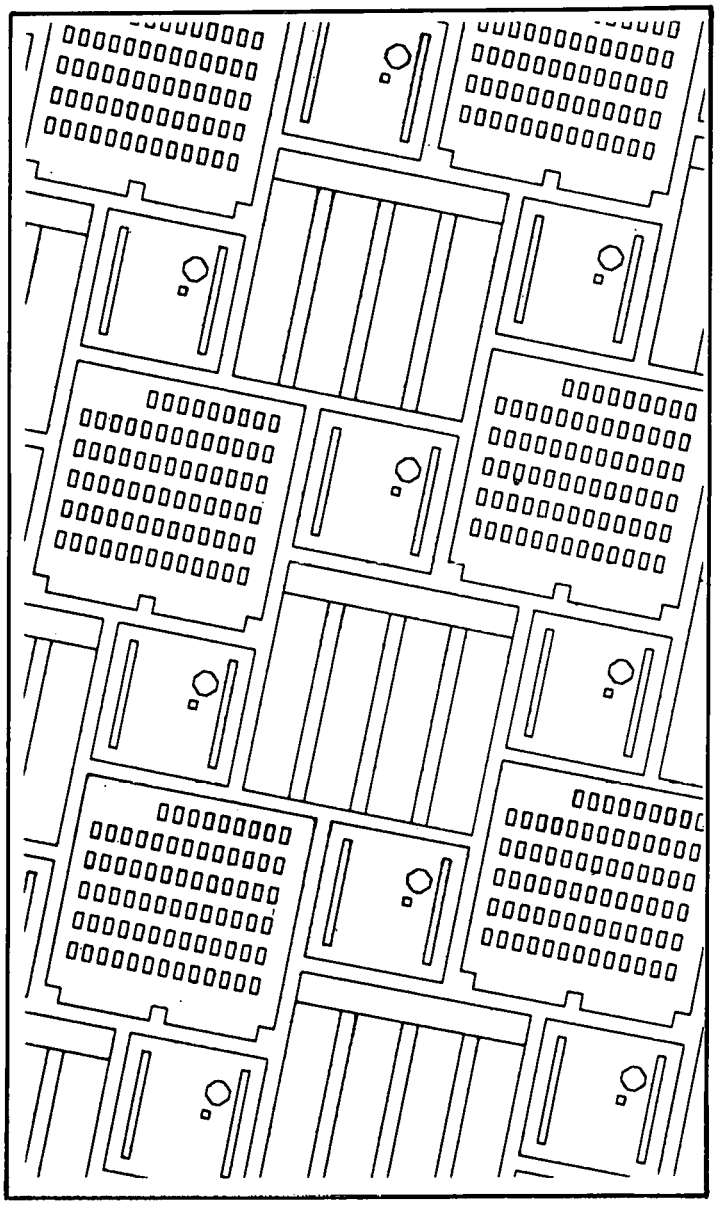
Prepared for
The Naval Systems Division, Office of Naval Research;
The Command and Control Division, Naval Electronics Systems Command; and
The Command, Control, Communication, and Intelligence Program Office,
Naval Material Command. Work in part performed under the auspices of
the U.S. Department of Energy under Contract No. W-7405-ENG-48.

LAWRENCE LIVERMORE LABORATORY

DISCLAIMER

DE82002155

Volume I: Architecture



1979 Annual Report
The S-1 Project

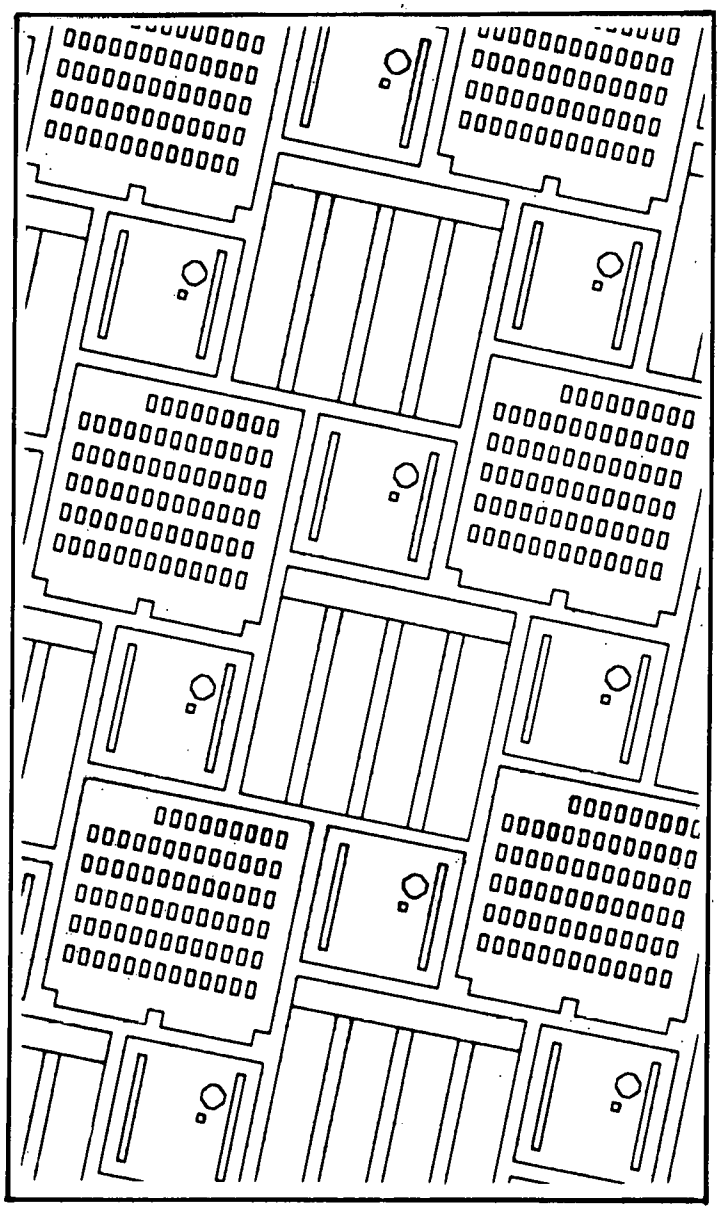
Prepared for
The Naval Systems Division, Office of Naval Research
The Command and Control Division,
Naval Electronics Systems Command
The Command, Control, Communication, and Intelligence
Program Office, Naval Material Command

LAWRENCE LIVERMORE LABORATORY

Thanks to Christine Ghinazzi, Lois Jones, L. R. Mendonca, Roland Portman, Joe Simpson, and Cindra Wheeler for help in various cutting, pasting, copying, proofing, and purchasing chores required to produce this book. Thanks to P. Michael Farmwald for the program which indexed the SMA-4 document.

CONTENTS

1



Executive Summary (EXS-79)

Lowell L. Wood

1 Executive Summary

The US Navy is one of the world's largest users of digital computing equipment having a procurement cost of at least $50,000, and is the single largest such computer customer in the Department of Defense. Its projected acquisition plan for embedded computer systems during the first half of the 80s contemplates the installation of over 10,000 such systems at an estimated cost of several billions of dollars. This expenditure, though large, is dwarfed by the 85 billion dollars which DoD is projected to spend during the next half-decade on computer software, the near-majority of which will be spent by the Navy; the life-cycle costs of the 700,000+ lines of software for a single large Navy weapons systems application (e.g., AEGIS) have been conservatively estimated at most of a billion dollars.

The S-1 Project is dedicated to realizing potentially large improvements in the efficiency with which such very large sums may be spent, so that greater military effectiveness may be secured earlier, and with smaller expenditures.

The fundamental objectives of the S-1 Project's work are first to enable the Navy to be able to quickly, reliably and inexpensively evaluate at any time what is available from the state-of-the-art in digital processing systems and what the relevance of such systems may be to Navy data processing applications: and second to provide reference prototype systems to support possible competitive procurement action leading to deployment of such systems.

The Project's efforts might seem to be addressed only to the hardware aspects of DoD's hundred billion dollar computing-related expenditures through end-FY85, and thus to be of relatively low leverage. However, many studies have documented the fact that use of efficient software generation practices, such as the exclusive use of high-level, structured programming languages, can result in software life-cycle cost savings of a factor of ten or more, relative to generating software in low-level, unstructured fashions, such as assembly- or microcoding-type languages. Indeed, extracting the maximum performance from its obsolescent computing plant has forced the Navy to employ the latter approaches in the large majority of its software generation activities. Computing hardware which supports the former type of more manpower- and cost-efficient software engineering practices with minimum penalties in utilization efficiency can therefore favorably impact the entire Navy computing cost structure. It is to the creation of such hardware, and the basic software necessary to support its cost-efficient utilization, that the S-1 Project is directly oriented.

During FY79, the Project's third full year of effort, its focus remained directed on the development of maximally cost-effective means for realizing digital data processing *systems* for use in Navy applications environments, means which can endure into the indefinite future. The general strategy which continues to be employed in pursuit of this capability is two-pronged:

- demonstration in fully operational prototypes of the maximally cost-effective hardware base of such digital processing systems, realized in a fashion which is manifestly repeatable as the underlying digital technology base continues to advance exponentially with time;

- timely creation of a suite of system software which is sufficient to completely support a

representative set of Navy digital computer-using applications employing this hardware base, and which is readily extensible to support the remainder.

Demonstration of the maximally cost-effective hardware base of such systems and establishing that the cost-effectiveness of this base may be maintained with the passage of time is at once the more challenging and the more novel of the Project's two major strategy components. Attainment of maximum cost-effectiveness of a digital processor at any given time clearly places an exponential premium on implementation in components which represent current technology, inasmuch as the underlying semiconductor technology has advanced exponentially in cost-effectiveness over the past two decades.

Only slightly less obvious are the stiff premiums placed on efficient use of the best current components to realize a complete but not excessively redundant or specialized set of data processing capabilities within a rapidly implemented processor system. Such a system must feature high levels of innovation in architectural conception, discipline in translation of the architecture into a detailed design, and alacrity in the entire process of moving from conception through initial implementation and evaluation to mass production of a proven prototype.

The type of digital data processing system on which the S-1 Project has focussed its attention is essentially unrepresented in commercial computer systems, as it is basically a hybrid of an advanced programmable signal processor and a general purpose, high-performance scientific (i.e., non-business-oriented) computer system. Moreover, S-1 systems emphasize reliability, maintainability and security features to extents almost unparalleled in commercial computer offerings. S-1 systems may therefore be expected to find many military applications, particularly where uniquely great challenges are posed to the real-time reactivity, data throughput/performance, reliability and secure natures of digital data processing systems.

The primary means chosen by the S-1 Project to be able to design and implement its processor systems in the most current technology has been the creation of the S-1 Structured Computer-Aided Logic Design (SCALD) System, a powerful aid for the computer system architect/designer which substitutes for essentially all the junior engineer/draftsman-level effort ordinarily required to support such work. Use of the SCALD System leaves the system architect free to specify the design on a relatively abstract, highly conceptual level, and requires only a specification of the technologies in which the design is to be implemented and general directions as to how its major sub-systems are to be located in three-dimensional space when the design is realized in a system package.

The SCALD System supported the design and hardware implementation of the first S-1 processor (the Mark I) in less than a year's time, through a process requiring only two person-years of total effort. FY79 has seen its major extension to support the design of the much more powerful S-1 Mark IIA processor, an endeavor which has been completed in its major aspects during this past year with less than three person-years of design effort, and is expected to culminate in the initial operation of a pair of fully functional Mark IIA systems in mid-FY80, after the expenditure of another two person-years of design and implementation endeavor. In contrast, the median time hitherto required in industry to design and implement a state-of-the-art digital processor has been four to six years, involving the expenditure of 100-300 person-years of effort.

One year typically elapses between the announcement of a new digital processor-related technology (e.g., a new integrated circuit memory element) and its availability in quantities which can reasonably support prototyping; another year passes before quantities adequate for mass production become available at reasonable prices. It therefore seems clear that both of the first two generations of the S-1 SCALD System adequately satisfy the timely design-and-implementation criterion noted above: one year to design and construct a state-of-the-art computer system, followed by a year for evaluation prior to commitment to large scale production, tracks quite well the composite learning curve of the underlying technologies. This time schedule was followed for the S-1 Mark I system development, and has been tracked through most of the development of the substantially more challenging Mark IIA system, as well. It therefore seems likely to be generally applicable to all computing system developments which employ the SCALD System. This constitutes early attainment of one of the Project's basic goals: providing the Navy and all of its potential computer system suppliers in the industrial sector with the means for realizing computers on much shorter and smaller time, effort and cost scales than hitherto attainable.

The first generation SCALD System has been described in two papers presented at the 15th Annual Design Automation Conference, and in last year's Annual Report. It has been extensively presented during FY79 to American industry, as well as to academic and Governmental audiences, including presentations made at two day-long Project Open House sessions held at LLL, primarily for American industry. One of these was devoted exclusively to SCALD for benefit of the then-emerging VHSIC community, by Navy direction.

SCALD I has been transported to a large number of organizations in these communities which are interested in using it to support large digital design efforts. The second generation of SCALD (SCALD II) is documented at high level in this Report, and will also be presented at the 17th Annual Design Automation Conference. As a substantial generalization of SCALD I, it is expected to be received even more enthusiastically than SCALD I has been. It will be distributed with extensive supporting documentation as soon as its correct end-to-end functioning is verified by its successful use in creating the Project's Mark IIA systems.

The high level of architectural innovation required to realize a new type of digital computing system--one which stresses extremely powerful real-time signal processing capabilities well-integrated into a powerful general-purpose processor--has been insured by the usual practice of considering all those features which have been found useful in previous digital systems of both types, supplemented by the rather unique means of formally soliciting detailed comments and suggestions from a relatively large group of academic, industrial and Government computer scientists. This process determined the basic architecture of the Project's Mark I processor, as reported in the end-FY77 Report.

The existence and operational status of the Mark I processor stimulated a great deal more, and more detailed, commentary on the S-1 architecture from the various segments of the American computer science community during FY78 and FY79, commentary made with the knowledge that criticisms meeting with widespread peer approval would be reflected in the uniquely plastic S-1 architecture

literally overnight. As a result, the S-1 architecture has rapidly become one of the most extensively studied and criticized in existence, and is without doubt the most extensively revised in the history of computer technology; it has presumably benefitted greatly from this intensive and unusually broad-based peer review process.

A comprehensive effort has been made during FY79 to integrate all such inputs consistent with basic Project goals and Navy interests into the design of the Project's second-generation (Mark IIA) processor. The external peer review portion of this inter-generation enhancement process has been similar to that employed in developing the architecture of the Project's Mark I processor, and has been greatly facilitated by the completely non-proprietary, non-commercial nature of the Project.

The usually very demanding task of maintaining the integrity of a computer architecture from one generation to the next, work in which the Project has been substantially engaged during the past two years, has been very substantially simplified by the Project's taking the unprecedented step of carrying all the microcode (*firmware*) of its processors in writeable memory. A uniquely *plastic* processor also results from this basic architectural decision, which has been supported by major advances in pertinent semiconductor technology during the past few years.

Highly efficient, and thus maximally cost-effective, use of an S-1 processor's hardware then results for any reasonable microcode specification by the processor's user(s). In particular, S-1 processors may be readily re-configured to quite efficiently emulate other computer architectures (particularly those with word lengths of <= 36 bits), simply by replacing the largely microcode-expressed S-1 *native* architecture with a microcoded expression of the architecture of the *target machine*.

The first step in this process, the creation of a macrocode-based simulator, was completed during FY78 for the widely used (e.g., by the DARPA community) PDP-10 computer systems and for the central processing unit (CPU) of the Navy's top-of-the-line general purpose processor, the AN/UYK-7. These S-1 Mark I processor-based simulators have been used to flawlessly execute substantial low-level (e.g., numeric object-time) programs for each of these computer systems. Such extensive, production-type use of these simulators during this past year included support of the Project's design of the Mark IIA via routine execution of the Stanford University Drawing System (which is written in 30,000 lines of PDP-10 assembly language) which serves as the graphics editor of the SCALD System, and support of the creation of a true emulator of the AN/UYK-7 CPU architecture which, for example, executes a Navy tactical air warfare program significantly faster on the Mark I processor than does a real UYK-7.

Discipline in realization of a processor architecture in a detailed, implementation-directed design is facilitated by the nature of the SCALD System itself; SCALD System usage discourages and highlights designer-level architectural modifications, while supporting rapid design completion by a small team of architect-designers who are able to communicate closely throughout the design period. SCALD System extensions realized during this past year and tested in supporting the detailed design of the Mark IIA processor system have further enhanced these aspects of realizing an architecture in hardware.

The architecture of the S-1 family of processors and the rationale leading to it are described in

detail in the S-1 Architecture section of this Report. The S-1 architecture is at once

- *powerful*, as is indicated by the 15 MIPS instruction-issuing rate of its second generation expression, the Mark IIA processor, its essentially 3-address instruction format and its advanced arithmetic/logic unit, all of which support its ability to execute a typical mix of Navy technical applications codes written in high level language comparably rapidly as any general-purpose processor in existence, and far more rapidly than any present Navy computer system;

- *highly plastic*, due to its completely writeable and very capacious microcode storage areas and its sophisticated instruction-decode and operand-fetching unit, so that it can efficiently emulate a wide variety of other processors at the hardware level, thus affording a very high level of protection to the Navy's investment in architecture-specific software written for other computers;

- *readily extensible* in stand-alone capabilities via addition of specialized arithmetic modules to its basic instruction-executing unit, and in system capabilities via interconnection to a number of other such processors to constitute uniquely powerful and reliable multi-processing systems;

- *time-wise durable*, by virtue of both its uniquely large address space, which permits immediate exploitation of exponentially decreasing memory costs and associated Navy applications demands for ever larger working memory space, and by its very broad hardware capabilities and extremely readily extended instruction set, which facilitates adaptation to changing Navy applications requirements;

- *cost-effective*, in that it makes very efficient use of its major hardware endowments, employs the most modern LSI components effectively, and adapts readily and in a software-invisible fashion to further semiconductor technology base advances (e.g. VLSI MOS memory elements, LSI ECL and CMOS logic modules).

Continued evaluation of the Project's Mark I processor, primarily for reliability, maintainability and performance in exceptional circumstances, was a significant hardware-related activity during FY79. This work included the completion of microcoding of the processor to express the full instruction set specifying the S-1 architecture, the examination of the functionality of the Mark I processor to ascertain that the hardware-microcode combination properly expressed the system architecture under all circumstances (including extremely rare *exception* cases and combinations thereof, of which a sophisticated *pipeline* implemention such as that of the Mark I has many), and the determination of the performance of the Mark I processor on various types of applications programs.

In order to carry out portions of this evaluation process, it was necessary to replace the 256 K word main memory system of the Mark I processor, which was implemented in 8 K bit MOS RAMs and had been procured in FY77 from a commercial custom memory systems source, with a Project-designed and -implemented memory system of 512 K words capacity, implemented in 16 K

bit RAMs. This was accomplished expeditiously during FY78. Continuing evaluation work and SCALD II exercise requirements during FY79 impelled the further expansion of the Mark I memory implementation to 2048 K words, with a Project-designed and -implemented memory unit which also served to prototype the memory module for Mark IIA uni- and multi-processor systems. This extension was accomplished at a hardware and implementation manpower cost of 5 K$/megabyte, and involved no alteration of the Mark I uniprocessor. Thus, the S-1 Mark I system now contains more computing power and more memory capacity than does the entire AEGIS computing plant.

Extension of the high cost-effectiveness of single S-1 processors to Navy applications requiring more computing capability than is available from single processors is accomplished by interconnecting a number of such processors into an S-1 multiprocessor system. Such a system at once provides a very high bandwidth, completely flexible means by which all member processors may exchange data (via a relatively very inexpensive full Crossbar Switch which uniformly interconnects all processors with all memory banks), as well as a variety of means by which processors may very rapidly signal to each other. In addition to carrying the extraordinary unit cost-effectiveness of single S-1 processors into almost arbitrarily demanding Navy applications regimes (which can exploit concurrent processing capabilities), the multiplicity of processors in such an interconnection also allows the creation in system software of extremely graceful system failure modalities: unexpected loss of *any* relatively small number of processors or memory banks need not degrade system performance at all, if appropriate software constructs are employed to automatically substitute reserve processors and memory units for ones which fail.

A general description of the S-1 multiprocessor architecture appears within this report, accompanied by a discussion of the strengths and weaknesses of such an architecture, relative to alternative approaches to meeting the digital processing requirements of the most demanding Navy applications.

The heart of the multiprocessor system, the Crossbar Switch, has been designed to be readily partitionable into a number of smaller, electrically isolated crossbar switches, and includes a diagnostics/maintenance processor which supports such software-controlled re-partitioning to isolate faulted processors or memory banks. Use of ECL-10K MSI circuits in implementation permits this switch to sustain 8 billion bit/second data transfer rates between 16 processors and 16 memory banks, while having an implementation cost somewhat less than that of a single processor. Interestingly enough, only about 20% of this cost (or less than 1% of the cost of a 16 processor, 16 memory unit multiprocessor system) has a growth rate which is quadratic in the processor or memory unit population size; the other 80% has a growth rate which is linear in this population size. A 128 processor, 128 memory bank S-1 multiprocessor system implemented in current technology would thus require a Crossbar Switch costing less than 10% of the cost of the total system. Since the advance of semiconductor electronics into the VLSI regime will necessarily produce components that reduce the size and cost of the Crossbar Switch before it produces components that reduce the size and cost of processor and memory, the fractional cost of the crossbar switch portion of S-1 multiprocessor systems, already quite small at present, may be expected to decline especially rapidly.

Software for the S-1 prototype processor family must be available in an essentially complete, reliable, documented, cost-effective and timely fashion to enable high *hardware* cost-effectiveness to be

translated into comparably high *system* cost-effectiveness, and providing for such is the other major component of the Project's strategy.

The S-1 Project, after surveying alternatives, elected to commence meeting these requirements by exploiting recent developments in software technology (e.g., highly transportable compilers and operating systems) in a selected academic computer science environment, which offered relatively inexpensive and highly talented (i.e., highly cost-effective) software design and implementation capabilities. A FY77 sub-contract effort at Stanford University's Computer Science Department supporting the S-1 Project produced and documented a PASCAL compiler, a FORTRAN compiler design, a P-Code translator, a symbolic assembler and a simulator for the first S-1 prototype processor; a companion loader was generated by the Project staff in FY77. The productivity of the Stanford effort was determined to be about an order of magnitude higher than industry programmer productivity standards, due both to its higher average talent level and the comprehensive use of recent software technology.

This software development task has been greatly facilitated by several features in the processor's architecture which permit high-level-language programs to make unusually efficient use of the processor's hardware resources. For instance, the Stanford-produced Pascal compiler was determined during this past year to produce code for a wide range of scientific-type problems which required an average of only 34% greater execution time than did optimally hand-coded programs for the same set of problems; the peephole and global optimizers whose development commenced in FY79 are projected to bring this high-level language average efficiency penalty to less than 10%. Even the initial one-third efficiency penalty is unusually small for use of a high-level language on a high-performance computer system.

The previously commenced software development work at Stanford has been continued during FY79. These efforts included detailed definition and initial development of the extended Pascal to be used in future SCALD development, Pascal*; the completion of the development of the common intermediate stack-oriented language, U-Code; the completion of the scientific function run-time library implementation; major progress in the development of the common global optimizer for the Pascal and FORTRAN compilers, and the completion of an enhancement program for the FORTRAN compiler. These Stanford software projects were complemented by LLL-centered software efforts which completed a Pascal-based general-purpose microcode assembler, extended the single-user, batch-type operating system being used on the S-1 Mark I processor, and made notable progress in the transport of the UNIX operating system to the S-1 processor family (the OS-1 effort) and in the detailed definition and design of the full-capability operating system (OS-2, or Amber).

Essentially all of this software will be transported without modification to subsequent generations of S-1 processors, whose architectures will be upward-compatible with previous generations (in order to minimize software development expenses associated with Navy evaluation of S-1 systems, and to serve as a significant scale demonstration of such long-term architecture upgrading capability).

The foregoing summarizes fulfillment of the S-1 Project's FY79 Work Statement referenced in

ONR Order N00014-79-F-0021, as re-negotiated with cognizant ONR and NAVELEX officials during this past year to accommodate to changing Navy programmatic and budgetary guidance.

During FY80, the S-1 Project has proposed to:

- implement a pair of prototypes of the second-generation S-1 processor (the Mark IIA), which will incorporate the advances made in semiconductor technologies since the Mark I processor was implemented, and which will include a very high performance arithmetic module to enhance real-time signal processing performance levels to well beyond that of any other general-purpose processor in existence;

- complete the low-level design and the implementation of a 16 processor, 16 memory unit, high performance Crossbar Switch on which a full-sized S-1 multiprocessor system may be centered;

- implement a multiprocessor system with an aggregate processing capability at least an order-of-magnitude greater than the most powerful single digital processing system in existence, centered on the Crossbar Switch and containing 16 processors and 16 memory units, thereby demonstrating an ability to extend processing capability and greatly augment system reliability at constant, high cost-effectiveness for all Navy applications allowing concurrent processing;

- implement two uniprocessor systems for installation and on-site evaluation at two locations to be designated by the Navy, and to provide reasonably comprehensive, LLL-based systems support for such evaluation activities;

- pursue software development (both at Stanford University, via continuation sub-contract arrangements, and within the Project at LLL) through the development of a multi-tasking operating system for an S-1 multiprocessor system, the design of a full-functionality operating system for an S-1 multiprocessing system, completions of a microcode-augmented emulator capability for the UYK-7 computer system, system integration, check-out and documentation of S-1 LISP, completion of the Pascal* development, the enhancement of the SCALD System to support design and implementation of the Project's third-generation processor, the Mark III.

- support initial Navy evaluation of the S-1 architecture and the suitability of the Mark IIA uni- and multi-processor systems for various Navy applications, by making them available for both local and remote (via ARPANET) Navy applications studies.

The material in this Report is divided by topic area into three volumes for easier handling. The remainder of this first volume is devoted to a detailed characterization of the S-1 architecture, highlighted with examples. Two articles constitute this Report's second volume, and describe major features of the Project's FY79 work: one reviews the basic features of the design of the Mark IIA uniprocessor system, and the other represents a highly user-oriented, comprehensive description of

the Project's second-generation SCALD system, which is intended to have widespread utility in US computer creating activity following its validation in creating the first Mark IIA systems. Volume III contains a discussion of the Project's major FY79 software developments, and is supplemented by a microfiche-based listing of all of the major software modules developed by the Project during the FY79 period. A summary-by-title of both the articles-in-text and the microfiched software immediately follows this Executive Summary.

2 Titles of the Articles of this Report

Volume I: Architecture

1. Executive Summary.

2. S-1 Multiprocessor Architecture. An overview of a multiprocessor system composed of multiple S-1 Uniprocessors sharing memory through a crossbar switch.

3. Investigation of the Partitioning of Algorithms Across an MIMD Computing System. Research on adapting existing algorithms to take advantage of the additional computing power available in a multiprocessing system.

4. S-1 Uniprocessor Architecture. The native mode instruction set for the S-1 Uniprocessor, and the syntax for an assembler which processes that instruction set.

5. UYK-7 Emulation. A novel technique which emulates the existing UYK-7 architecture by appropriately substituting a sequence of S-1 Native Mode instructions, rather than microcode, for each UYK-7 instruction.

Volume II: Hardware

1. Highlights of the Design of the Mark IIA Uniprocessor. Annotated drawings describing the high level aspects of the Mark IIA uniprocessor hardware, prepared as input to the SCALD II computer-aided logic design system.

2. SCALD II User's Manual. A document describing the SCALD II system from a user's viewpoint.

Volume III: Software

1. Overview of the Amber Operating System. The Amber Base System serves as the foundation for a family of problem systems capable of fully exploiting the power of both S-1 Uniprocessors and Multiprocessors.

2. Overview of Interim Operating Systems. Descriptions of an interim operating system for the Mark I Uniprocessor and of the effort to transport UNIX for use with the Mark IIA Uniprocessor.

3. User's Guide to S-1 Pascal and Fortran. How to use languages and utilities available on the interim Mark I system.

4. Pascal and Pascal* Compiler Systems; Pascal*: A Pascal Based Systems

Programming Language. An overview of the family of language translators which share a common intermediate language called U-Code; and a description of an extension to Pascal for SCALD development support.

5. PASMAC: A Macro Processor for Pascal. A description of the Mark I Pascal macro facility.

6. UFORT: A Fortran to U-Code Translator. A description of the FORTRAN facility of the Mark I system.

7. S-1 U-Code: A Universal P-Code. The definition of the U-Code intermediate language.

8. S-1 Code Generator and Optimizer. Documentation of a code generator and optimizer for the S-1 family of language translators.

9. UASMINT: A U-Code Assembler and Interpreter. An interpreter which executes U-Code, allowing the testing of a language translator independently of the code generator.

10. Portable Runtimes for a Portable U-Code System. Runtime support routines for the family of language translators.

2.1 Summary of Microfiche Accompanying this Report

The following files appear, in order, on the microfiche included with this Report. The three-character extension following the "." in each name indicates the source language:

FAI	FAIL (DECSystem-10 assembly language)
SAI	Stanford Artificial Intelligence Language (SAIL), a variant of ALGOL.
S1	S-1 Native Mode assembly language
PAS	Pascal

FASM2.FAI The source for a macro assembler which processes S-1 Native Mode assembly language. "S-1 Uniprocessor Architecture" in Volume I of this Report explains how to use this assembler.

FSIM2.FAI The source for a simulator for the S-1 Native Mode architecture. "User's Guide to S-1 Pascal and Fortran" in Volume III of this Report explains how to use this simulator.

RDOPS.FAI The source for a program which reads a file defining opcode mnemonics and produces a table which an assembler or simulator can use to map mnemonics onto opcode values.

CMDSCN.FAI The source for a program used by FASM2.FAI to parse a command line specifying input, output, and indirect files.

FLINK.S1 A linker which processes ".LDI" files and produces a ".RIM" file. The linker is automatically invoked by various command files described in "User's Guide to S-1 Pascal and Fortran" in Volume III of this Report.

PPIMPL.PAS A version of a 2D hydrodynamics and heat conduction program used at Lawrence Livermore National Laboratory, converted to Pascal and organized for parallel computation. "Investigation of the Partitioning of Algorithms Across an MIMD Computing System" in Volume I of this Report describes this program.

The following files relate to the U-Code language translators. The programs themselves are preliminary versions.

UFORT2.PAS Documented in "UFORT: A Fortran to U-Code Translator" in Volume III of this Report.

UPAS0.PAS A Pascal to U-code translator, whose use is described in "User's Guide to S-1 Pascal and Fortran", in Volume III of this Report.

UINT.PAS A U-code interpreter, documented in "UASMINT: A U-Code Assembler and

Interpreter" in Volume III of this Report.

SUPN05.PAS A U-code to S-1 code translator

PIO.PAS Pascal I/O runtimes, documented in "Portable Runtimes for a Portable U-Code System" in Volume III of this Report.

FIO.PAS Fortran I/O runtimes, documented in "UFORT: A Fortran to U-Code Translator" in Volume III of this Report.

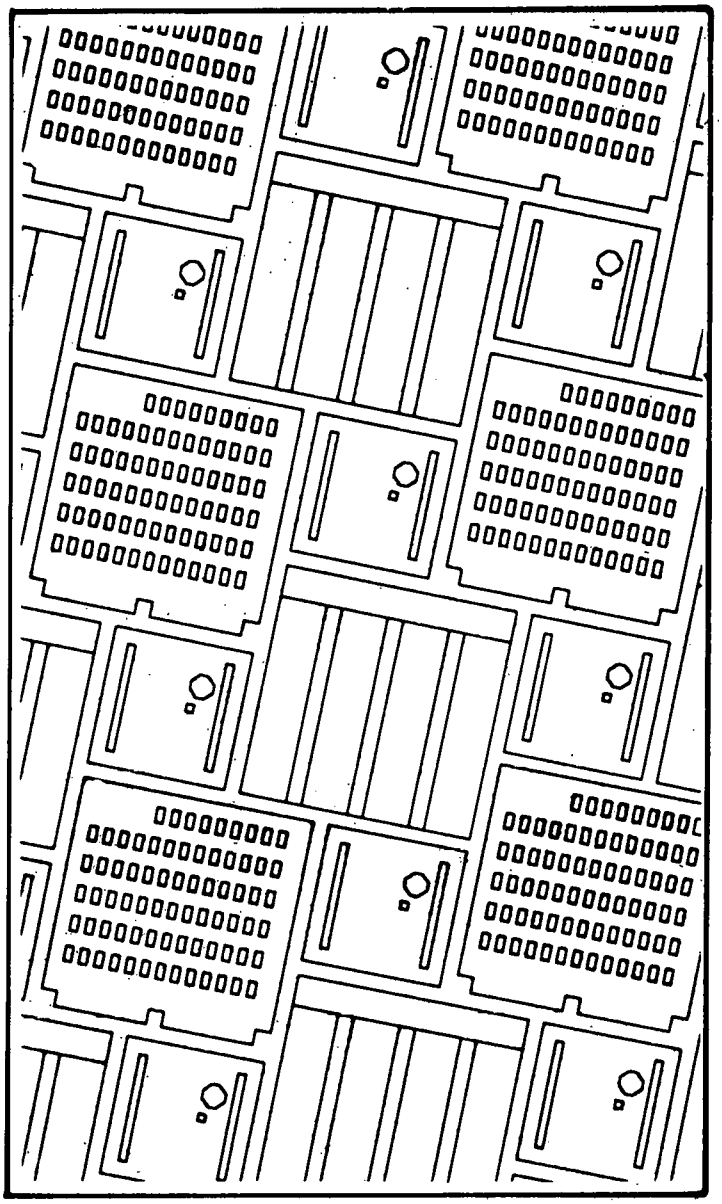
SIO.S1 Primitive I/O runtimes, documented in "Portable Runtimes for a Portable U-Code System" in Volume III of this Report.

SNUM.S1 Primitive numerical runtimes

VERCH.PAS A version-changer for Pascal programs, described in "User's Guide to S-1 Pascal and Fortran" in Volume III of this Report, which provides a facility similar to the IF switches available in many assemblers.

PAS.SAI, PAS2.SAI
A Pseudo-monitor for the Pascal system at SAIL, described in "User's Guide to S-1 Pascal and Fortran" in Volume III of this Report.

2



S-1 Multiprocessor Architecture (MULT-2)

L. Curtis Widdoes, Jr.

TABLE OF CONTENTS

1 S-1 Multiprocessors

One of the S-1 Project's main thrusts is the development of a multiprocessor which computes at an unprecedented aggregate rate on a wide variety of problems. The S-1 Multiprocessor will be implemented first with second-generation S-1 Uniprocessors (S-1 Mark IIAs). For a large class of important numerical problems, including signal processing, it will achieve a computation rate roughly an order of magnitude greater than that of the Cray-1 computer. The Cray-1 in turn offers performance two to four times greater than that of the CDC 7600, and outperforms all other existing computers in general numerical computation work.

A multiprocessor is a network of computers which *concurrently* execute a number of independent instruction streams on separate data streams (i.e., a multiple-instruction-multiple-data machine, according to [Flynn 1972]) and which *closely share main memory*. A typical S-1 Multiprocessor consists of sixteen independent, identical S-1 Uniprocessors and provides a computation rate for many technical problems more than an order of magnitude greater than the rate of a single S-1 Uniprocessor, which alone processes typical scientific mixes of logical and numerical operations at approximately the same speed as a Cray-1.

Along each of several dimensions, a multiprocessor design offers significant advantages over a uniprocessor design providing an equivalent computation rate. The advantages result from the modularity inherent in a multiprocessor architecture, and can be categorized as advantages of *reliability*, *economy*, and *size*.

The advantage of *reliability* has been validated by commercial systems such as the Tandem Nonstop (see [Datapro 1979]) and the BBN Pluribus (see [Ornstein 1975]), which provide ultra-reliable operation in handling banking transactions and ARPA Network traffic, respectively. In a well-designed multiprocessor system, failure of a single module (for example, a component uniprocessor, a bus, a crossbar switch, or a memory bank) does not entail failure of the entire system. Indeed, the S-1 Multiprocessor Operating System (Amber) is designed to detect such module failures and effect graceful replacement in function from the available complement of reserve modules of the multiprocessor system.

Of primary importance among the advantages of *economy* are the economies during machine construction due to replication of a single module type. This economy during the construction phase is extremely important with respect to current and projected semiconductor technologies, since the unit replication cost of VLSI chips varies nearly inversely with the replication factor, except for a small additive base cost.

A *second economy* of scale relates to the cost of the design work; the design cost per processing element is reduced asymptotically to zero as the processing element is replicated. Actually, any real multiprocessor must include some design costs per processing element which grow as the number of processing elements is increased (for example, the cost of designing the interconnection network), but these costs can be made negligible, and in fact are negligible in the case of the S-1 Multiprocessor.

A *third important economy* is the potentially reduced time lag between the freezing of the system design and the delivery of the first operational system. By replicating a relatively simple processing element many times and using a regular interconnection network, this lag can be made very small; it is virtually independent of the processing power of the total system. As a result, the semiconductor technology used in a properly designed multiprocessor can be essentially state-of-the-art, whereas the technology used in a more complex processing structure must be considerably more out of date. This time lag phenomenon will continue to seriously degrade the cost-

effectiveness of delivered complex systems as long as advancing semiconductor technology continues to provide exponentially more cost-effective components, but may be greatly reduced in advanced multiprocessors.

One additional economy is the economy which results from the freedom of the multiprocessor designer to choose the most cost-effective uniprocessor element structure, independent of the processing rate of the element. Cost-effectiveness of uniprocessor structures is not constant over all levels of processing power. Because the design of a digital processing system must be aimed not only toward maximum cost-effectiveness, but also toward some minimum processing power, designers of high-performance uniprocessor systems have not been able to utilize structures with possibly higher cost-effectiveness but lower processing power. On the other hand, the designer of a multiprocessor may be able to achieve a total cost-effectiveness which is nearly the same as the cost-effectiveness of the component uniprocessor and, since that uniprocessor need not be constrained to have a large minimum processing power, to achieve substantially higher cost-effectiveness of the resulting system.

Independent of these economic advantages is the advantage of *size*. Regardless whether it is economical to build arbitrarily powerful uniprocessors, at some point it becomes physically impossible (with state-of-the-art technology) to build these machines; multiprocessors, however, have a higher processing-rate ceiling. This advantage of multiprocessor structures is important because maximal computing rates will be necessary for certain applications into the foreseeable future; numerical weather prediction with its real-time constraints is an obvious example.

Figure 1-1 shows the logical structure of a typical S-1 Multiprocessor. This S-1 Multiprocessor includes sixteen independent S-1 Mark IIA Uniprocessors, of which two are shown. The internal logical structure of the S-1 Mark IIA is indicated at a very high level. All sixteen uniprocessors are connected to main memory through the S-1 Crossbar Switch; one possible access pattern is shown with dots. Sixteen memory banks are shown, each of which can contain up to 2^{30} (one billion) bytes of semiconductor memory. Input and output are done through peripheral processors (for example, LSI-11s); as many as eight can be attached to each S-1 Mark IIA Uniprocessor. The Synchronization Box is based on a shared bus connected to each member uniprocessor providing for specialized medium-bandwidth communication associated with the synchronization of tasks performed by individual uniprocessors. Each module in the S-1 Multiprocessor is connected to a diagnostics-and-maintenance processor (an LSI-11), which allows convenient remote display-oriented maintenance and control of the multiprocessor.

All sixteen identical S-1 Uniprocessors can execute independent instruction streams on independent data streams. Thus, all sixteen uniprocessors can cooperate in the solution of a single large problem. The high-bandwidth, low-latency inter-processor communications provided by the Crossbar Switch facilitate problem partitioning with little efficiency loss, but the sixteen uniprocessors also have the capability to process completely independent tasks, for example, each S-1 Uniprocessor might service different users. Memory requests from the member uniprocessors are serviced by sixteen memory banks with an aggregate maximum capacity of 2^{34} (sixteen billion) nine-bit bytes. Connectivity between uniprocessors and memory banks affords the maximum generality; any processor can *uniformly* access all of main memory through the S-1 Crossbar Switch. The programmer thus sees a huge, uniform address space, as each memory request from each uniprocessor is decoded by hardware in the Crossbar Switch and sent to the appropriate memory bank.

The Crossbar Switch processes requests from member uniprocessors to perform read or write access to specific (essentially randomly indexed) memory banks. In the first multiprocessor implementation, the Switch allows only one request for a given memory bank to be honored at

any instant (hence, at most sixteen transactions can be ongoing simultaneously, and as many as sixteen only if no two uniprocessor requests are for access to the same memory bank). Conflicting requests are queued *fairly*, that is, in a queue which guarantees service to each requesting processor once before service is given to any requesting processor twice.

The Crossbar Switch has a maximum peak bandwidth of over 10 billion bits per second when all of its sixteen channels are transferring data simultaneously. Although the growth rate of such a square crossbar is asymptotically $O(N^2)$, where N is the number of processors or memories, the S-1 Crossbar costs somewhat less than a single S-1 Uniprocessor. Less than 25% of the Switch, or 0.8% of total system cost (arbitrarily assuming that half of the total system cost is invested in the memory), exhibits $O(N^2)$ growth rate; the remainder exhibits O(N) growth rate. Hence, it is economically quite feasible to implement crossbar switches for uniprocessor and memory populations much greater than sixteen; the generality of full interconnectivity between processors and memory may be obtained at very low (although asymptotically $O(N^2)$) cost.

The S-1 Multiprocessor design allows component uniprocessors and memory banks to be physically distributed over distances which are limited only by average bandwidth requirements (which obviously degrade linearly with increasing length). Because of the relatively large latency introduced in main memory transactions due to the lengths of the cables, because of the Switch transaction time, and because of the access time of relatively slow but highly cost-effective memory chips, each member uniprocessor contains private cache memories. These caches automatically (that is, without guidance from the programmer) retain recently referenced data and instructions within a relatively small amount of ultra-high-performance memory, in the expectation that those data will be referenced again in the near future. Whenever a reference to such a retained datum or instruction is made, the information is immediately delivered directly from the cache, thus eliminating the latency required for a main-memory transaction. Although a similar efficiency can be realized if main memory contains a special high-speed area (such as the SCM of the CDC 7600), such a design places on every programmer the burden of managing a variety of memory systems in order to maximize efficiency of program execution.

The presence of caches in a multiprocessor necessarily introduces problems of *cache coherence* (see [Censier 1978]); without a guarantee of cache coherence, programming of certain problems in a cache-based multiprocessor would be inconceivably difficult. A system of caches is coherent if and only if a read done by any processor P of a memory location M (which may be cached by other processors) always delivers the value written to M *most recently*. *Most recently* in this context has a special meaning in terms of a partial ordering on reads and writes of memory throughout the multiprocessor (see [Lamport 1978]), but for an intuitive understanding of the problem it is sufficient to think of recentness in terms of absolute time. In these terms, whenever a write is done by one processor P to a memory location M, completion of the write must guarantee that all subsequent reads of location M *by any processor* will deliver the new contents of M, until another write to M is completed.

The caches of the member uniprocessors of S-1 Multiprocessors are *private* in the sense that there are no special communication paths connecting the caches of one uniprocessor with the caches of any other uniprocessor; the cache coherence problem is therefore especially challenging. To solve it, the S-1 Multiprocessor includes a design closely related to one independently proposed in [Censier 1978]: a small tag is associated with each line (a set of sixteen words) in physical memory. This tag identifies the (unique) member uniprocessor (if any) which has been granted permission to retain (that is, *own*) the line with *write access*, and identifies all processors which own the line with *read access*. The memory controller allows multiple processors to own a line with read access, but responds with a special error flag when a request is received to grant read or write access for any line which is already owned with write access, or when a request is received to

grant write access for any line which is already owned with read access. Any uniprocessor receiving such an access-denial response is responsible for requesting (through a simple interrupt mechanism) that other uniprocessors flush the contested line from their private caches. This procedure maintains cache coherence dynamically, hence extremely efficiently, without requiring any effort by the programmer.

To support low-latency, semaphore-type communication between member uniprocessors, a Synchronization Box attaches to one of the eight I/O ports of each uniprocessor. The Synchronization Box is centered on a shared bus; one major function of this unit is to transmit interrupts and small data packets from one uniprocessor to any subset of other uniprocessors in order to coordinate processing streams.

For reliability, all single-bit errors which occur in memory transactions are automatically corrected, and all double-bit errors are detected, regardless whether the errors occur in the Switch or in the memory system. For single-point failure immunity, the S-1 Multiprocessor allows for the permanent connection of multiple Crossbar Switches which are electronically selectable; operation of the S-1 Multiprocessor can thus continue in the event of a single Switch failure. Furthermore, the Crossbar Switch can be configured to keep a backup copy of every datum in memory, so that failure of any memory bank will not entail loss of crucial data. Each I/O processor may be connected to I/O Ports on at least two uniprocessors, so that failure of a single uniprocessor does not isolate any I/O device from the multiprocessor system. To enhance maintainability, each member uniprocessor, each Crossbar Switch, and each memory bank is connected to a diagnostic computer which can probe, report, and change the internal state of all modules which it monitors, in great detail and with precise timing.

In a typical S-1 Mark IIA Multiprocessor, sixteen Mark IIA Uniprocessors execute independent instruction streams and communicate with main memory through a high-bandwidth Crossbar Switch. Private caches implemented with extremely fast but quite expensive memory components within the member uniprocessors effectively hide the combined latency of the Switch and memory system, and hence allow the use of relatively slow but extremely cost-effective memory components to store virtually all of the the data and instructions to be processed.

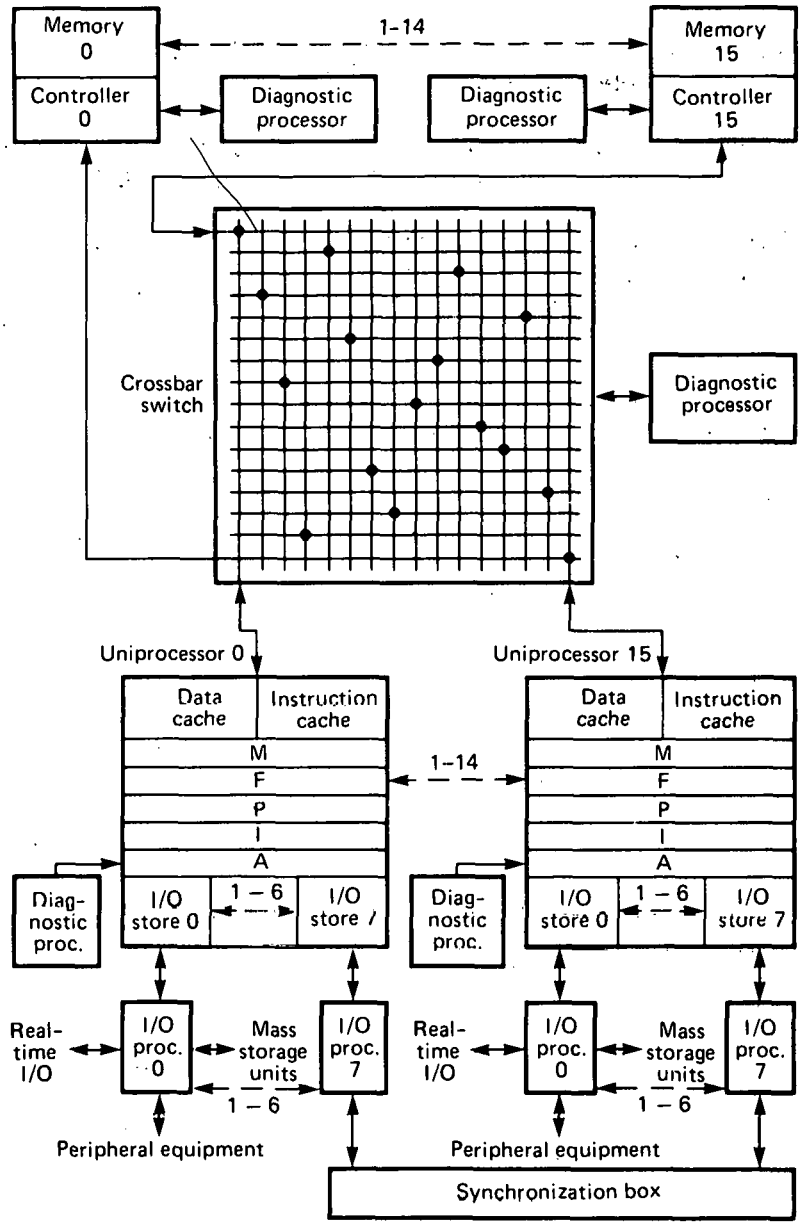


Figure 1-1
Logical Structure of the S-1 Mark IIA Multiprocessor

2 References

[Censier 1978] Censier, L. M., and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, December, 1978, Volume C-27, Number 12, pp. 1112–18.

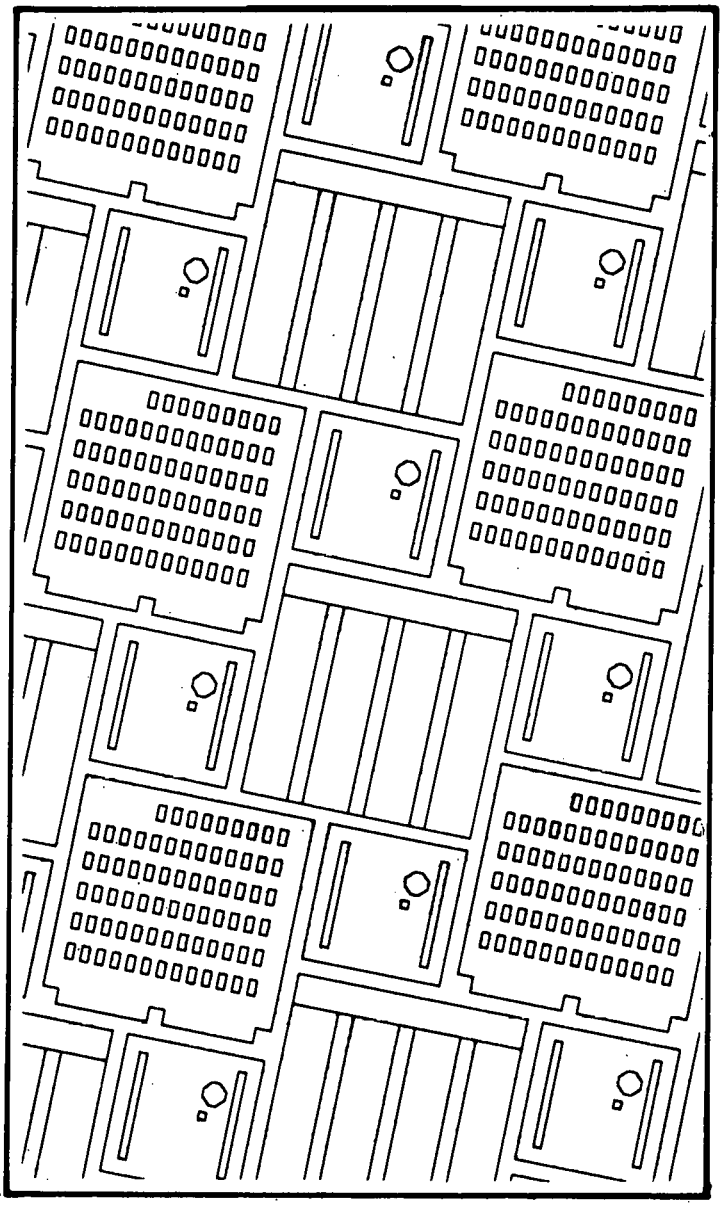
[Datapro 1979] Datapro Research Corporation, "Tandem Non-Stop Systems", in *Datapro Reports on Minicomputers*, Datapro Research Corporation, Delran, N.J., January, 1979.

[Flynn 1972] Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, September, 1972, Volume C-21, Number 9, pp. 948–60.

[Lamport 1978] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, July, 1978, Volume 21, Number 7, pp. 558–65.

[Ornstein 1975] Ornstein, S. M., et al., "Pluribus -- a Reliable Multiprocessor", *Proceedings AFIPS 1975 NCC*, 1975, Volume 44, pp. 551–59.

3



Investigation of the Partitioning of Algorithms Across an MIMD Computing System (IMAP-1)

Erik. J. Gilbert

1 Introduction

Multiprocessors (strictly speaking, Multiple-Instruction-Multiple-Data processor systems [4]) are potentially extremely attractive systems for realizing greatly enhanced computing capabilities. Potential benefits include significant improvements over both the Single-Instruction-Single-Data and Single-Instruction-Multiple-Data types of uniprocessor systems in the areas of availability, configurability, cost-effectiveness, and raw computing power. The primary concern of this paper is in the area of raw computing power enhancement available from a multiprocessor. Particular reference is made to a classic multiprocessor architecture being explored by the S-1 Project [8,9].

In order to best realize the computing power increase potentially available from a multiprocessor on a single application problem, it must be possible to express the algorithmic solution to the problem in some partitioned fashion in order to make effective use of several processors at once. The simplest and most obvious, but still useful, scheme for partitioning is to run several different, independent copies of the application algorithm on different sets of data which are of interest to the researcher; such an approach is predicated on the different data sets being totally independent of each other.

However, the more interesting case occurs when the algorithm is structured to take advantage of parallelism inherent in the problem when processing a single set of data. Such an approach admits of possibly very large gains in effective processing speed, and thus potentially allows many more cases of interest to be studied *in serial order* per unit of wallclock time; such an approach is required if subsequent data sets have features determined from computational study of previous ones. It is this particularly useful case to which the present investigation is addressed.

This report documents aspects of progress made to date in the continuing investigation of application partitioning across classic MIMD multiprocessors. The goal of this investigation is to demonstrate the practicality of the partitioned application mode of multiprocessor use for large classes of realistic problems, particularly in the context of a large-scale multiprocessor such as the S-1 project has designed and will be implementing. The investigation so far has included a broad spectrum of studies, ranging from general research on multiprocessing issues to specific experiments with algorithms for particular application problems.

This report covers several different topics, roughly following the chronological development of the investigation to date. After some definitions and further motivation for application partitioning, there is a brief discussion of generally applicable techniques for partitioning. Next is a historical perspective of the process of selecting a "representative" application for further detailed study. An overview is then given of the algorithm chosen for specific study, followed by a description of the methods used for partitioning that algorithm. After that appears a discussion of some simulation results, followed by some analytic results. Finally, there is a discussion of some of the detailed implications of this study in terms of synchronization and communication mechanisms found to be desirable for support of application algorithm partitioning. The report concludes with a discussion of directions which such investigations may profitably take in the future.

2 Definitions

The term "multiprocessor" will be used in this paper to refer to a generalization of the structure of the S-1 multiprocessor. A few important attributes of this generalization are listed here. It is assumed that there is a moderate number (say 2 to 200) of extremely fast single processors tightly coupled to a relatively large amount (at least 10 million words) of uniformly accessible global memory. Each processor may also have a moderate quantity of very high performance memory (e.g. cache) local to it, but it must also have high bandwidth (although not necessarily short latency) access to the global memory. Many of the ideas contained herein apply also to other multiprocessor structures (e.g. larger numbers of slower processors), but the S-1 structure has been the primary focus for optimization of the partitioning approach developed in this study.

"Problem partitioning" refers to the process of taking a particular application problem and constructing an algorithmic solution for it which can take advantage of the potential for parallel execution available in a multiprocessor. The primary motivation assumed for partitioning a problem is to substantially decrease the absolute wallclock time taken to run each instance of the application (as opposed to other motivations such as improved reliability and/or recoverability). For partitioning to be realistically useful in this way, the partitioned application must run substantially faster than a uniprocessor version, even when all possible overheads are taken into account, including operating system, multiprocess communication, and synchronization.

The "speedup" of a multiprocessor algorithm is the ratio of wallclock elapsed time for uniprocessor execution to wallclock elapsed time for multiprocessor execution. It is, of course, a function of the number of processors, and possibly other algorithm parameters. The speedup provides a measure of the success with which the problem has been partitioned, indicating greater success as the speedup approaches the number of processors. There are actually conditions in which the speedup can theoretically exceed the number of processors; these will be noted in more detail later.

3 Motivation

Depending on the exact nature of the application, the process of constructing an effectively partitioned solution can vary greatly in difficulty. As mentioned earlier, any uniprocessor code can be immediately run on a multiprocessor in the mode of multiple independent data files; but this is not a partitioned single application as defined here. This mode does serve to characterize a class of applications whose partitioning is trivial. Any application which consists of several already independent computations can be easily partitioned in this way. A simple example (in which each of the independent computations has the same structure) might be a Pascal compiler which has the ability to process multiple input procedures in a "separate compilation" mode.

There is another class of applications which is almost as easy to partition. It is all those which have a basic iterative "outer loop" with perhaps a summary data gathering step at the end of each iteration, but with several otherwise independent computation blocks occurring in each iteration. Examples of this structure of computation may be found in Monte Carlo approaches to simulation [5].

To approach the issue of difficulty of partitioning from another standpoint, it is reasonable to ask for what kinds of applications is a substantial amount of partitioning effort justified. In particular, if an application is hard to partition it could be argued that it is better to run it unpartitioned in timesharing mode along with other user problems in order to still gain the cost-effectiveness benefits of the multiprocessor. However, there are several interesting application areas in which any gains in absolute wallclock execution time are valuable. Classic examples include the weather prediction problem and many types of real-time processing, such as radar signal processing. Also, as the number of processors in the multiprocessor increases, the attractiveness of the partitioning approach increases for more and more problems.

4 Techniques for partitioning

As the number of designed or implemented multiprocessors increases, a few general techniques for problem partitioning are beginning to emerge [6]. Three such techniques which have been considered could be called "synchronous partitioning," "asynchronous partitioning," and "pipelining." From the descriptions below it should become apparent that these techniques are by no means mutually exclusive, and hence may be used in combination in a partitioned application.

The technique of synchronous partitioning is perhaps the most obvious and most widely applicable of the three. In this technique, either the data structure or the program (or both) is divided up into comparatively independent units, and multiple processes compute in parallel within these units. Occasionally, two or more processes must synchronize with each other in order to maintain data consistency or pass summarizing information among processes.

The technique of asynchronous partitioning [1] is less intuitive and can lead to debugging difficulties due to the lack of exact reproducibility of results, but offers advantages by avoiding the potentially large overheads of frequent process synchronization. This technique is best understood in the context of iterative numerical algorithms. For instance, consider an application containing a large two dimensional matrix of real numbers which are being updated by an iterative algorithm such that each new point value depends in some simple way on previous values of neighboring points. The points may be partitioned into groups among the available processors. If the correctness of the algorithm does not depend on the use of a precisely defined previous iteration value for neighboring points in the updating procedure, and if instead any reasonably recent value will suffice for convergence, then the processes may iterate without synchronization at each iteration. The termination test for convergence is most easily implemented if the error measure is defined so that it can be tested locally in each process, determining process convergence independent of other processes. Thus the only form of synchronization is implicit in the shared point values, which are continually updated in parallel. Note that a pure implementation of this technique has the characteristic that no process is ever in synchronization wait, and so all processes are always actively working towards the solution. However, it is possible for convergence to be slower than in a synchronized solution due to nonuniform use of previous values. The general ideas of this technique have been the subject of research for several years, often appearing under the name "chaotic relaxation" [2].

The pipelining technique is very similar to the pipelined approach in high-performance uniprocessor hardware implementation. In this technique, the computation is divided into several parts, called "stages," which have the characteristic that the output from one stage becomes the input to the next stage. So, once the computation is well under way, all of the stages can be computing in parallel with the data streaming into the first stage and the results streaming out of the last stage. An example of this approach might be the division of a compiler into scanner, parser, global optimization, and code generation stages.

One note about the interaction between implementations and multiprocessor efficiency and speedup deserves mention here. Some problem partitionings, especially those using pipelining, lead to an implementation which has a fixed maximum speedup, e.g. the number of pipeline stages.

Other partitionings which are parameterized by the number of processors (and possibly some measure of data size) have no obvious fixed maximum speedup, and thus (at least for large data sizes) can continue to benefit from additional processors. Thus, the implementor should be aware that, by requiring a fixed length pipeline or division into a fixed number of parallel processes, a limit on future flexibility for expansion is being imposed.

5 Selection of a sample application

Since the main goal of this investigation is to demonstrate the practicality of partitioned execution of real-world problems, the study includes considering several application areas and specific codes as possible candidates for partitioning. A number of possible codes were considered from many different application areas, but most of the emphasis to date has been concentrated on one particular code, named SIMPLE [3]. SIMPLE may be characterized as a large scale numerical physical simulation, using well known techniques for the widely important problem of solving partial differential equations on a reasonably large two dimensional mesh.

SIMPLE was chosen for several reasons: (1) it seems to be representative of techniques used in many physical modelling codes, in that it contains both explicit and implicit PDE solvers, it uses a two dimensional Lagrangian formulation, and it uses table lookup for the required equations of state of the fluids being modelled; (2) it is sufficiently simplified from a full-scale code to be quite manageable in size (as it consists of about 1800 lines of Fortran); (3) it has been studied by others in the academic sector as a candidate application for a number of novel processor architectures, such as data-flow machines.

Large scale numerical simulations such as these form one significant class of applications for which multiprocessor partitioning seems to be appropriate. Several other application areas have been suggested and studied by other researchers. One application considered because it is widely used but still fairly self-contained is sorting. Internal (main memory) sorting is fairly CPU intensive but still difficult to partition effectively, since obvious partitionings are often theoretically limited to less than linear speedup [7]. Another general area of application is heuristic search of large tree structures such as those found in artificial intelligence problems. One other application which has been studied in this light is set partitioning integer programming [7].

6 Overview of SIMPLE

The intent of the SIMPLE code is to give a simple, yet realistic, example of computational fluid dynamics and heat flow. It solves the differential equations of inviscid compressible shock hydrodynamics and simple heat conduction using a Lagrangian formulation. It works in two dimensions on a region with a regular boundary. It uses simple table lookup to represent the equations of state of an ideal gas.

The differential equations are reduced to difference equations. The equations for hydrodynamics and for heat conduction are solved in separate sections of the code employing different techniques. The hydrodynamics equations are solved explicitly, while the heat conduction equations are solved implicitly.

The basic data structure in SIMPLE is used in the representation of the mesh covering the problem domain. This consists of 13 two dimensional arrays of real numbers to store the physical quantities involved, plus a few additional arrays for working storage. There are also one dimensional arrays to store the tabular definition of the equation of state, and of course several scalars to store miscellaneous other quantities.

The outer loop structure (after the problem is set up) is a simple iteration as the time value is increased:

```
repeat
    hydrodynamics pass;
    heat conduction pass;
    compute new delta t;
    advance time by delta t;
until done
```

The hydrodynamics pass has the following structure:

```
for each mesh zone, calculate new pressure using EOS lookup;
for each boundary zone, calculate geometry;
for each boundary zone, set up boundary physics;
for each mesh point, calculate new velocities;
for each mesh point, calculate new coordinates;
for each mesh zone, calculate new density and change in specific volume;
for the boundary, sum up the work done on the boundary by hydrodynamics;
for each mesh zone, calculate artificial viscosity and Courant delta t limit;
for each mesh zone, calculate hydrodynamic work and update energy, using EOS;
for all zones, sum up the kinetic energy for the entire problem;
for each mesh zone, calculate new temperature via table lookup;
```

The heat conduction pass has the following structure:

```
for each mesh zone, calculate two material properties;
for the boundary, set the boundary properties to neighboring values;
for each mesh zone, calculate coupling constants in the K direction;
```

for each mesh zone, calculate coupling constants in the L direction;
for the boundary, set some appropriate initial values;
over the entire mesh, perform a forward and backward sweep in L (see text);
over the entire mesh, perform a forward and backward sweep in K (see text);
for each mesh zone, calculate new energy using EOS, and new delta t limit;
for the boundary, sum up the energy flow across boundaries;
for all zones, sum up a new internal energy for the entire problem;

Notice that, with one significant exception, all of the steps in both passes have a very similar structure. A typical step passes over the entire mesh (or maybe just the boundary) making local computations at each mesh zone or mesh point. These local computations typically involve updating one or more quantities at the given place in the mesh, after examining the previous value and perhaps the previous values of a few neighboring elements. Also, of course, computations involving only the boundary contribute much less to the CPU time used than computations over the whole mesh. Below in figure 1 is a pictorial representation of a typical SIMPLE mesh processing step, showing the obvious left to right and top to bottom ordering of mesh element computation. This will be compared in the next section with the multiprocessor partitioned ordering.

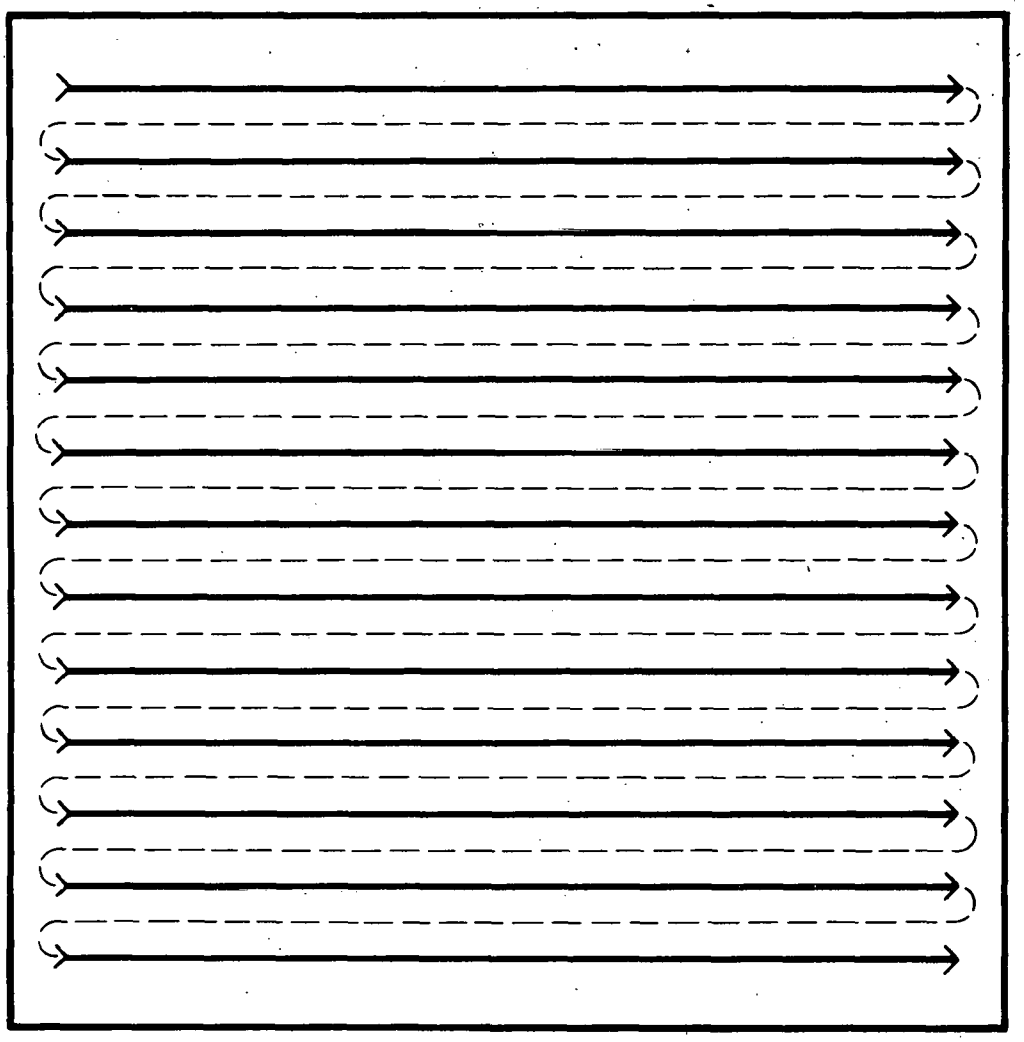


Figure 1: Typical mesh processing order

The one exception to this structure occurs in the steps in the heat conduction pass called "forward and backward sweeps." Superficially, even these steps may appear to have a similar structure. There is one important difference, arising from the implicit nature of the PDE solution technique used. In order to solve a tridiagonal linear system of equations, the sweeps evaluate a recurrence of the form X[I] := A[I]*X[I-1] + B[I] for increasing values of I. The key here is that each new X quantity depends on the new X quantity which was computed in the immediately preceding inner loop iteration. This dependence causes some difficulty in the partitioning of the sweeps, which will be discussed in the next section on partitioning of SIMPLE.

Another algorithmic structure which is used is the table lookup in the EOS and temperature calculations. In both cases this consists basically of locating between which pair of entries in an increasing table of values some physical quantity belongs numerically, and then using the

corresponding index into other tables to compute an interpolated function value. The lookup search is a straightforward sequential ordered table search. The only unusual part of the algorithm is that each table index is saved as a starting place for the next search, which reduces the search time assuming that successive uses of the function tend to pass arguments of similar magnitude.

7 Partitioning SIMPLE

Given the basic structure of SIMPLE as mostly performing localized operations fairly uniformly across a large data structure (the mesh), the most reasonable approach seems to be a data-directed synchronous partitioning. Specifically, each of several processes is assigned to operate on some subset of the mesh, computing independently of the other processes whenever possible. Occasional synchronization is required for keeping one mesh section from advancing too far beyond the others, for mesh-wide data summarizing operations, and in the sweep steps (as explained later).

An important factor to consider in partitioning a program which has a large shared data structure like SIMPLE's mesh is the presence of per-processor cache memory on S-1 multiprocessors. Due to the large difference in access time to a word in central shared memory and a word already in a processor's cache, it seems reasonable to select a programming style which has a high degree of per-processor data locality of reference. In a code like SIMPLE, where the computation within the large shared data structure is quite evenly distributed, an easy way to do this is to statically partition the data structure into fixed equal size pieces, with one piece per process. Each process is then responsible for updating its piece, and most of the references to that piece are made by that process, thus assuring locality. Notice that it is also being assumed that there is at least an approximate one-to-one mapping between processes and processors, and that processes do not migrate from one processor to another very often. Otherwise, the advantages of having all recently referenced data in cache would be lost. These assumptions are valid on the bare hardware of the S-1 multiprocessors, and must be supported by any operating system which is intended to maximally benefit from this type of operation.

For SIMPLE, the chosen static mesh partitioning is into "column groups." Each process is assigned a different fixed subrange of columns of all of the arrays representing the various physical quantities in the mesh. Of course, any process can still access any quantity at any point in the mesh since the entire mesh is in global shared memory. It is just assumed that most of the references within a column group will be by the assigned process, and hence that the column group data will reside largely in the corresponding processor's cache. Below in figure 2 the column grouping of the mesh is shown, along with the ordering within processes of a typical mesh computation, corresponding to the uniprocessor version in figure 1.

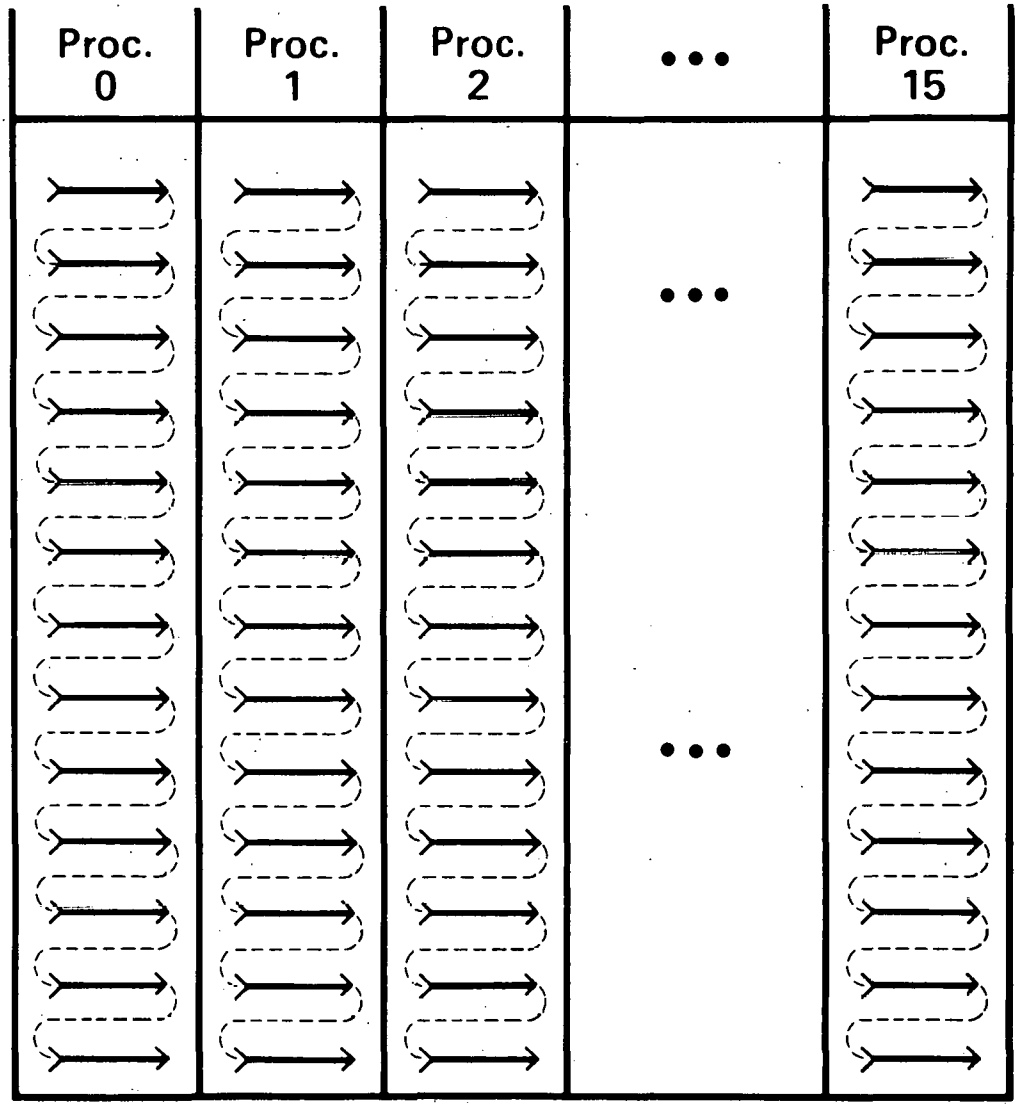


Figure 2: Partitioned mesh processing order (independent mesh computations)

The presence of the caches has another interesting performance implication, on the theoretical speedup achievable for a program like SIMPLE. For some reasonable mesh sizes, it is quite possible that all of the mesh data will not fit in a single processor's cache, but that it will all fit in all of the caches combined. In this case, the uniprocessor execution of the program could continually cause cache misses and corresponding lengthy delays while cache lines are transferred to and from main memory. However, the multiprocessor version, with properly partitioned references to the data, would be able to retain the entire mesh distributed in all of the caches, thus causing cache missing to be insignificant. In this way, if the efficiency of processor utilization is high enough, the speedup over the uniprocessor version could actually exceed the number of processors executing the program!

Here are some details of the process of partitioning a mesh-processing part of SIMPLE other

than the difficult sweep steps. One simplified typical code fragment might appear as follows (where K is the row index and L is the column index):

```
for L := LMN to LMX do
   for K := KMN to KMX do
      begin
      A[K,L] := (X[K,L]+Y[K,L]) * (Z[K,L-1]-Z[K-1,L]);
      B[K,L] := B[K,L] + Z[K,L]*Y[K-1,L+1];
      end;
for L := LMN to LMX do
   for K := KMN to KMX do
      begin
      P[K,L] := P[K,L] + A[K-1,L]*A[K,L];
      Q[K,L] := Q[K,L] + B[K,L-1]*B[K,L];
      end;
```

Notice that the computation at each mesh point is in terms of other quantities at the same mesh point or at a neighboring mesh point, thus maintaining the desirable locality mentioned above. The only references outside of local column groups occur when L is in the first or last column of a group and an off-column reference like Z[K,L-1] or Y[K-1,L+1] is made. Also notice that within each loop pair the computations at each mesh point are completely independent of each other, and so they may be performed in parallel with no interprocess synchronization needed. However, the second loop pair is dependent on the results of the first loop pair, so synchronization is needed to insure that the second loop pair is not executed by some process before the A and B values needed have been stored by perhaps a different process. An easy way to insure this is to insert a "synchall" synchronization call between the loop pairs. This call forces each process to wait at that point of execution until all processes have arrived there, and then they are all allowed to proceed. Since each process is performing essentially the same amount of work on its column group as any other process, all processes may be expected to complete the first loop pair at about the same time and not cause very much overhead wait time at the synchall point.

So, the partitioned version of the code fragment might appear as follows (where PR is the index of the process executing the code, and LMN and LMX have been expanded into arrays specifying the column boundaries of the column groups):

```
for L := LMN[PR] to LMX[PR] do
   for K := KMN to KMX do
      begin
      A[K,L] := (X[K,L]+Y[K,L]) * (Z[K,L-1]-Z[K-1,L]);
      B[K,L] := B[K,L] + Z[K,L]*Y[K-1,L+1];
      end;
SYNCHALL;
for L := LMN[PR] to LMX[PR] do
   for K := KMN to KMX do
      begin
```

```
P[K,L] := P[K,L] + A[K-1,L]*A[K,L];
Q[K,L] := Q[K,L] + B[K,L-1]*B[K,L];
end;
```

Another code fragment worth considering is one which includes a summary data gathering of some sort, such as the result of a summation or a maximum over some function of the mesh points. Such a computation requires a complete pass over the mesh with a single scalar output, rather than updated mesh values. A typical step of this sort in SIMPLE might appear as follows:

```
TOTAL := 0.0;
for L := LMN to LMX do
   for K := KMN to KMX do
      begin
      TOTAL := TOTAL + A[K,L]*X[K,L];
      end;
```

The obvious approach to partitioning this code fragment is to let each process compute a total for its column group, and then to have one process compute a grand total at the end. If the number of processes is sufficiently large, the simple grand total computation should perhaps be replaced by a multiprocess version which could compute pairwise subtotals, eventually reducing the number of totals to one grand total. So, a partitioned version of this code fragment could appear as follows (where MAXPROC is the number of processes):

```
PTOTAL[PR] := 0.0;
for L := LMN[PR] to LMX[PR] do
   for K := KMN to KMX do
      begin
      PTOTAL[PR] := PTOTAL[PR] + A[K,L]*X[K,L];
      end;
SYNCHALL;
if PR = 1 then   (* processor 1 computes the grand total *)
   begin
   TOTAL := 0.0;
   for P := 1 to MAXPROC do
      TOTAL := TOTAL + PTOTAL[P];
   end;
```

One more code segment which should be discussed is the table lookup in the EOS and temperature calculations. As mentioned previously, these code segments are essentially straightforward sequential ordered table searches, which can be executed independently by several processes in parallel with no synchronization since they are computing function values from read-only data. The only exception to this is the mechanism for retaining the search index from the previous search for use as a starting point the next time. The obvious way of partitioning this mechanism is to retain the previous search index on a per process basis, so that processes executing in unrelated portions of the mesh do not try to use each other's previous search indices.

Finally, some consideration must be given to the somewhat more difficult problem of partitioning the forward and backward sweeps in the heat conduction pass. It was noted previously that the difficulty arises from the recurrence inherent in the loops, in which each inner loop iteration is dependent on results computed in the previous iteration. Even this structure would not be difficult to partition if such iterations only traveled up and down columns, and hence were evaluating each recurrence only within a single process. Unfortunately, recurrence iterations are performed both up and down columns and across rows. So, some of the recurrences must be evaluated across process boundaries, requiring some form of synchronization at very frequent intervals (once per process boundary crossed, i.e. several times per row of the mesh in a single sweep). All previously discussed partitionings of SIMPLE required only about one synchronization per computation over the entire mesh.

A partitioned forward mesh sweep recurrence is diagrammed below in figure 3. In the figure, the mesh rows have been grouped into blocks of three rows each; row blocking is not used in the code below, but it will be discussed later in the section on analytic study. The vertically circled column group boundaries show points at which synchronization must occur. The diagonally circled column group portions represent a single time snapshot of how much computation can proceed in parallel, due to the skew enforced by the left to right recurrence. As time proceeds, more and more processes become actively executing in parallel. The average degree of parallelism depends on the "angle of attack" of the diagonal part, which is determined by the amount of row blocking, the mesh dimensions, the number of processors, and the synchronization overhead. These quantities will be studied in detail later, in the analytic section.

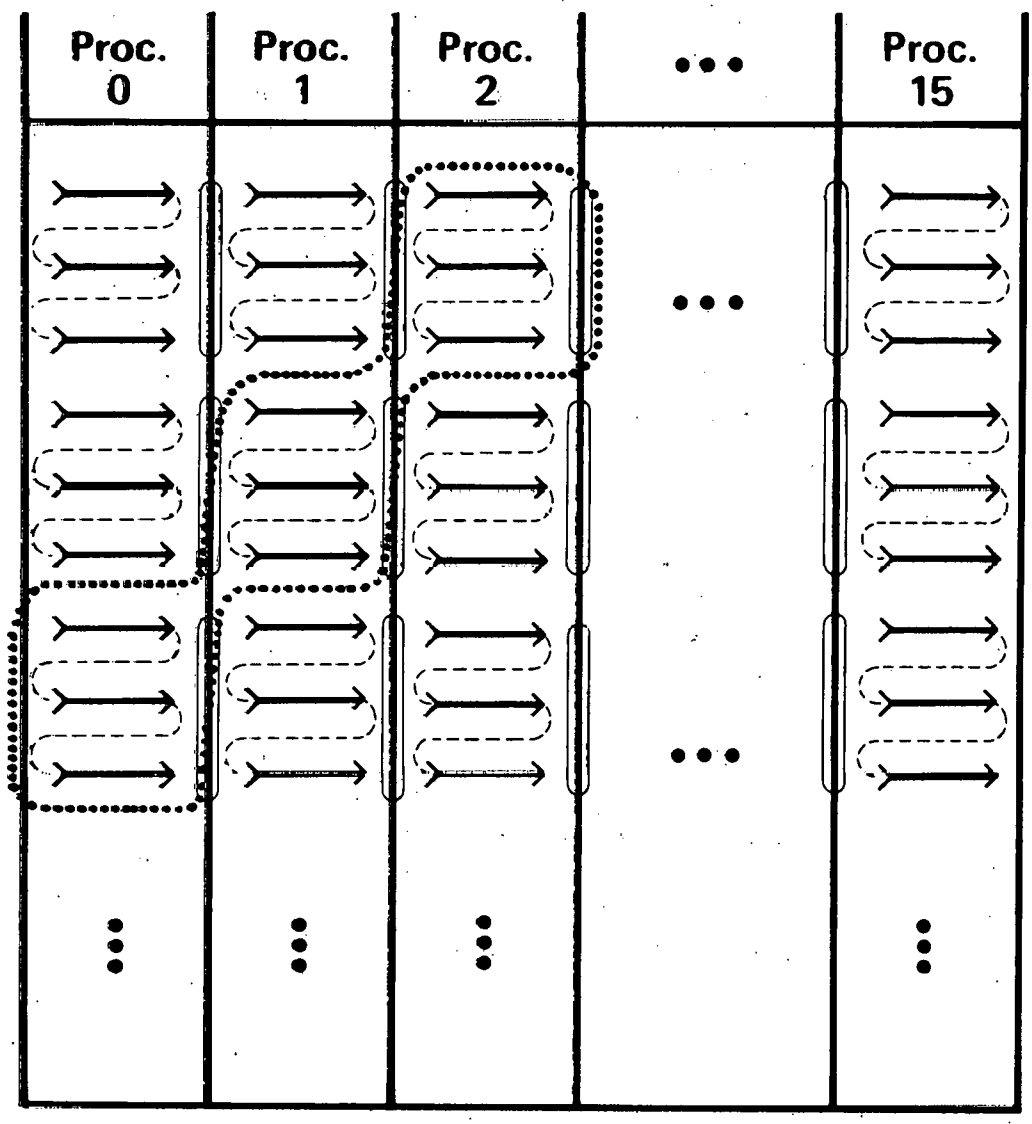


Figure 3: Partitioned forward sweep processing (with row blocking)

For reference, a slightly simplified version of the unpartitioned troublesome sweep code fragment appears below:

```
for K := KMN to KMX do
   begin
   for L := LMN to LMX do
      begin
      A[K,L] := Q[K,L] / A[K,L-1];
      B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
      end;
   for L := LMX downto LMN do  (* note stepping by -1 *)
```

```
        begin
        T[K,L] := A[K,L]*T[K,L+1]+B[K,L];
        end;
    end (*for K*);
```

For partitioning this code fragment, there must be a somewhat more detailed synchronization mechanism than the synchall call used previously. Let AWAIT(n) and SIGNAL(n) correspond roughly to Dijkstra-style semaphore operations P(SEM[n]) and V(SEM[n]). So, AWAIT(n) will be used to await a signal on channel n, and SIGNAL(n) sends a signal on channel n. Notice that the signal channels contain counters, so more than one signal may be outstanding on a channel. In this example, AWAIT(n) will be used to wait for a signal from process n that it is finished with the next row's worth of column group. Given these definitions, the partitioning discussed above might be expressed in this code fragment as follows:

```
for K := KMN to KMX do
    begin
    if PR > 1 then AWAIT(PR-1);
    for L := LMN[PR] to LMX[PR] do
        begin
        A[K,L] := Q[K,L] / A[K,L-1];
        B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
        end;
    if PR < MAXPROC then SIGNAL(PR);
    end (*for K*);
for K := KMN to KMX do
    begin
    if PR < MAXPROC then AWAIT(PR+1);
    for L := LMX[PR] downto LMN[PR] do  (* note stepping by -1 *)
        begin
        T[K,L] := A[K,L]*T[K,L+1]+B[K,L];
        end;
    if PR > 1 then SIGNAL(PR);
    end (*for K*);
```

There exists an alternative to the above frequently synchronizing structure for partitioning the sweeps. It would be possible to transpose the mesh quantities needed, perform the sweeps in the "easy" direction (up and down columns), and then transpose back. This unwieldy sounding approach could actually be quite feasible in practice when compared to the high overhead method outlined above, if the problem of efficiently transposing a matrix on the multiprocessor can be solved. At the moment this problem appears to be quite complicated, since it must attempt to keep all of the processors busy at the same time as avoiding delays from simultaneous access to any single central memory unit. Some further analysis of how much time the high overhead method spends in waiting will be presented below in the section on analytic speedup computation. Also, a new hardware-supported mechanism will be proposed in the section on synchronization and communication which should eliminate most of the overhead associated with loops like this one, thus

obviating the need for such a transpose mechanism.

The above examples tend to blur the distinction between variables which are shared by all processes and variables which are private to each process. In any actual implementation, of course, this distinction must be explicitly specified by the user to the system software. For SIMPLE, shared variables include the mesh quantities, the EOS lookup tables, and miscellaneous globally known scalars. Private variables include loop indices and temporaries.

8 Multiprocessor SIMPLE simulation

As part of this study, a modest simulation of the SIMPLE code running on a multiprocessor has been implemented. One of the major goals of this simulation was just to force the process of considering the entire code line by line, to be sure there were no major conceptual problems in partitioning it for a multiprocessor. Another goal was to study in general the effectiveness of the previously discussed approaches to partitioning, with particular emphasis on the viability of a static mesh partitioning. The simulation is accurate in the sense that it still actually solves exactly the same problem as that solved by the uniprocessor code, but it is incomplete in its consideration of the complexities of the multiprocessor environment. The entire source code for the multiprocessor SIMPLE simulator is included in the microfiche appendix to this report, in the file named "PPIMPL.PAS".

The basic approach of the simulation was to begin with the code of SIMPLE (translated into Pascal from Fortran), and to start by considering how to partition each stage for multiprocessor execution. However, each code segment which was intended to run independently in different processes is actually enclosed in a loop which executes the code segment successively for each process, varying the process number over all possible values. Variables which were private to each process (and had a useful lifetime long enough to justify keeping the values across major processing steps) were changed into arrays indexed by the process number.

To this structure were added timing, synchronization, and statistics gathering functions. The main timing function is assignment of CPU time spent in mesh computation to the simulated process which is spending that time. This is done by surrounding each code segment with calls to start and stop charging of CPU time to a specified process. The only synchronization function simulated at present is the synchall function described earlier. It is simulated by a procedure call to update timing statistics at each synchall point. The most interesting statistic is of course the speedup achieved. It is computed by assuming that wallclock time advances at the same rate as the maximum CPU time used by any process at each stage. Again, this assumes essentially that each process has its own dedicated processor. Other statistics gathered include per-process CPU usages, which may be examined to determine how successfully the workload is being balanced among the processes.

The results of sample runs of the simulation were quite encouraging. The per-process CPU usage was very well balanced, indicating that the static mesh partitioning appears to be a reasonable choice. The speedup reported for a small mesh on a 16 processor system varies between 9.7 and 14.5, depending on how it is chosen to account for CPU time which was spent but not attributed by the simulation to any particular process. Both the accuracy of the simulation and the speedup value are expected to increase as the size of the mesh increases.

There are a number of ways in which the simulation to date is incomplete, and so future improvements could increase the accuracy of the simulation. One minor improvement would be to accurately model the subtotal accumulation part of each summary data gathering step; at present these parts are assumed to be negligible and are not included. Also lacking is a detailed study of exactly which synchall points are absolutely necessary; at the moment they are scattered liberally

throughout the code wherever there is any possibility that global resynchronization might be needed. The influence of the caches was included in some analytical study (discussed later), but the simulation assumes uniform access to all of shared memory. The critical points in SIMPLE where cache misses will happen due to column group boundary crossing have been isolated but not yet included in the simulation. Probably the most important omission in the present simulation is accurate accounting for the complicated interactions in the heat conduction forward and backward sweeps. At present the simulation assumes that a no cost mesh transpose is done; this is obviously unrealistic.

9 Analytic speedup computation

The simulation studies of SIMPLE to date have ignored the implications on memory access times imposed by the per-processor caches of S-1 multiprocessors. The presence of the caches is quite important to consider due to the possibility of a more than tenfold increase in access time for a word not in cache over the access time for a word in the proper cache. In particular, accessing a word in cache takes only about 50 nanoseconds, whereas accessing a word from the cache of another processor will probably take about 300 nanoseconds (averaged assuming all words of a cache line will be accessed, corresponding to a cache line access time of 4 to 5 microseconds).

To augment the simulation results, some analytic study has been done of potential speedup of portions of the SIMPLE code, allowing for the presence of the caches. The portions chosen for analytic study are the sweep steps in the heat conduction pass and a time-consuming nested loop representative of the hydrodynamics pass. The sweep analysis is simplified by only considering the overhead implied by cache misses and cache line transfers, and not considering any overhead associated with process synchronization. The next section of this report proposes a mechanism which can reduce both types of overhead.

The sweep analysis will be presented for the forward sweep only. The forward sweep part of the slightly simplified code fragment which appeared earlier is repeated below for reference:

```
for K := KMN to KMX do
   begin
   for L := LMN to LMX do
      begin
      A[K,L] := Q[K,L] / A[K,L-1];
      B[K,L] := (Q[K,L-1]*B[K,L-1]) / A[K,L-1];
      end;
   end (*for K*);
```

It is assumed that the two dimensional arrays are stored by columns, i.e. that element A[1,1] is followed in memory by element A[2,1]. Thus each S-1 16 word cache line contains 16 elements of a column of an array. Since cache transfers happen in units of 16 word lines rather than single words, it is reasonable to assume that the overhead would be less if each process computes the above recurrence on a block of rows within its column group before letting the next process start on those rows, rather than synchronizing on each single row. This blocked approach allows more than one word to be used from each cache line each time it is transferred across from one processor to another at a column group boundary. For simplicity, the unit of time used here will be the length of time it takes one processor to execute a single loop iteration with no cache misses.

Define the following parameters:

B = blocksize = number of rows in a block
W = time to compute the recurrence over one block of one column group
P = number of processors

R = number of rows (KMX-KMN+1)
C = number of columns (LMX-LMN+1)
TSP = elapsed time for entire forward sweep on single processor
TMP = elapsed time for entire forward sweep on multiprocessor

The speedup for this section of code is then defined by:

$$\text{Speedup} = \frac{\mathbf{TSP}}{\mathbf{TMP}}$$

By the definition of the unit of time,

$$TSP = R \cdot C$$

Similarly, notice that since a block is B rows high and C/P columns wide, W would be equal to B•C/P in the absence of cache misses.

To formulate the value of TMP, the exact sequence of the multiprocessing sweep execution must be observed. Each of the P processors computes the recurrence at each element in all R rows in its assigned column group of C/P columns. In other words, each processor computes over R/B blocks, taking time W•R/B for the whole computation. If all the processors could execute for the whole sweep fully in parallel, W•R/B would also be the elapsed time of the entire computation. However, no processor can begin its computation until the previous processor has finished computing on its first block. So, processor P must wait for P-1 block computations until it can start on its first block. From then on all processors can run in parallel, assuming that each block computation takes the same amount of time. Thus, accounting for the delayed startup of processor P, the total elapsed time is:

$$TMP = W \cdot (P-1 + \frac{R}{B})$$

In formulating W, processor to processor cache line moves must be accounted for, in addition to the basic iteration compute time. The basic iteration time (of the real code in SIMPLE) is estimated at about one microsecond, and a cache line move takes 4-5 microseconds, so it seems a reasonable estimate that a cache line move takes about the same time as 4 basic iterations. Assuming that a previous step computed values for the array Q, causing its data to reside in the caches of assigned column group processors, the read-only use of Q[K,L-1] in each iteration causes each processor to participate in two cache line moves (one from the previous processor and one to the next processor) every 16 rows. So, the contribution to each block computation of accessing Q[K,L-1] is twice B/16 times the cache line move time, i.e. 2•B/16•4.

Each iteration also uses the values of A[K,L-1] and B[K,L-1], but *not* in a read-only fashion, i.e. each value used was written on a recent earlier iteration. So, the cache lines containing these values at column group borders must be moved between processors (twice) for every block which is processed, not just every 16 rows. In other words, when processor p finishes computation on a block, the cache line containing the last column of that block must be moved to processor p+1, and if the

next block to be computed by processor p also contains any part of that cache line it must be moved back to processor p. So, where $\lceil x \rceil$ ("ceiling of x") is the smallest integer greater than or equal to x, the contribution to each block computation of accessing both A[K,L-1] and B[K,L-1] is twice $2 \cdot \lceil B/16 \rceil$ times the cache line move time, i.e. $2 \cdot 2 \cdot \lceil B/16 \rceil \cdot 4$.

Therefore, the final formula derived for W is:

$$W = \frac{C \cdot B}{P} + 2 \cdot \frac{B}{16} \cdot 4 + 2 \cdot 2 \cdot \lceil \frac{B}{16} \rceil \cdot 4$$
$$= \frac{C \cdot B}{P} + \frac{B}{2} + 16 \cdot \lceil \frac{B}{16} \rceil$$

From all of the above, the speedup can be expressed:

$$\text{Speedup} = \frac{R \cdot C}{(\frac{C \cdot B}{P} + \frac{B}{2} + 16 \cdot \lceil \frac{B}{16} \rceil) \cdot (P\text{-}1 + \frac{R}{B})}$$
$$= \frac{P}{(1 + \frac{P}{2 \cdot C} + \frac{16}{B} \cdot \lceil \frac{B}{16} \rceil \cdot \frac{P}{C}) \cdot (1 + \frac{B \cdot (P-1)}{R})}$$

Notice that this formula has the expected quality that as the number of rows and columns in the mesh approaches infinity, the speedup approaches the number of processors.

For determining some numeric values of the speedup formula, some interesting parameter values can be substituted. Specifically, by letting P=16, choosing sample values for R and C, and then maximizing over B, the following speedups are obtained:

R	C	speedup max	occurs at B =
128	128	7.0	4
128	1024	11.4	2
1024	1024	14.1	4

Now, a time-consuming nested loop representative of the hydrodynamics pass will be analyzed. The loop chosen performs the function listed earlier in the SIMPLE overview as "for each mesh point, calculate new velocities." This loop forms the majority of a subroutine which uses 39% of the CPU time used in the hydrodynamics pass, and 26% of the total CPU time in SIMPLE. It is also in the class of easily partitioned loops in SIMPLE, since it requires no potentially costly synchronization calls within the loop body. Thus, the major factor which might limit speedup for this section is the overhead of cache misses due to shared array access. For reference, the exact text of the loop in question appears below:

```
for L := LMN to LMX do
   for K := KMN to KMX do
      begin
      AU := (P[K,L]+Q[K,L]) * (Z[K,L-1]-Z[K-1,L]) +
            (P[K+1,L]+Q[K+1,L])*(Z[K+1,L]-Z[K,L-1]) +
```

```
      (P[K,L+1]+Q[K,L+1])*(Z[K-1,L]-Z[K,L+1]) +
      (P[K+1,L+1]+Q[K+1,L+1])*(Z[K,L+1]-Z[K+1,L]);
AW := (P[K,L]+Q[K,L]) * (R[K,L-1]-R[K-1,L]) +
      (P[K+1,L]+Q[K+1,L]) * (R[K+1,L]-R[K,L-1]) +
      (P[K,L+1]+Q[K,L+1]) * (R[K-1,L]-R[K,L+1]) +
      (P[K+1,L+1]+Q[K+1,L+1]) * (R[K,L+1]-R[K+1,L]);
AUW := RHO[K,L]*AJ[K,L]+RHO[K+1,L]*AJ[K+1,L]
       +RHO[K,L+1]*AJ[K,L+1]+RHO[K+1,L+1]*AJ[K+1,L+1];
AUW := 2.0/AUW;
AU  := -AU*AUW;
AW  := AW*AUW;
U[K,L] := U[K,L]+DTN*AU;
V[K,L] := V[K,L]+DTN*AW;
if ABS(U[K,L]) <= VCUT then U[K,L] := 0.0;
if ABS(V[K,L]) <= VCUT then V[K,L] := 0.0;
end (*for L,K*);
```

Define the following parameters:

P = number of processors
R = number of rows (KMX-KMN+1)
C = number of columns (LMX-LMN+1)
K = number of cross-cache references within one row of a column group
S = time for a single inner loop iteration with no cache misses
V = time to move one word from one cache to another
T = total time spent on all iterations on single processor

First, observe that:

$$T = R \cdot C \cdot S$$

Now, there are K cross-cache references within one row of a column group. There are R rows and P column groups. Each cross-cache move takes time V. So, the total time spent moving cache words on the multiprocessor is $K \cdot V \cdot P \cdot R$. But, this time is divided evenly among the P processors, so the cache word moving overhead contribution to the elapsed time is $K \cdot V \cdot R$. Assuming this is the only overhead and that the normal iteration time is also divided evenly among the processors, the speedup can be expressed as:

$$\text{Speedup} = \frac{T}{\frac{T}{P} + K \cdot V \cdot R} = \frac{P}{1 + \frac{K \cdot V \cdot P \cdot R}{T}}$$

$$= \frac{P}{1 + \frac{K \cdot V \cdot P}{C \cdot S}}$$

For the above code fragment, K can be computed by simply counting the number of different accesses of adjacent columns, i.e. column L-1 or L+1. In this case, K = 12 (not counting duplicate references to the same off-column element). The average value of V was estimated earlier to be about 300 nanoseconds. The value of S for this loop could be about 1200 nanoseconds. So, the speedup can be estimated:

$$\text{Speedup} = \frac{P}{1 + \frac{3 \cdot P}{C}}$$

Now, again letting P=16, and choosing the same sample values for R and C as for the sweep analysis, the following speedup estimates are obtained:

R	C	speedup
128	128	11.6
128	1024	15.3
1024	1024	15.3

And finally, a speedup estimate for the entire code can be computed, assuming that the sweep speedup is a good estimate of the heat conduction pass speedup and that the sample hydrodynamics loop speedup is a good estimate of the hydrodynamics pass speedup. The heat conduction pass consumes about 30% of SIMPLE CPU time, and the hydrodynamics pass consumes about 70%. The speedups are combined using the equation:

$$\text{Speedup} = \frac{100}{\frac{\text{percent1}}{\text{speedup1}} + \frac{\text{percent2}}{\text{speedup2}}}$$

This yields the following entire code speedup estimates:

R	C	speedup
128	128	9.7
128	1024	13.9
1024	1024	14.9

10 Synchronization and communication

The above studies have pointed out that a variety of process synchronization and communication mechanisms may be desirable for use under varying circumstances. The most obvious form of communication between processes on a multiprocessor like the S-1 is through the use of shared memory, which is implemented on the S-1 multiprocessor as several shared main memory modules and a cache coherence algorithm to keep the state of main memory and local caches consistent throughout all read and write accesses.

Shared memory does not necessarily directly implement the desired synchronization primitives, however. The (statically) most frequent synchronization primitive used in partitioned SIMPLE is the synchall call described earlier. Recall that it forces each process to wait at a given point of execution until all processes have arrived, after which all processes may continue. Synchall can be easily implemented in terms of classic Dijkstra-style P and V semaphore operations. For example, letting MAXPROC be the number of processes, if SLEEPINGPROCS is of type integer and MUTEX and SLEEP[1..MAXPROC] are semaphores, the following code can be used to implement synchall on process number PR:

```
(* Initially SLEEPINGPROCS=0, MUTEX=0, SLEEP[1..MAXPROC]=0 *)
P(MUTEX);
SLEEPINGPROCS := SLEEPINGPROCS + 1;
if SLEEPINGPROCS = MAXPROC then
   begin
   for I := 1 to MAXPROC do V(SLEEP[I]);
   SLEEPINGPROCS := 0;
   end;
V(MUTEX);
P(SLEEP[PR]);
```

The performance of this code in practice would of course depend very greatly on the underlying implementation of the P and V primitives. Also, it is important to note that in this code one process (the last one to execute the synchall) is responsible for issuing the V's that wake up *all* of the other processes. If the CPU time required for executing a V primitive is large enough compared to the CPU time between synchalls, and if the number of processes is large enough, this can be a severe performance bottleneck.

For allowing the implementation of synchronization primitives, the S-1 architecture contains "conditional move" instructions. One such is the MOVCSF ("move conditionally, skip on failure") instruction. This instruction tests to see if the values of its first and second operands are equal. If so, the contents of the first operand are replaced by the contents of register %12 (decimal). If not, the first operand is left unchanged and a skip is taken to the skip destination. The instruction operates indivisibly, so that nothing can change the value of the first operand before it is (conditionally) replaced.

Synchall can also be implemented in terms of the MOVCSF instruction. The following

example implementation is written in S-1 assembler code. It is implemented at a very low level, without any operating system calls such as might be desired for a more general implementation - all waiting is busy-wait looping. Notice that the process local index SW is used to toggle between the first and second words of SLEEPINGPROCS on successive synchalls, to avoid race condition trouble if one process reaches its next synchall before another process has realized it is time to wake up from the previous synchall.

```
;;; Initially (SW)=0, (SLEEPINGPROCS)=0, (SLEEPINGPROCS+4)=0
INCSLEEP: MOV A,SLEEPINGPROCS(SW)
          INC %12.,A
          MOVCSF SLEEPINGPROCS(SW),A,INCSLEEP   ;Increment SLEEPINGPROCS indivisibly
          SKP.NEQ %12.,MAXPROC,SLEEP
          MOV SLEEPINGPROCS(SW),#0              ;If incremented to MAXPROC, zero it
SLEEP:    JMPZ.NEQ SLEEPINGPROCS(SW),SLEEP      ;Wait for SLEEPINGPROCS = 0
          SUBV SW,SW,#4                         ;Switch: SW:=4-SW
```

For some kinds of synchronization and communication, it appears that a mechanism other than simple shared memory is very desirable. The cache line size of 16 words requires a substantial amount of overhead per cache line moved from one processor to another. This overhead can be amortized over the 16 words if the memory access pattern causes most of the 16 words to be used before the cache line must be moved again. This type of amortization is the reason that SIMPLE arrays were assumed to be stored by columns, and then the rows were processed in blocks in the heat conduction sweep analysis. In a straightforward non-blocked implementation, the sweeps in SIMPLE would require that about 4 words per C/P microseconds be transferred between processors. Especially for small numbers of columns, the "bulky" 16 word cache moves can be a significant bottleneck.

Also, timing cache line mesh data moves only includes communication overhead, and does not account for any synchronization overhead (mentioned in the "partitioning SIMPLE" section as AWAIT and SIGNAL primitives). So, it is reasonable to propose a new general purpose mechanism which combines the functions of communicating small packets of data at high bandwidth and providing synchronization between the processes sending and receiving the data.

The new proposal is a simple inter-processor message sending mechanism. Messages are transmitted on one-way "links," which are allocated in I/O memory space much like normal I/O mechanisms. The I/O memory allocation is performed by the operating system, so that transparent reallocation can be done if it becomes necessary to move a process from one processor to another. Once the link is set up, the user processes can use it at high speed via special instructions without substantial operating system intervention.

The user instructions are called SNDMSG and RCVMSG. They are specified to operate on small messages (doublewords) at very low overhead per message transmission. The hardware contains a small amount of buffering for smoothing the message flow, but both instructions have failure returns, indicating that either the buffers are momentarily full (for SNDMSG) or empty (for RCVMSG). It is expected that both instructions can execute in the 100-200 nanosecond range, with

a message latency between processors limited largely by physical factors such as interprocessor cable lengths.

As an example, a possible implementation of the forward sweep part of the slightly simplified SIMPLE code fragment which appeared earlier using AWAIT and SIGNAL is included below:

```
for K := KMN to KMX do
   begin
   if PR > 1 then RCV2WORDS(LINK[PR-1],AKLM1,BKLM1) else
      begin
      AKLM1:=A[K,LMN[1]-1];
      BKLM1:=B[K,LMN[1] 1];
      end;
   for L := LMN[PR] to LMX[PR] do
      begin
      A[K,L] := Q[K,L] / AKLM1;
      B[K,L] := (Q[K,L-1]*BKLM1) / AKLM1;
      AKLM1 := A[K,L];
      BKLM1 := B[K,L];
      end;
   if PR < MAXPROC then SEND2WORDS(LINK[PR],AKLM1,BKLM1);
   end (*for K*);
```

11 Directions for future study

In a broad ranging study such as this, there will always remain interesting problems to be addressed. There is more work to be done in each of the areas discussed in this report, and there are also many other related areas requiring study.

The simulation of partitioned SIMPLE is still incomplete in several ways mentioned earlier, especially in simulating the overhead time required for cache line moves and/or synchronization primitive execution. Also, a higher-level simulation could be done which does not actually solve the physics equations, but still models the multiprocessor behavior of the code for various mesh sizes and other parameters. The analytic studies of SIMPLE could be continued in several directions, including detailed analysis of other code sections, or studying previously analyzed sections under different assumptions, such as the use of SNDMSG and RCVMSG primitives. A more quantitative statement about the sensitivity of the speedup of various code segments to variations in the mesh size would also be useful.

Further detailed study of synchronization and communication mechanisms is desirable. Such mechanisms should be as easy to use and as general as possible, but must not sacrifice performance. It has already been observed that a variety of mechanisms with a variety of functional and timing characteristics is likely to be needed. In conjunction with these mechanisms, more study should be done on general techniques for partitioning of applications. The special issues arising in debugging a multiprocess implementation are particularly important. More tools need to be developed for evaluating the effectiveness of alternative implementations.

Another important dimension of study is the range of applications chosen for partitioned implementation. Study of partitioning in detail should be done (as it was for SIMPLE) for several other real-world applications, such as those mentioned in the section on "selection of a sample application." Also, several entirely different non-numerical areas of application should be considered in more detail for multiprocessing feasibility.

One final area of investigation needed is the implications of trying to use the S-1 multiprocessor hardware as cost-effectively as possible. A major topic is the interaction of the partitioned multiprocessing approach with the powerful vector processing capabilities of the S-1 Mark IIA. One other topic mentioned earlier is researching the possible implementation of a very efficient multiprocessor matrix transposition algorithm, for possible use in situations where matrix processing does not efficiently align with the chosen data partitioning of matrices.

12 Conclusion

This study to date has added to the evidence in favor of the partitioned application mode of multiprocessor use. It has demonstrated that applications representative of real-world large scale problems can reasonably be considered for multiprocessor partitioning. Some simulation and analytic estimates of code speedup have been obtained. Some general methodologies for partitioning have been suggested, and some specific mechanisms for multiprocess cooperation have been proposed.

It seems certain that general purpose multiprocessors will play a large role in the future spectrum of the world's computing needs. Part of this role will be assumed by large scale multiprocessors executing some of the most compute-intensive applications, partitioned across multiple processors to gain valuable increases in raw computing power per wallclock hour.

13 Acknowledgments

I am indebted to many of my fellow staff members of the S-1 Project for encouraging this research, and particularly to Mike Farmwald, whose stimulating advice and helpful comments from beginning to end of this work have been most warmly appreciated. Professor Gio Wiederhold also provided much useful guidance. I am also grateful to the Fannie and John Hertz Foundation for a Hertz Fellowship, which provided personal support during much of the time that this research was conducted.

The S-1 Project is sponsored by the Naval Material Command, through the Office of Naval Research and the Naval Electronic Systems Command. The Lawrence Livermore Laboratory is operated by the University of California for the US Department of Energy under Contract W-7405-Eng-48.

14 References

[1] Baudet, Gerard M., The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. thesis, Carnegie-Mellon University, April 28, 1978.

[2] Chazan, D. and W. Miranker, Chaotic Relaxation, *Linear Algebra and Its Applications*, 2, 1969, 199-222.

[3] Crowley, W. P., C. P. Hendrickson, and T. E. Rudy, The SIMPLE Code, Lawrence Livermore Laboratory report no. UCID-17715, February 1, 1978.

[4] Flynn, M. J., Very High-Speed Computing Systems, *Proceedings of the IEEE*, 54(12), December 1966, 1901-1909.

[5] Hammersley, J. M., and D. C. Handscomb, *Monte Carlo Methods*, Wiley, N. Y., 1964.

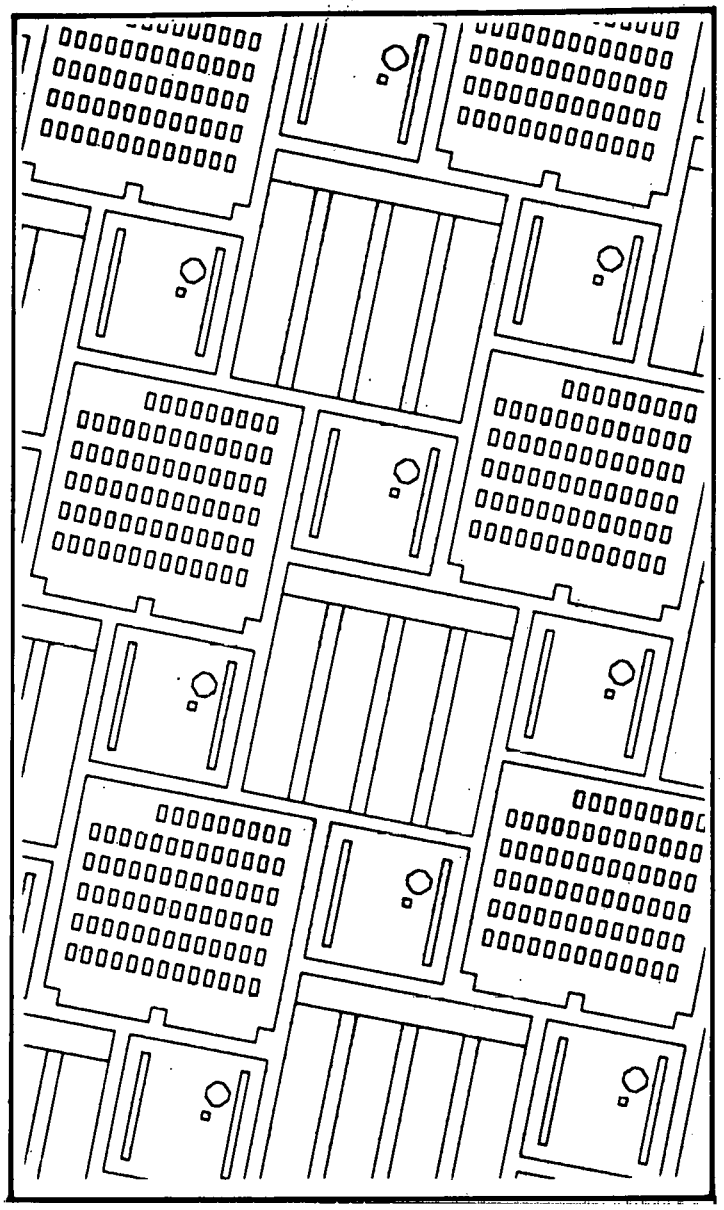
[6] Jones, Anita K. and Peter Schwarz, Experience Using Multiprocessor Systems: A Status Report, technical report, Carnegie-Mellon University, October 14, 1979.

[7] Raskin, Levy, Performance Evaluation of Multiple Processor Systems, Ph.D. thesis, Carnegie-Mellon University, August 1978.

[8] S-1 Project Staff, Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project, Lawrence Livermore Laboratory report no. UCID-18038, 1978.

[9] Widdoes, L. C., High-Performance Digital Computer Development in the S-1 Project, *Proceedings of IEEE CompCon*, Spring 1980, in press.

4



S-1 Uniprocessor Architecture (SMA-4)

Steven Correll

1 Introduction

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

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

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

1.1 Notation

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

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

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

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

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

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

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

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

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

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

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

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

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

SIGN_EXTEND(X)
says to extend the precision of X by replicating its sign bit.

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

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

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

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

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

and:

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

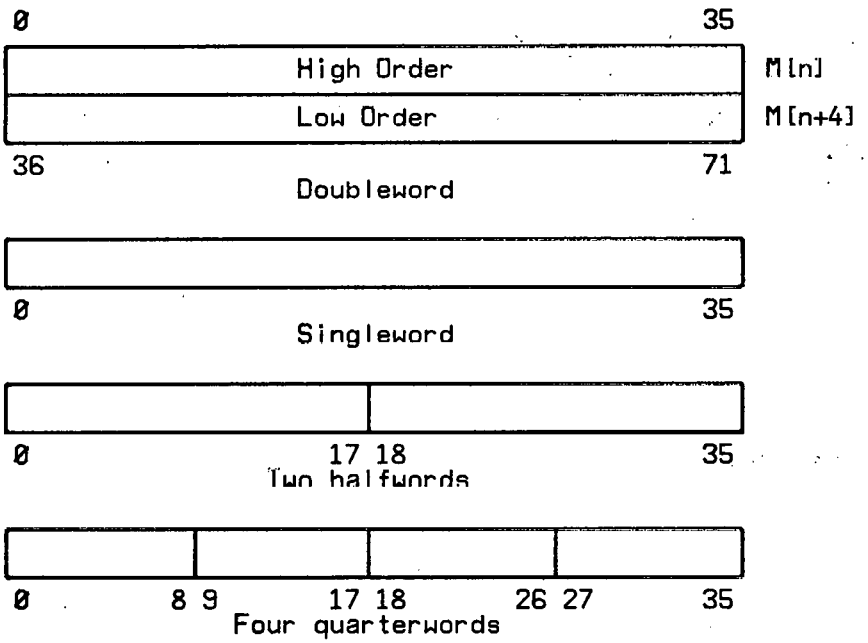
Data literals are discussed in section 3.3.4.

1.2 Words, Memory, and Registers

1.2.1 Words

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

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



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

```
ADD.Q
```

adds quarterwords while

```
ADD.D
```

adds doublewords.

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

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

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

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

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

1.2.2 Memory

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

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

More precisely, ADDRESS(X) is a singleword in the form of a *pointer*, as described in Section 1.8.1: a five-bit *tag* field, one of whose purposes is to specify a ring, followed by a 31-bit address field which can address any quarterword in an entire 2^{29}-singleword space. Thus, ADDRESS(X) specifies both a tag and a quarterword address.

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

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

PRIVILEGE_VIOLATION hard trap (Section 1.9.4).

1.2.3 General Purpose Registers

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

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

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

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

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

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

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

SIZEREG: Register R3 is also used to specify the lengths of vectors, and is then called SIZEREG.

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

When an instruction accesses RTA as a doubleword, it obtains both R4 and R5; we often refer to

R5 as "RTA1". Similarly, we often refer to R7 as "RTB1".

Stack frame and closure pointers: One of the subroutine calling mechanisms provided by the architecture maintains stack frames by using register R28 as a closure pointer and R29 as a frame pointer (Section 2.12).

Stack pointer/limit: Traps, interrupts, and subroutine calling instructions all use an upward-growing stack in memory to store return addresses and other context information. ("Upward-growing" means that pushing an item increases the address of the top of the stack.) R30 and R31 serve as the stack pointer and stack limit registers for this particular stack, and are also called SP and SL respectively. SP points to the first free location on the stack. SL points to the first location past the end of the area reserved for the stack. (The instruction set makes it easy to use other registers or even memory locations as stack pointer/limit pairs to implement additional stacks for other purposes, as described in Section 2.11. But when we talk about "the stack" rather than "a stack", we mean the stack whose pointer is register SP.)

The table below summarizes the uses of the registers.

Register	Special characteristics
R0 . . R2	Cannot be base for pseudoregister mode
R3	When used as base gives program counter instead; also used to specify vector length
R4, R5	RTA area
R6, R7	RTB area
R8 . . R27	None
R28, R29	Closure and frame pointers, CP and FP
R30, R31	Subroutine stack pointer/limit, SP and SL

1.3 Program Counter

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

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

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

1.4 Processor and User Status Registers

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

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

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

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

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

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

REGISTER_FILE
Determines which of the sixteen register files is currently available to unprivileged processes. See Section 2.15.

PRIORITY Determines what priority an interrupt must have in order to interrupt the processor. See Section 2.13.

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

does the clearing. See Section 1.11.

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

CALL_TRACE_ENB

Analogous to TRACE_ENB, this bit enables a separate trap for tracing instructions which call subroutines and return from them. Section 1.11 details the behavior of the trap and Section 2.12 enumerates the instructions to which it applies.

CALL_TRACE_PEND

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

UNMAPPED_MODE

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

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

FLAGS This field is available for use by software.

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

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

Bits	Purpose
0	CARRY
1 . . 2	FLT_OVFL_MODE
3 . . 4	FLT_UNFL_MODE
5 . . 6	FLT_NAN_MODE
7	INT_OVFL_MODE
8	INT_Z_DIV_MODE
9 . . 13	RND_MODE
14	FLT_OVFL
15	FLT_UNFL
16	FLT_NAN
17	INT_OVFL
18	INT_Z_DIV
19	FLT_REP
20 . . 31	Reserved
32 . . 35	FLAGS

The fields which deal with integer arithmetic (CARRY, INT_OVFL, INT_Z_DIV, INT_OVFL_MODE, and INT_Z_DIV_MODE) are described in Section 2.1 and the fields which deal with floating point arithmetic (FLT_OVFL, FLT_UNFL, FLT_NAN, FLT_REP, FLT_OVFL_MODE, FLT_UNFL_MODE, FLT_NAN_MODE, and RND_MODE) are described in Section 2.3.

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

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

1.5 Instruction Formats

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

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

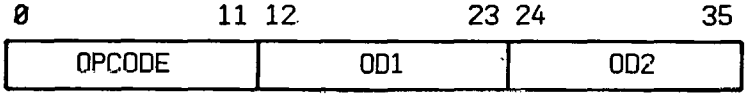
If an instruction takes more than one modifier, the order of the modifiers is significant. If one modifier refers to the first operand and the other to the second, the modifier for the first operand comes first. For example, MOV.S.Q converts a quarterword to a singleword whereas MOV.Q.S converts a singleword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

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

Operands: Most instructions specify operands by means of an *operand descriptor* (OD), a 12-bit field that can indicate a constant, a register, a memory location anywhere within the 2^{29} singleword address space, or indexed addressing using some combination of constants, registers, and memory.

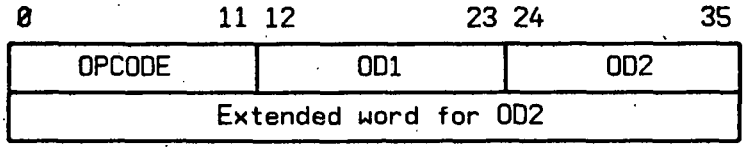
Sometimes the OD itself suffices to encode the operand--a small constant or a register, for example. Such an operand is called a *short operand* or SO. Obviously, more elaborate operands require more than twelve bits, so frequently an operand descriptor will tell the processor to use a word following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

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



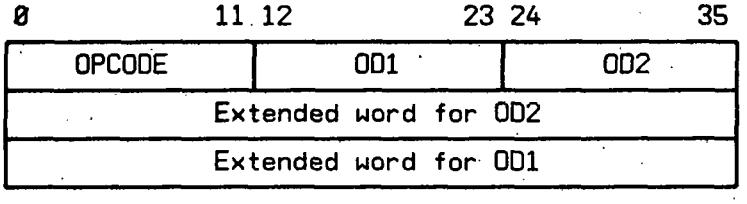
Both operands fit inside ODs

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



OD2 calls for extended word

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



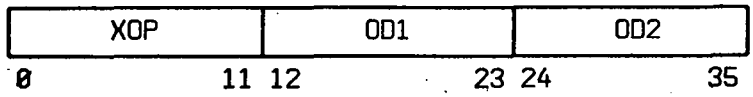
Both operands call for extended words

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

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

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

1.5.1 Two-address Format (XOP)



XOP Format

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

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

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

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

The following example uses INC more flexibly:

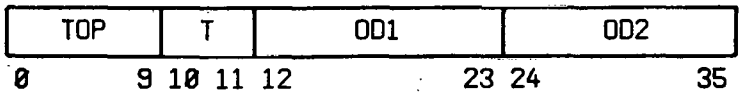
```
INC COSTPLUS1,COST   ; COSTPLUS1 := COST + 1
```

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

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

When an XOP instruction stores results in both operands, it stores OP2 first (see the example under the EXCH instruction in Section 2.7).

1.5.2 Three-address (TOP) Format



TOP Format

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

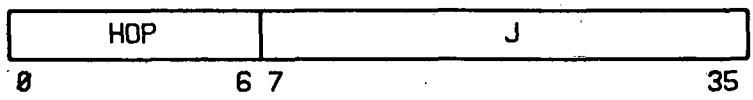
```
ADD X,X,Y              ; X := X + Y
ADD X,Y                ; X := X + Y

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

When an ordinary TOP instruction stores more than two results, it stores S2 before S1 and S1

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

1.5.3 HOP Format



HOP Format

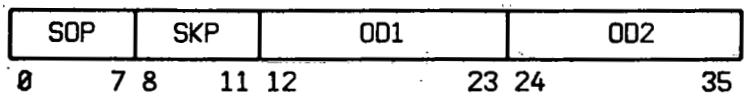
A single instruction, SJMP, uses this format to jump to a location relative to the current program counter. The processor uses the "J" field as an unsigned displacement, expressed in singlewords. The address calculation "wraps around" if it exceeds the maximum address:

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

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

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

1.5.4 Skip (SOP) Format



SOP Format

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

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

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

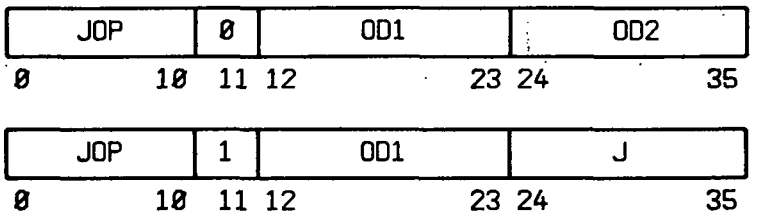
To use a SOP instruction in FASM, simply provide a label for the skip destination. The assembler will automatically subtract the current location to compute the offset.

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

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

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

1.5.5 Jump (JOP) Format



JOP Format

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

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

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

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

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

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

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

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

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

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

1.5.6 Vector Instructions

Vector instructions generally use the same format as XOP instructions. OD1 and OD2 are operand descriptors which may specify either scalars or vectors, depending on the particular instruction.

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

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

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

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

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

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

V:=VS

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

SS:=VV

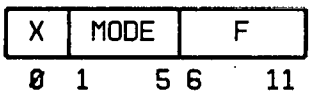
1.6 Operand Descriptors

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

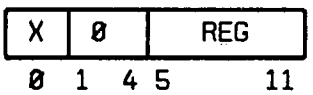
1.6.1 Subfields of an Operand Descriptor

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

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



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



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

Constant EW

Simple-base EW

Complex-base EW

1.6.2 Constant Operands

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

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

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

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

```
ADD.S A,#[-5]
ADD.S A,#[TABLESIZE-1]
ADD.S A,#[1061251031131
```

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

The precision of an instruction is inherent in the opcode, not the operands, so a constant in either

SO or LO format is ordinarily converted from a 36-bit entity to the desired precision at execution time, either reducing precision by discarding high-order bits or increasing precision by extending the sign bit.

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

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

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

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

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

NOTATION

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

SHORT OPERAND CONSTANTS

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

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

LONG OPERAND CONSTANTS

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

Figure 1-1
Constant Operand Formats

1.6.3 Short Operand Variables

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

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

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

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

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

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

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

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

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

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

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

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

NOTATION

Symbol	Meaning	
r	0 . . 127	register
pr	12 step 4 until 124	pseudoregister base
sao	-128 step 4 until 124	short aligned offset
R[x]	Contents of register location x	
M[x]	Contents of memory location x	
B[x]	Evaluate x as a base pointer; if x=R3 use PC instead	

SHORT OPERAND VARIABLES

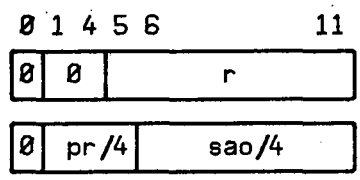
FASM notation	Evaluation	OD Format
		
%r	R[r]	
(%pr)sao	M[B[R[pr]]+sao]	

Figure 1-2
Short Operand Formats

1.6.4 Long Operand Variables

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

Fixed-base: This mode uses a 31-bit field to specify a base address in memory. (The tag is implicitly that of the ring in which the instruction is executing; no field is provided to encode a tag explicitly.) One may either use the entity at that address as the operand, or treat it as a new base pointer for indirect addressing:

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

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

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

NOTATION

Symbol	Meaning	
ar	0 step 4 until 124	aligned register
la	$0 . . 2^{31}-1$	long address
sd	$-2^{25} . . 2^{25}-1$	short displacement
M[x]	Contents of memory location x	
R[x]	Contents of register location x	
B[x]	Evaluate x as a base pointer; if x=R3 use PC instead	

LONG OPERAND VARIABLES

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

Figure 1-3
Long Operand Variable Formats

1.6.5 Combined Long and Short Operand Variables

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

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

(BASE POINTER)OFFSET[INDEX]↑SHIFT

Unless otherwise mentioned, the base pointer is a singleword pointer (that is, it has a tag in the upper five bits and an address in the lower 31 bits.) The offset and index values are added to the 31-bit address using modulo 2^{31} arithmetic. This means that the sum cannot overflow into the tag

field, and that when the offset is 31 bits long, one may regard it either as a signed value or as an unsigned value that "wraps around" the virtual address space.

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

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

(BASE POINTER)OFFSET@[INDEX]↑SHIFT

or afterward:

(BASE POINTER)OFFSET[INDEX]↑SHIFT@

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

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

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

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

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

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

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

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

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

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

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

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

(SF)TABL

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

(SF)TABL@

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

(SF)TABL@[(SF)VI]↑2

The entire loop might look like this:

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

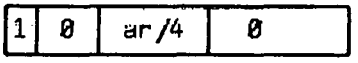
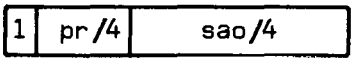
(We assume VI and TABL are not too large to fit within this operand format, and that the value of I is not used again following the loop.)

NOTATION

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

COMBINED LONG AND SHORT OPERAND VARIABLES

Substitute either of these short operands . . .

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

. . . for "SO" in the following:

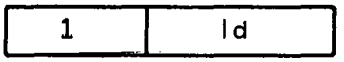
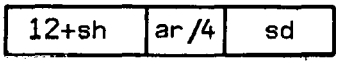
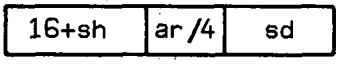
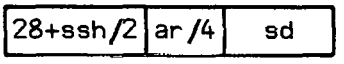
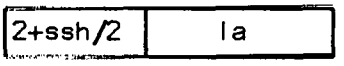
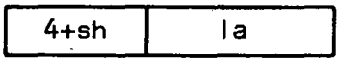
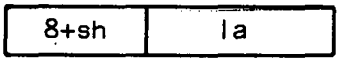
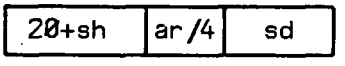
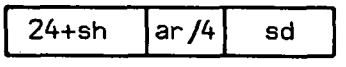
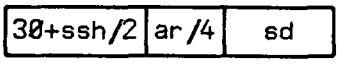
FASM notation	Evaluation	EW Format
(SO)ld	M[B[SO]+ld]	
(SO)ld@	M[B[M[B[SO]+ld]]]	
(SO)sd[%ar]↑sh	M[B[SO]+sd+R[ar]*2↑sh]	
(SO)sd@[%ar]↑sh	M[B[M[B[SO]+sd]]+R[ar]*2↑sh]	
(SO)sd[%ar]↑ssh@	M[B[M[B[SO]+sd+R[ar]*2↑ssh]]]	
la[SO]↑ssh@	M[B[M[B[la]+SO*2↑ssh]]]	
la[SO]↑sh	M[B[la]+SO*2↑sh]	
la@[SO]↑sh	M[B[M[B[la]]]+SO*2↑sh]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

Figure 1-4
Combined Long and Short Operand Variable Formats

1.6.6 NEXT Versus FIRST/SECOND

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

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

Certain other instructions deal with pairs of elements: the operand and the single element following that operand. For example, the DIV instruction divides two integers, stores the quotient in operand DEST, and stores the remainder in the element following DEST. In these cases, we use the terminology "FIRST(*x*)" and "SECOND(*x*)" to describe the operand *x* and its successor. If the precision of the instruction is quarterword or halfword, then the two elements must align together to form a single entity of twice that precision.

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

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

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

When an operand *x* described in terms of FIRST/SECOND is a constant and the precision of the instruction is quarterword, halfword, or singleword, the instruction expands the constant to twice that precision, uses the high order half as FIRST(*x*), and uses the low order half as SECOND((*x*). (When expanding a singleword constant to a doubleword, it observes the special constant addressing modes for doing so.) For instance, the BNDSF.B instruction is a TOP which sets its destination true or false according to whether S2 lies within the bounds specified by FIRST(S1) and SECOND(S1), so the following example:

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

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

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

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

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

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

1.6.7 Forbidden Operand Formats

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

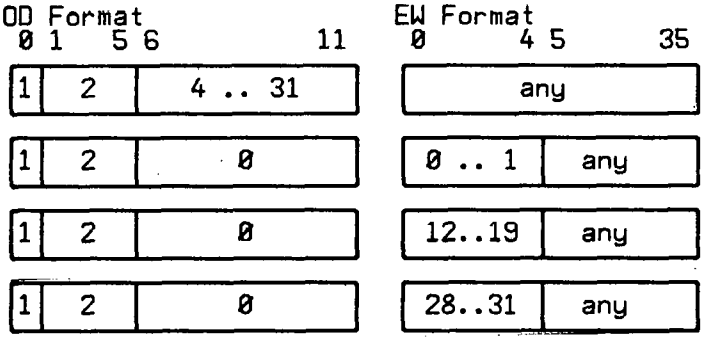


Figure 1-5
Forbidden Operand Formats

1.7 Virtual to Physical Address Translation

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

1.7.1 Paging

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

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

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

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

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

The Descriptor Segment, of course, contains nothing but pointers to segmentitos that make up the 4

virtual address spaces. In fact, this page of pointers is identical in form to the Descriptor Segmentito Table, except that it has more entries and the entries point to pages inside one of the virtual address spaces instead of inside the Descriptor Segment. Thus, we have labeled it a "Target Segmentito Table." (Note, however, that the page shown is probably only one of many pages of segmentito pointers required to describe the entire ring, and that the Descriptor Segment is a continuous list of such pointers, not a separate table for each ring.) Bits 6 . . 14 select one STE from this table, which points to the physical address of the first word of a Target Page Table, which has an entry for each of the 16 pages comprising that segmentito.

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

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

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

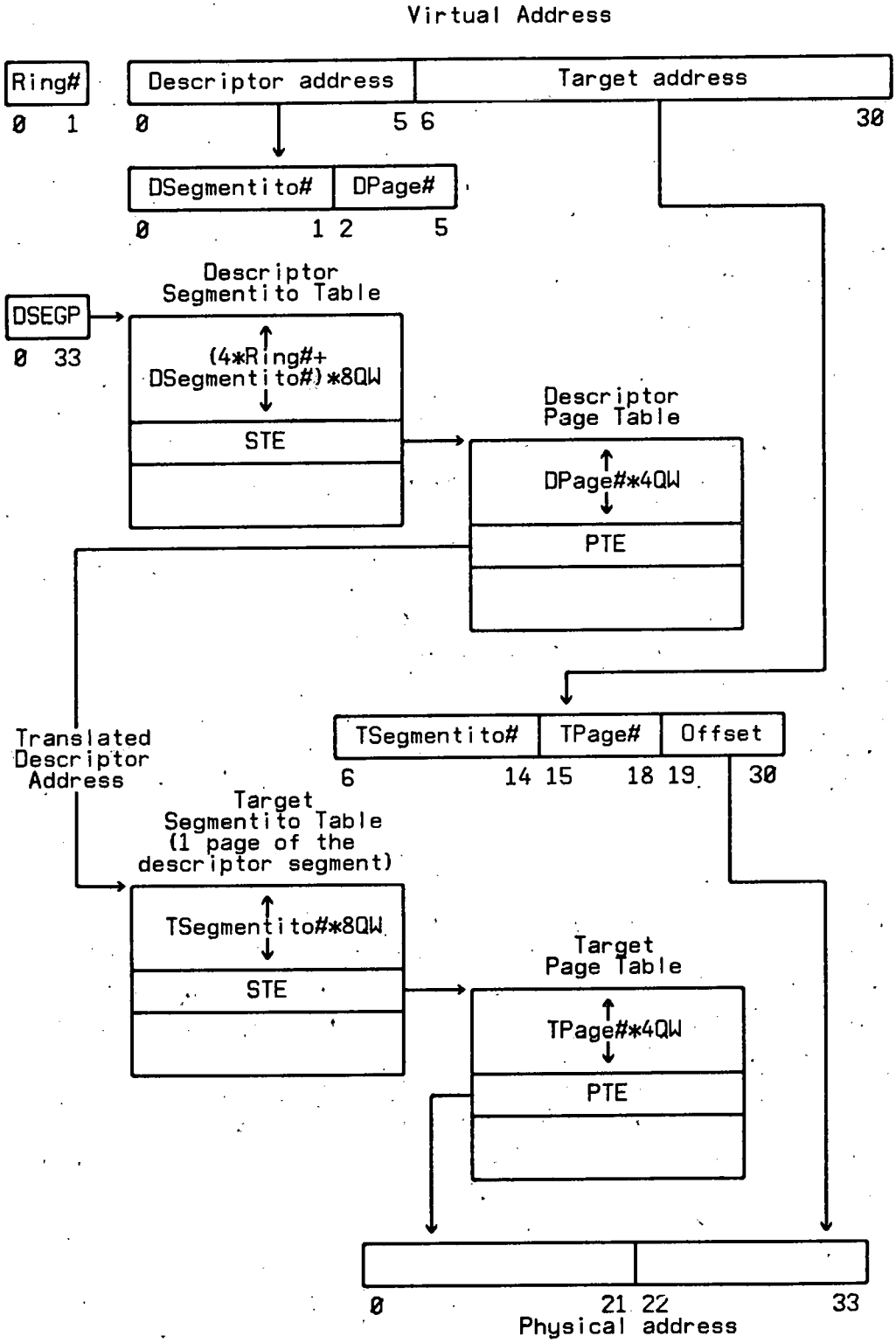


Figure 1-6
Virtual-to-physical address translation

1.7.2 Segmentation

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

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

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

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

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

Segment bounds checking: Every memory address calculation begins with a base pointer, establishing which segment is being addressed. The rule for bounds checking is simply that a

memory access must lie within the same segment as the previous base pointer. Thus, the base pointer plays the role of address *x* in the previous example, and the actual operand being accessed serves as *y*.

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

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

1.7.3 Segmentito and Page Table Entries

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

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

PTA Singleword physical address of the corresponding page table.

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

EB Execute bracket. Attempting to execute this segmentito from a ring (or, more formally, with a validation level) greater than EB causes an ACCESS_VIOLATION hard trap. Note that a cross-ring call via the instruction CALLX and the gate mechanism (Section 2.12.2) is not considered an attempt to execute the called segmentito, and is thus exempt from EB restrictions.

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

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

segmentito.

SIZE Specifies the size of the segment that contains this segmentito, expressed as a base 2 logarithm of the number of pages in the segment (for example, SIZE=8 indicates the segment contains 2^8 pages, which is 2^3 segmentitos). SIZE must not be less than 4 (2^4 pages, or 1 segmentito) or greater than 19 (2^{19} pages, or 15 segmentitos, or an entire 2^{31} quarterword address space.)

FLAGS Reserved for use by software.

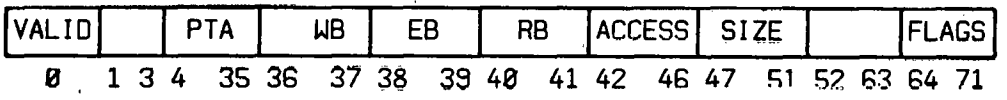


Figure 1-7
Segmentito table entry format

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

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

USED If VALID=1, this bit indicates the page has been accessed. (More precisely, the processor sets this bit when it brings into the map cache (Section 2.14) the mapping information for this page.)

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

FLAGS Reserved for use by software.

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

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

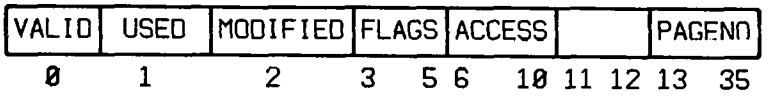


Figure 1-8
Page table entry format

Access modes: The access permitted for a particular page is the logical AND of the ACCESS fields in the STE and the PTE for that page. They permit an operating system to mark a page for read-only access, write-only access, execute-only access, or any combination of reading, writing, and execution. An instruction which attempts to access a memory location in violation of these markings will cause an ACCESS_VIOLATION hard trap. (Of course, the attempted access must pass the checking defined by the RB, EB, and WB fields in the STE, too.) Within each ACCESS field, the bits have the following meanings:

WRITE_PERMIT
Instructions may alter this segmentito/page.

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

READ_PERMIT Instructions may read from this segmentito/page.

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

Figure 1-9
Bits in ACCESS field

1.8 Rings and Protection

The uniprocessor architecture provides three principal kinds of protection.

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

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

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

1.8.1 Pointer Format

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

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

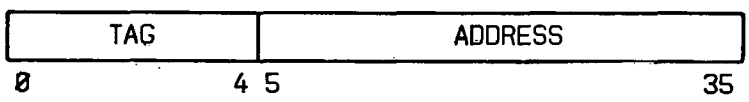


Figure 1-10
Pointer Format

Various values of the TAG field have the following meanings:

Tag	Meaning
0	Fault. When an instruction attempts to access memory through this pointer, or when the instructions MOVP or BASEPTR attempt to manipulate this pointer, a BAD_POINTER_TAG hard trap occurs.
1	Gate. As explained in Section 2.12.2, the CALLX instruction can use a pointer with a gate tag to implement a procedure call from one ring to another. If any instruction attempts to use such a pointer to reference memory, however, or if the BASEPTR instruction attempts to manipulate such a pointer, a BAD_POINTER_TAG hard trap occurs. The MOVP instruction may, however, move such a pointer.
2	NIL. If an instruction attempts to use this pointer to reference memory, or if the BASEPTR instruction attempts to operate on this pointer, a BAD_ADDRESS_TAG hard trap occurs. The MOVP instruction may, however, move this pointer. A language translator such as LISP, Pascal, or PL/I may use this pointer to implement the NIL or NULL construct.
3	Reserved. Any attempt to reference memory using this pointer, or to manipulate it with MOVP or BASEPTR, causes a BAD_POINTER_TAG hard trap.
4	Ring 0 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 0 address space.
5	Ring 1 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 1 address space.
6	Ring 2 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 2 address space.
7	Ring 3 tag. An instruction which references memory through this pointer will attempt to access the specified ADDRESS within the ring 3 address space.
8 . . 30	User tag. An instruction which references memory though this pointer will attempt to access the specified ADDRESS within the same ring from which it obtained the pointer (more precisely, it will access memory using as the initial validation level the validation level derived in fetching the pointer; see Section 1.8.2.) Because these 23 tags are equivalent architecturally, software may use them for its own purposes, such as encoding the data type of the entity being addressed.
31	Fault. This behaves exactly like a tag of zero. Because all but the

largest-magnitude positive and negative integers will have either 0 or 31 in the tag field, assigning special meanings to tags of 0 and 31 increases the likelihood that the erroneous use of a random singleword as a pointer will be detected as an error.

1.8.2 Address Validation

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

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

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

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

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

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

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

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

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

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

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

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

```
MOV RTA,(R7)100.@
```

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

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

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

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

The validation mechanism discussed in this section applies to the operands of jump and call instructions as well. The PC is itself a pointer. When the PC changes due to a jump, call, or return, the new tag of the PC is the ring tag corresponding to the final validation level of the jump destination or pointer used to change the PC. This prevents an instruction executing in a higher-numbered ring from calling a routine located in a lower-numbered ring. Because such calls are needed to permit user code to obtain operating system services, the architecture provides two mechanisms that circumvent the validation scheme in a controlled fashion: the TRPEXE instruction, discussed in section 1.9, and the CALLX instruction with gates, discussed in section 2.12.2.

1.8.3 Pointer Validation

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

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

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

Pointer validation involves two steps:

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

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

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

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

   c. If the tag is a user tag and the validation level equals the number of the ring of execution, preserve the tag. (Because the pointer was obtained from the ring of execution, it cannot possibly be moving to a more protected ring. Moving it to a less protected ring is harmless; at worst, if the pointer is fetched from that ring

and used for indirection, it will appear to point to a less protected entity than it did before.)

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

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

```
MOVP.P.P TRUSTED, (RØ)
```

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

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

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

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

```
MOVP.P.P TRUSTED,RØ
MOVP.P.P TRUSTED,TRUSTED@
```

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

1.9 Traps and Interrupts

Traps and interrupts signal the processor to change its context temporarily and deal with an exceptional situation. Traps usually result from errors, while interrupts are usually invoked by external devices in need of I/O service.

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

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

- Traps which can be handled by a process at its own level of privilege. These include *soft traps* caused by errors as well as traps caused by the TRPSLF instruction.
- Traps which must be handled by privileged code. These include *hard traps* caused by errors.
- Interrupts, all of which must be handled by privileged code.
- Traps caused by the TRPEXE instruction, which are in effect calls to the executive.
- The trap-like mechanism which uses gates to make cross-ring calls (Section 2.12.2).

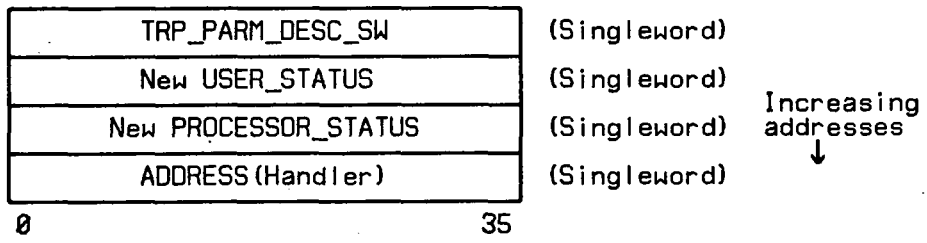
Each class of traps and interrupts has its own set of vectors. A register called the *trap descriptor block pointer* (TDBP) contains the 34-bit physical address of a series of singlewords containing ordinary tagged pointers, each of which points to the first singleword of a set of vectors:

Singleword	Points to set of vectors for:
0	Ring 0 TRPSLF traps
1	Ring 1 TRPSLF traps
2	Ring 2 TRPSLF traps
3	Ring 3 TRPSLF traps
4	Ring 0 soft traps
5	Ring 1 soft traps
6	Ring 2 soft traps
7	Ring 3 soft traps
8	Hard traps
9	Interrupts from I/O
10	Interrupts from counters
11	TRPEXE traps
12	Gate descriptor block for entering ring 0

13	Gate descriptor block for entering ring 1
14	Gate descriptor block for entering ring 2

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

Each trap or interrupt vector has the following format:



Gates are a trap-like mechanism for cross-ring procedure calls which will be described in Section 2.12.2.

1.9.1 How the Processor Responds to a Trap or Interrupt

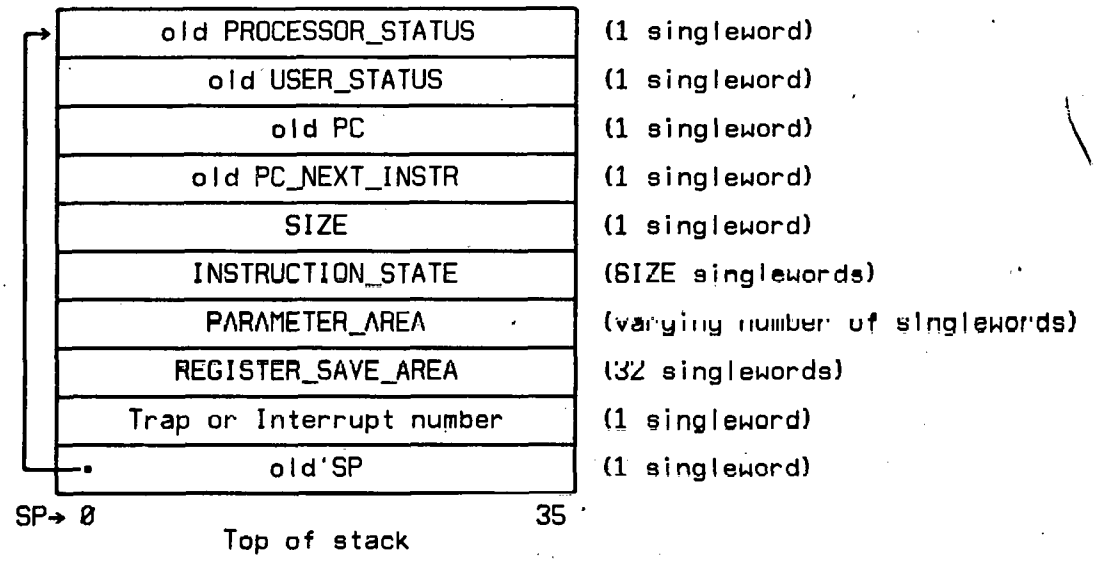
When the processor responds to a trap, it follows these steps (the same steps apply to interrupts):

1. Locate the trap vector.

   Within each set of traps, the possible traps are numbered consecutively starting at 0. When a particular trap occurs, the processor finds the appropriate trap vector using the TDBP, the pointer to the appropriate set of traps, and the number of that trap within the set. If, for example, hard trap number five occurs, the processor fetches (from the eighth singleword past the one pointed to by TDBP) a pointer to the set of hard traps, and then uses the vector located 5*4 singlewords beyond the start of that set (because each trap vector is 4 singlewords long).

2. Push the current state onto the stack pointed to by the SP in the register file specified by the new PROCESSOR_STATUS found in the trap vector. The act of pushing this information onto the stack is atomic, and any interrupts will remain pending until it is complete. A hard trap may result, however--if, for example, the SP crosses a segment boundary, exceeds SL, or touches an absent page--and such a hard trap does intercede (Section 1.9.6).

The information is pushed onto the stack in the following format, known as the *save area* for the trap:



If the trap is a soft trap or TRPSLF, it pushes a singleword zero in place of the old PROCESSOR_STATUS, because such traps are not privileged and thus may not access PROCESSOR_STATUS.

SIZE is the number of singlewords occupied by the INSTRUCTION_STATE portion of the save area. If SIZE=0, then INSTRUCTION_STATE does not appear at all. INSTRUCTION_STATE itself stores instruction-dependent and implementation-dependent information required for restarting the instruction that was in process when the trap occurred. Some instructions are said to be *interruptable*, meaning that interrupts can occur during their execution. A vector arithmetic instruction, for example, may encounter a trap or interrupt part way through the processing of the vector. INSTRUCTION_STATE would in such a case contain the information needed to proceed with the remainder of the vector after handling the trap or interrupt, since it would be wasteful or even incorrect to start over at the beginning of the instruction.

PARAMETER_AREA contains information about the cause of the trap, and varies in content and size from one trap to another. The programmer may infer the size of this area in any particular instance by comparing SP with the old SP value provided on the stack.

REGISTER_SAVE_AREA is not used by the architecture; the trap handler routine may save the registers here if it so desires.

The "old SP" pointer specifies where the top of the stack was prior to the trap (note that it points to the stack used in handling the interrupt, not necessarily the same as the stack that was in use when the trap occurred). Because the SP stack grows upward and the pointer for upward-growing stacks indicates the free location atop the stack, it turns out that "old SP" points to the beginning of the save area itself.

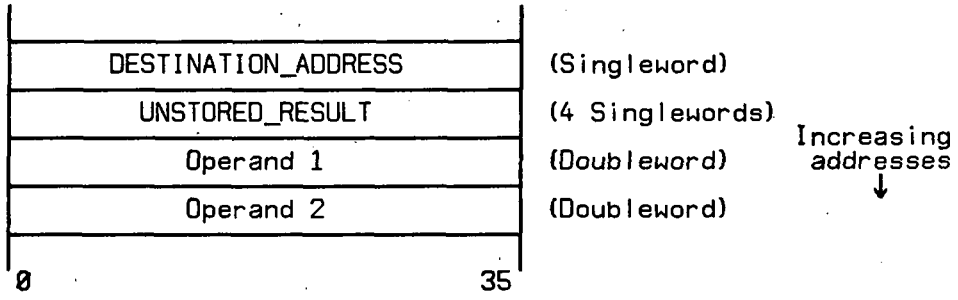
3. Load the new USER_STATUS value given by the trap. Provided the trap is not a soft trap or TRPSLF, load the new PROCESSOR_STATUS value given by the trap vector. (Because the user may be allowed to handle soft traps and TRPSLF traps within an unprivileged ring, these traps cannot alter PROCESSOR_STATUS.)

4. Jump to the trap handler specified in the trap vector. The trap handler address is a pointer, so this jump is subject to pointer validation checking, using as the initial validation level the number of the ring containing the trap vector.

1.9.2 Soft Traps

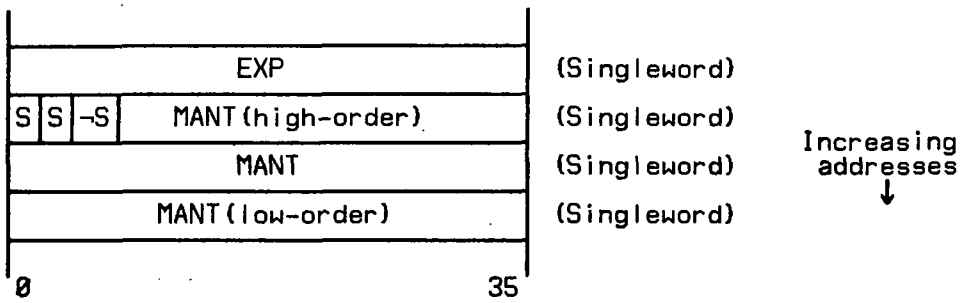
As mentioned earlier, soft traps are those which can be handled without increasing the level of privilege.

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



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

UNSTORED_RESULT is the result that would have been stored in the destination address if no trap had occurred. If it is an integer, it is sign-extended to be four singlewords long, with the most significant portion in the singleword having the lowest address. If it is a floating point value, it appears in the following format, where "S" is the one-bit sign and "¬S" is the hidden bit (Section 2.3.1):



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

Soft traps include:

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

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

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

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

4: INT_OVFL_TRAP
Integer overflow occurred and INT_OVFL_MODE=0.

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

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

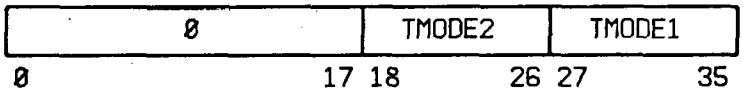
7: FFT_TOO_LONG
An FFT instruction was required to operate on a vector whose size exceeded the maximum for this implementation.

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

1.9.3 TRPSLF and TRPEXE Traps

The TRPSLF and TRPEXE instructions effectively let the user add a number of software-defined instructions to the instruction set. Simply assign a trap vector number to each new instruction and provide a corresponding trap handler routine to implement the instruction. Like XOP instructions in general, TRPSLF and TRPEXE instructions can take zero, one, or two operands. And like certain XOP instructions, they can place restrictions on whether each operand can be a constant, a quarterword, a singleword, et cetera.

The number of operands and the restrictions on operands for a particular trap are specified in a word called TRP_PARM_DESC_SW (*trap parameter descriptor singleword*) in the trap vector itself, which has the following format:



When the instruction executes, it evaluates OD1 as specified by TMODE1 and places the result in the first doubleword of the PARAMETER_AREA pushed onto the SP stack. It evaluates OD2 as specified by TMODE2 and places the result in the second doubleword of the PARAMETER_AREA. Those two doublewords constitute the entire PARAMETER_AREA for TRPSLF and TRPEXE traps.

A TMODE value outside the range 0 . . 7 causes a BAD_T_MODE hard trap to interrupt the execution of the TRPSLF or TRPEXE. TMODE values within that range have the following meanings:

0: Unused operand
The operand must be unused (that is, the descriptor must be zero) or a hard trap interrupts the execution of the TRPSLF or TRPEXE.

1: Undecoded OD Without decoding it, copy the operand descriptor into the high order half of the doubleword parameter, right-justified in a field of zeroes. Do not fetch any extended address word. This is analogous to the treatment of the JUMPDEST field in the relative form of a JOP instruction. The low-order half is undefined.

2: Undecoded OD and extended word
Without altering it, copy the operand descriptor into the high order half of the doubleword parameter, right-justified in a field of zeroes. If the descriptor calls for an extended word, copy that into the low order half; otherwise, the low-order half is undefined.

3: Virtual address

Obtain a pointer to the operand and store that, rather than the operand itself, in the high order half of the doubleword parameter. The low order half is undefined. This corresponds to the behavior of instructions like MOVP.P.A and PUSHADR. Note that the address validation mechanism must use the ring number of the ring which executes the TRPEXE, not the ring containing the vector or the TRPEXE handler. If the operand is a constant or a register, a hard trap interrupts the execution of TRPSLF or TRPEXE.

4: Quarterword Interpret the operand descriptor to obtain a quarterword and store it in the high order half of the doubleword parameter, left justified in a field of zeroes. The low order half is undefined. This treats the operand exactly as would a ".Q" instruction like "ADD.Q": thus, for example, it discards the high order bits of a constant if necessary.

5: Halfword Interpret the operand descriptor to obtain a halfword and store it in the high order half of the doubleword parameter, left justified in a field of zeroes. The low order half is undefined. This treats the operand exactly as would a ".H" instruction like "SUB.H": thus if, for example, the operand specifies a memory location, that location must be halfword aligned or a hard trap interrupts the execution of TRPSLF or TRPEXE.

6: Singleword Interpret the operand descriptor to obtain a singleword and store it in the high order half of the doubleword parameter. The low order half is undefined. This treats the operand exactly as would a ".S" instruction like "SHFA.LF.S": thus if, for example, the operand specifies a memory location, that location must be singleword aligned or a hard trap interrupts the execution of TRPSLF or TRPEXE.

7: Doubleword Interpret the operand descriptor to obtain a doubleword and store it in the doubleword parameter. This interprets the operand exactly as would a ".D" instruction like "ANDTC.D": thus if, for example, the operand specifies a memory location, that location must be singleword aligned or a hard trap interrupts the execution of TRPSLF or TRPEXE. Similarly, if the operand specifies a constant addressing mode using "!0 ?" or "? !0", the constant will be extended properly before it is placed in the doubleword.

Note that the return from a handler routine for TRPSLF or TRPEXE will ordinarily use the RETUS.A or RETFS.A instruction to avoid repeating the trap indefinitely.

1.9.4 Hard Traps

Hard traps are:

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

1: DSEG_SEGMENTITO_FAULT
The VALID field in the STE for a segmentito within the descriptor segment is zero, implying the required segmentito is not present in memory.

2: DSEG_PAGE_FAULT
The VALID field in the PTE for a page within the descriptor segment is zero, implying the required page is not present in memory.

3: SEGMENTITO_FAULT
The VALID field in the STE for a target segmentito is zero, implying the required segmentito is not present in memory.

4: PAGE_FAULT
The VALID field in the PTE for a target page is zero, implying the required page is not present in memory.

5: ACCESS_VIOLATION
Accessing an operand would have violated access mode checking (the ACCESS field within an STE or PTE) or segmentito ring bracket checking (the WB, EB, and RB fields within an STE).

6: GATE_INDEX_TOO_BIG
A cross-ring call used a gate pointer whose index exceeded the maximum index for the ring in question, or whose ring number was 3.

7: BAD_POINTER_TAG
An ordinary instruction tried to use a pointer with a fault tag or reserved tag to reference memory; or the MOVP or BASEPTR instruction tried to manipulate a pointer with a fault tag or reserved tag.

8: BAD_ADDRESS_TAG
An instruction tried to reference memory through a pointer with a NIL or gate tag, or a BASEPTR instruction tried to manipulate a pointer with a NIL or gate tag.

9: OUT_OF_BOUNDS
Accessing an operand would have violated segment bounds checking.

10: PRIVILEGE_VIOLATION
A privileged instruction attempted to execute in user mode.

11: ILLEGAL_INSTRUCTION
The instruction opcode is undefined.

12: TRACE_TRAP
The TRACE_PEND bit in PROCESSOR_STATUS is 1.

13: CALL_TRAP The CALL_TRACE_PEND bit in PROCESSOR_STATUS is 1.

14: STACK_OVERFLOW
The instruction would have caused a stack pointer to exceed the corresponding stack limit.

15: RESERVED_ADDRESS_MODE
An OD and/or its associated EW has an undefined value.

16: OPERAND_NOT_REQUIRED
An unused operand descriptor was not set to zero.

17: ALIGNMENT_ERROR
An operand was not properly aligned.

18: ILLEGAL_OPERAND_MODE
The instruction attempted to use a register as an operand where forbidden; examples are vector instructions and instructions which find ADDRESS(*x*).

19: ILLEGAL_CONSTANT
The instruction attempted to use a constant as a destination or a jump address, or attempted to find the address of the constant.

20: ILLEGAL_BYTE_PTR
The position or offset field of a byte pointer was invalid.

21: ILLEGAL_SHIFT_ROTATE
The bit count for a shift, rotate, or bit reversal instruction was negative or too large .

22: ILLEGAL_TRACE_PEND
An instruction (such as SWITCH or RETFS) is attempting to resume execution

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

23: ILLEGAL_IOMEM

An instruction attempted to access an I/O memory not attached to this uniprocessor.

24: RING_ALARM_TRAP

A ring alarm occurred upon changing the ring of execution.

25: ILLEGAL_STATUS

An instruction attempted to place an illegal value in USER_STATUS or PROCESSOR_STATUS.

26: ILLEGAL_REGISTER

One of the privileged register access instructions specified a register or register file number out of range.

27: ILLEGAL_PRIORITY

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

28: NONEXISTENT_MEMORY

The processor attempted to access memory which does not physically exist at this installation.

29: BAD_A_VALIDATION

A memory access would violate the rules for address validation.

30: BAD_P_VALIDATION

A memory access would violate the rules for pointer validation.

31: VMM_TRAP The processor was in virtual machine mode and attempted to execute any privileged instruction, or one of the user mode instructions which are specified to trap in virtual machine mode.

32: BAD_T_MODE

A TRPSLF or TRPEXE instruction found an invalid value in the TMODE1 or TMODE2 field of the trap parameter descriptor singleword.

Parameters for hard traps: Only the following hard traps push any PARAMETER_AREA within the save area:

1. DSEG_SEGMENTITO_FAULT, DSEG_PAGE_FAULT, SEGMENTITO_FAULT, PAGE_FAULT, and ACCESS_VIOLATION provide one singleword giving the virtual address, in pointer form, of the operand being referenced.

2. GATE_INDEX_TOO_BIG provides a copy of the gate pointer containing the invalid index.

3. BAD_POINTER_TAG and BAD_ADDRESS_TAG give a copy of the pointer whose tag was invalid.

4. OUT_OF_BOUNDS provides a copy of the last base pointer encountered prior to the error, followed by a singleword giving the effective offset from that pointer (which may be the sum of an offset and index) which caused the error.

1.9.5 Interrupts

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

1.9.6 Recursive Traps

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

If the original trap was a soft trap, the SP is left at its original position preceding the soft trap while the hard trap occurs. If the handler for the hard trap solves the stack problem and returns with a RETFS.R instruction, the operation which caused the soft trap is restarted and presumably the soft trap will recur, this time completing without encountering a hard trap.

If the original trap was a hard trap, the processor will halt. The front end processor must take appropriate action, since this situation indicates a serious system failure.

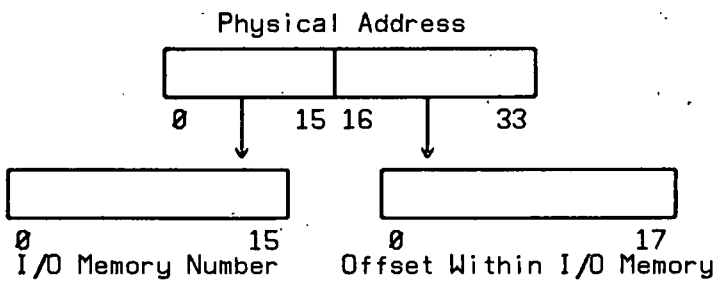
1.10 Input/output

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

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

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

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



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

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

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

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

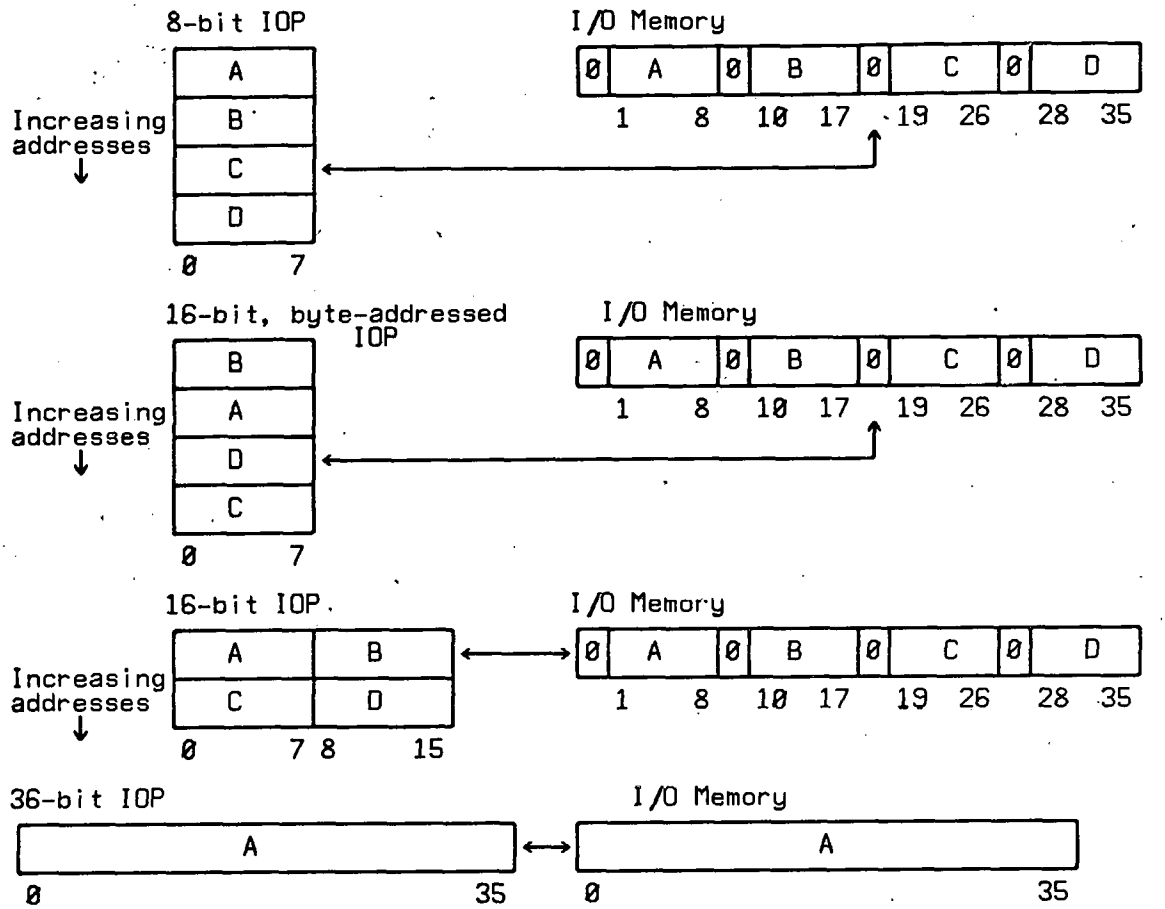
Note that setting the S-1 processor priority to 0 permits every I/O memory to interrupt, while setting it to 31 prevents any I/O memory from interrupting.

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

1.10.1 I/O Memory Translation

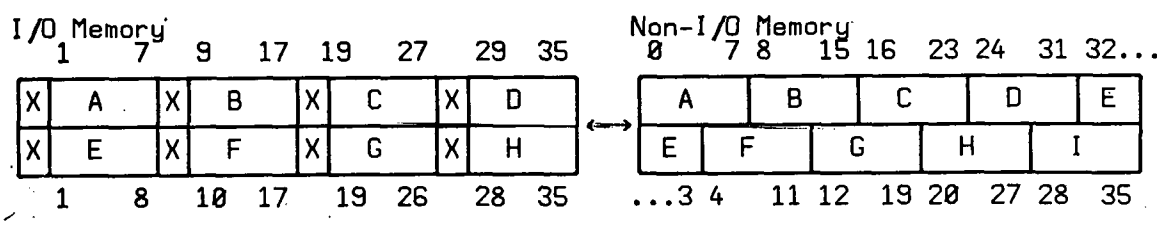
Mapping the IOP memory format onto the S-1 processor format may involve two separate transformations. First, the hardware design of the I/O memory converts the IOP format to a 36-bit singleword format. Second, certain I/O instructions translate portions of the singleword as they copy between I/O memory and non-I/O memory.

The hardware conversion will vary among IOPs, so the architecture does not specify it. But in most cases, a reasonable conversion is obvious. The following diagram shows reasonable conversions for 8-bit, 16-bit, and 36-bit IOPs:

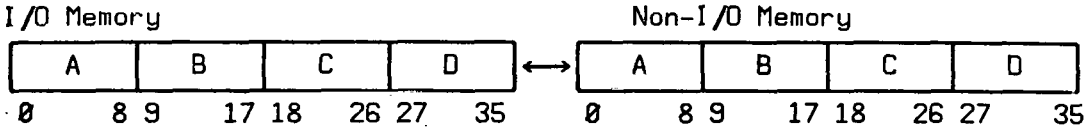


Some I/O instructions perform no further transformation, but simply copy an anyword between I/O memory and non-I/O memory. Others--the I/O instructions which use the modifiers {B,Q,H,S}--provide four different ways to translate singlewords by shifting fields within them: bitwise, quarterword, halfword, and singleword translations. In the diagrams that follow, "X" indicates that the corresponding field is ignored when an I/O instruction reads it or set to zero when an I/O instruction writes it.

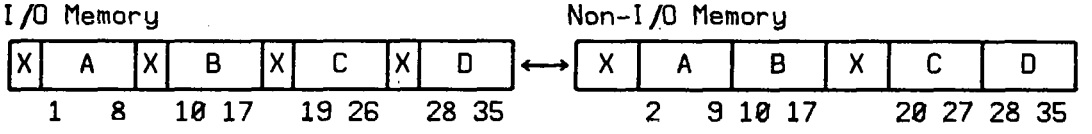
Bitwise translations map the eight low-order bits of each quarterword in I/O memory onto all 36 bits of each singleword in non-I/O memory:



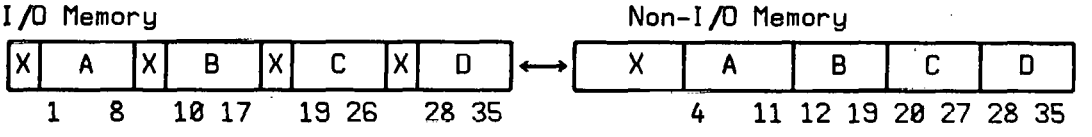
Quarterword translations map each quarterword of I/O memory onto the corresponding quarterword of non-I/O memory:



Halfword translations map the eight low-order bits of two successive quarterwords within an aligned halfword of I/O memory onto the sixteen low-order bits of a halfword in non-I/O memory:



Singleword translations map the eight low-order bits of four successive quarterwords within an aligned singleword of I/O memory onto the 32 low-order bits of a singleword in non-I/O memory:



1.11 Instruction Execution Sequence

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

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

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

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

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

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

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

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

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

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

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

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

Similarly, if the value of CALL_TRACE_ENB saved in step 4 is 1, and the instruction just executed in step 5 was a call or return (Section 2.12 defines these), then set CALL_TRACE_PEND to 1.

8. Clear the instruction state.

1.12 Mark IIA Implementation

Individual implementations of the S-1 Native Mode Architecture may vary in some respects from the description in this document. The S-1 Mark IIA Uniprocessor embodies the following:

1. Segment bounds checking does not take place during the evaluation of an operand which is fetched as an instruction rather than as data.

2. Segment bounds checking does occur when an instruction is fetched from address PC_NEXT_INSTR. Due to the instruction pipeline, the four singlewords following the first singleword of an instruction must lie within the segment and on a page with EXECUTE_PERMIT access, regardless of the number of singlewords occupied by the instruction and its operands.

3. The USED bit in a PTE may, as a result of wrong-branch evaluation in the pipeline, indicate that a page was used when in fact it was not. A similar statement applies to the MODIFIED bit.

4. Attempting to take the FFT of a vector of more than 2^{14} elements causes an FFT_TOO_LONG soft trap.

5. Only the 11 low-order bits of address space IDs are significant.

6. Instructions for which rounding is inexact guarantee their results are monotonic--that is, if $x \geq y$ then $F(x) \geq F(y)$--with an error that is less than or equal to 0.75 of the least significant bit of the mantissa. Instructions for which rounding is exact guarantee an error less than or equal to 0.5 of the least significant bit.

The following instructions exhibit inexact rounding:

FRECIP
FCMAG, VFCMAG
FSQRT, VFSQRT
FLOG
FEXP
FSIN
FCOS
FSINCOS
FATAN, FATANV
VF2DIS, VF3DIS
FCFFT, FCFFTV

7. RETFS.A will not copy CALL_TRACE_PENDING from the value of CALL_TRACE_ENABLE in the saved PROCESSOR_STATUS. If one aborts a call or return instruction, one must intervene anyway to patch up the control flow of the program, and one can explicitly reinvoke tracing. RETFS.A will handle TRACE_PENDING as specified.

2 Instruction Set

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

2.1 Signed Integer Arithmetic

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

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

2.1.1 Integer Arithmetic Exceptions

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

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

INT_OVFL Integer overflow (that is, the result is greater than or equal to MAXNUM or the result is less than or equal to MINNUM).

INT_Z_DIV Integer division by zero.

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

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

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

INT_OVFL_MODE

0	Invoke INT_OVFL_TRAP soft trap without storing a result.
1	Retain as many low-order bits of the result as possible for the precision in question, overwriting the sign bit.

INT_Z_DIV_MODE

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

2.1.2 CARRY Algorithm

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

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

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

Instruction	X1	X2	X3
ADD	S1	S2	0
ADDC	S1	S2	C_IN
SUB	S1	¬S2	1
SUBV	¬S1	S2	1
SUBC	S1	¬S2	C_IN
SUBCV	¬S1	S2	C_IN
INC	1	OP2	0
			(i.e., CARRY:=1 if OP2 = -1)
DEC	-1	OP2	0
			(i.e., CARRY:=1 if OP2 ≠ 0)
NEG	0	¬OP2	1
			(i.e., CARRY:=1 if OP2 = 0)
ABS	0	¬OP2	1
			(i.e., CARRY:=1 if OP2 = 0)

2.1.3 Signed Integer Arithmetic

ADD **Integer add**

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

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

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

ADDC — Integer add with carry

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

Purpose: DEST:=S1+S2+CARRY

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

The following adds two "quadruple-word" integers at X and Y represented as a pair of DWs with the low-order DW having the higher address. The result is stored in X and X+8:

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

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

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

SUB **Integer subtract**

SUB . {Q,H,S,D} **TOP**
SUBV . {Q,H,S,D} **TOP**

Purpose: SUB computes DEST:=S1-S2; SUBV computes DEST:=S2-S1.

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

SUBC — Integer subtract with carry

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

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

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

MULT — Integer multiply

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

Purpose: DEST:=LOW_ORDER(S1*S2)

Restrictions: None

Exceptions: INT_OVFL

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

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

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

MULTL — Integer multiply long, long result

MULTL . {Q,H,S} TOP

Purpose: DEST:=S1*S2

Restrictions: Next

Exceptions: None

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

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

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

QUO — Integer quotient

QUO . {Q,H,S,D} **TOP**
QUOV . {Q,H,S,D} **TOP**

Purpose: QUO computes DEST:=S1/S2; QUOV computes DEST:=S2/S1. QUO (or QUOV) rounds its result toward zero.

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

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

The following illustrates a simple quotient calculation:

```
QUO.Q RTA,#[345],#3      ;RTA:=115 (QW)
```

Given a positive singleword NUM, this code stores in RTA the next-higher number with the same number of one-bits. This can be useful in combinatorial algorithms. For example, starting with 17_8 and repeatedly applying this algorithm until the result exceeds 10000_8, will produce bit masks indicating all possible ways of choosing four bits out of twelve:

```
NEG.S TEMP,NUM
AND.S RTA,NUM,TEMP       ;RTA gets just the lowest bit of NUM
ADD.S TEMP,RTA,NUM       ;TEMP gets NUM with the lowest string of "1"
                         ; bits cleared, and a new "1" bit above where
                         ; they were
XOR.S RTB,NUM,TEMP       ;RTB get just the differences between
                         ; TEMP and NUM, i.e. a copy of the lowest
                         ; string of "1" bits in NUM plus one more
                         ; "1" bit to the left
QUO.S RTB,RTA            ;recall that RTA has one bit set, and
                         ; so is a power of two; the effect is to
                         ; right-justify the string in RTB, which is
                         ; one bit longer than the lowest string of
                         ; "1" bits in NUM
SHF.RT.S RTB,#2          ;shift this two bits to the right; now the
                         ; string is one bit SHORTER
OR.S RTA,RTB,TEMP        ;merge RTB and TEMP to form the final result
```

QUOL — Integer quotient, long dividend

QUOL . {Q,H,S} TOP
QUOLV . {Q,H,S} TOP

Purpose: QUOL computes DEST:=S1/S2; QUOLV computes DEST:=S2/S1. QUOL (or QUOLV) rounds its result toward zero.

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

Precision: DEST has the same precision as the modifier. For QUOL, S2 has the precision of the modifier and S1 has twice the precision of the modifier. For QUOLV, S1 has the precision of the modifier and S2 has twice the precision of the modifier. The double precision operand must be aligned accordingly.

The following example takes a quotient with a long dividend:

```
QUOL.Q RTA,#[1221],#3    ;RTA:=333 (QW)
```

QUO2 — Integer quotient by power of 2

QUO2 . {Q,H,S,D} TOP
QUO2V . {Q,H,S,D} TOP

Purpose: QUO2 computes DEST:=S1/(2^{S2}); QUO2V computes DEST:=S2/(2^{S1});. QUO2 (or QUO2V) rounds its result toward zero. (Alternatively, the SHFA.RT instruction may be used to divide by a power of two, rounding toward negative infinity.)

The operand serving as the exponent may be negative, in which case a multiplication by a positive power of two is performed.

Restrictions: None

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

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

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

```
QUO2.S RTA,#-3,#1                ;RTA:=-1
```

QUO2L — Integer quotient by power of 2, long dividend

QUO2L . {Q,H,S} TOP
QUO2LV . {Q,H,S} TOP

Purpose: QUO2L computes DEST:=S1/(2^{S2}); QUO2LV computes DEST:=S2/(2^{S1}). QUO2L (or QUO2LV) rounds its result toward zero. The operand serving as the exponent may be negative, in which case a multiplication by a positive power of two is performed.

Restrictions: None

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

Precision: DEST has the same precision as the modifier. For QUO2L, S1 has twice the precision of the modifier and S2 has the precision of the modifier; for QUO2LV, S2 has twice the precision and S1 has the same precision as the modifier. The double precision operand must be aligned accordingly.

The following divides the long operand by 16 (decimal):

```
QUO2L.Q RTA,#1221,#4     ;RTA:=S1 (QW)
```

REM — Integer remainder

REM . {Q,H,S,D} **TOP**
REMV . {Q,H,S,D} **TOP**

Purpose: REM stores in DEST the remainder from S1 / S2. The result is the remainder produced by a division that rounds toward zero (as in the QUO instruction). The result (DEST) has the same sign as the dividend (S1), or is zero.

REMV, the reverse form, stores in DEST the remainder from S2 / S1.

Restrictions: None

Exceptions: INT_Z_DIV

Precision: S1, 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 (QW)
REM.Q RTA,#5,#-3       ;RTA:=2 (QW)
REM.Q RTA,#-5,#3       ;RTA:=-2 (QW)
REM.Q RTA,#-5,#-3      ;RTA:=-2 (QW)
```

REML... Integer remainder, long dividend

REML . {Q,H,S} **TOP**
REMLV . {Q,H,S} **TOP**

Purpose: REML stores in DEST the remainder from S1 / 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 (S1), or is zero.

REMLV, the reverse form, stores in DEST the remainder from S2 / S1.

Restrictions: None

Exceptions: INT_Z_DIV

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

For REMLV, S1 and DEST have the precision of the modifier; S2 has twice that precision and must be aligned accordingly.

The following illustrates the remainder using a long dividend:

```
REML.Q RTA,#[12345],#[300]        ;RTA:=245 (QW)
```

MOD — Integer modulus

MOD . {Q,H,S,D} TOP
MODV . {Q,H,S,D} TOP

Purpose: The MOD instruction produces the remainder from a division S1/S2 that rounds toward negative infinity (in contrast with the REM instruction, which produces the remainder from a division that rounds toward zero) and stores that remainder in DEST. That remainder has the same sign as the divisor, or is 0.

MODV, the reverse form, computes the remainder from S2/S1.

Note that the MOD function provided in many high-level languages such as Pascal actually corresponds to the assembly language REM instruction, not the MOD instruction.

Restrictions: None

Exceptions: INT_Z_DIV

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

The following examples illustrate the operation of MOD and REM for various combinations of signs. In each case, the instruction discards the quotient and places the remainder in RTA:

```
MOD.Q RTA,#5,#3       ; 5/3      = 1 remainder 2
REM.Q RTA,#5,#3       ; 5/3      = 1 remainder 2
MOD.Q RTA,#5,#-3      ; 5/(-3)   = -2 remainder -1
REM.Q RTA,#5,#-3      ; 5/(-3)   = -1 remainder 2
MOD.Q RTA,#-5,#3      ; (-5)/3   = -2 remainder 1
REM.Q RTA,#-5,#3      ; (-5)/3   = -1 remainder -2
MOD.Q RTA,#-5,#-3     ; (-5)/(-3) = 1 remainder -2
REM.Q RTA,#-5,#-3     ; (-5)/(-3) = 1 remainder -2
```

MODL — Integer modulus, long dividend

MODL . {Q,H,S} **TOP**
MODLV . {Q,H,S} **TOP**

Purpose: MODL computes the remainder from a division S1/S2 that rounds toward negative infinity rather than toward zero as the REML instruction does, and stores it in DEST. That remainder has the same sign as the divisor (S2), or is zero.

MODLV, the reverse form, computes the remainder from S2/S1. Note that the MOD function provided in many high-level languages such as Pascal actually performs the assembly language REM instruction, not the MOD instruction.

Restrictions: None

Exceptions: INT_Z_DIV

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

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

The following illustrates the modulo operation using a long dividend.

```
MODL.Q RTA,#12345,#300  ;RTA:=245 (QW)
```

DIV — Integer divide

DIV . {Q,H,S,D} **TOP**
DIVV . {Q,H,S,D} **TOP**

Purpose: DIV computes FIRST(DEST):=S1/S2 and SECOND(DEST):=S1 *rem* S2. DIV is like doing both a QUO instruction and a REM instruction.

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

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

Precision: S1, S2, FIRST(DEST), and SECOND(DEST) have the same precision as the modifier. FIRST(DEST) and SECOND(DEST) must align together to form a single entity with twice the precision of the multiplier.

The following produces a quotient-remainder result:

```
DIV.Q RTA,#[345],#3      ;RTA:=114001 (two QWs)
```

The following subroutine accepts a positive singleword in location X (which is destroyed) and prints it in a radix in the range 2..35 specified by RADIX, using the digits 0-9 and A-Z (A=10, B=11, etc.). The subroutine should be called by JSR X+4,PRINUM. Location X+4 (the singleword after X) is used, but its original contents are saved and restored. The subroutine prints a character by using TRPEXE.13, which is assumed to trap to an executive character print routine. The remainder method of generating digits produces them "backwards", and so a recursive call using JSR saves each digit on the stack as it is generated, and then the digits are printed as the stack is unwound.

```
PRINUM: DIV.S X,RADIX            ;X+4 gets next digit, X gets quotient
        SKP.EQL.S X,#0           ;skip if resulting quotient is zero
        JSR X+4,PRINUM           ;otherwise save that digit and do more
        CMPSF.LEQ.S RTA,X+4,#9.  ;digit now in X+4; is it ≤9?
        ADD.S X+4,<["0" ? "A"-10.]+4>[RTA]↑2 ;if so, use 0-9; if not, use A-Z
        TRPEXE.13 X+4            ;print character
        RETSR X+4,(SP)           ;return, restoring X+4 to previous value
```

DIVL — Integer divide, long dividend

DIVL . {Q,H,S} **TOP**
DIVLV . {Q,H,S} **TOP**

Purpose: DIVL computes FIRST(DEST):=S1/S2 and SECOND(DEST):=S1 *rem* S2. DIVL is like doing both a QUOL instruction and a REML instruction.

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

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

Precision: For DIVL, operands S2, FIRST(DEST), and SECOND(DEST) have the same precision as the modifier. S1 has a precision *twice* that of the modifier and must be aligned accordingly. FIRST(DEST) and SECOND(DEST) must align together to form a single entity having twice the precision of the modifier.

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

```
DIVL.Q RTA,#[12345],#[300]      ;RTA:=33245 (two UWs)
```

INC — Integer increment

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

Purpose: OP1:=OP2+1

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

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

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

DEC — Integer decrement

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

Purpose: OP1:=OP2-1

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

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

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

TRANS — Signed integer translate

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

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

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

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

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

Restrictions: None

Exceptions: INT_OVFL

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

The second instruction illustrates the sign-extension of TRANS:

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

NEG — Integer negate

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

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

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

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

The following negates the value in RTA:

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

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

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

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

ABS — Integer absolute value

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

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

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

Restrictions: None

Exceptions: CARRY, INT_OVFL

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

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

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

MIN — Integer minimum

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

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

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

Restrictions: None

Exceptions: None

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

The following sets RTA to 0 if RTA is positive:

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

MAX — Integer maximum

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

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

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

Restrictions: None

Exceptions: None

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

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

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

Suppose that A and B are two byte pointers. Then the following instruction puts in RTA the byte pointer which indicates the byte starting higher in memory than the other; or, if they start at the same bit, whichever points to the longer byte. (This is a consequence of the representation of byte pointers--see Section 2.10). Similarly, all D-precision integer comparison instructions--such as MIN.D, CMPSF.D, SKP.D, etc.--can be used to compare byte pointers in this fashion:

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

LMINMAX — Lengthwise integer minimum and maximum

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

Purpose: Select the minimum and maximum elements of a vector of signed integers whose first element is OP2. Put the minimum in FIRST(OP1) and the maximum in SECOND(OP1).

Restrictions: None

Exceptions: None

Precision: FIRST(OP1), SECOND(OP1), and each element of vector OP2 have the precision of the modifier. FIRST(OP1) and SECOND(OP1) must align to form an entity with twice the precision of the modifier.

The following sets RTA to -4 and RTA1 to 16:

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

2.2 Unsigned Integer Arithmetic

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

The architecture provides instructions specifically for unsigned multiplication and division. These instructions were designed to be used for arithmetic on numbers of arbitrarily great precision (as exemplified by "bignums" in Maclisp). The instructions for signed addition and subtraction work properly on unsigned data provided the program ignores the INT_OVFL side effect and uses the CARRY to signal overflow or to propagate bits from one word of a bignum to another.

UMULT — Unsigned integer multiply

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

Purpose: DEST:=LOW_ORDER(S1*S2)

Restrictions: None

Exceptions: INT_OVFL; UMULT sets INT_OVFL whenever MULT does. In addition, UMULT sets INT_OVFL whenever one operand has its high order bit set and the other operand exceeds 1.

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

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

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

UMULTL — Unsigned integer multiply, long result

UMULTL . {Q,H,S} TOP

Purpose: DEST:=S1*S2

Restrictions: None

Exceptions: None

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

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

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

UDIV — Unsigned integer divide

UDIV . {Q,H,S,D} **TOP**
UDIVV . {Q,H,S,D} **TOP**

Purpose: UDIV places the quotient of the unsigned integer division S1/S2 in FIRST(DEST) and the unsigned integer remainder S1 *rem* S2 in SECOND(DEST).

UDIVV produces the quotient and remainder from integer division S2/S1.

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

Precision: For UDIV, S1, S2, FIRST(DEST), and SECOND(DEST) all have the same precision as the modifier. FIRST(DEST) and SECOND(DEST) must align together to form an entity having twice that precision.

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

```
UDIV.Q RTA,#775,#22.     ;RTA:=027003 (two QWs)
```

UDIVL — Unsigned integer divide, long dividend

UDIVL . {Q,H,S} **TOP**
UDIVLV . {Q,H,S} **TOP**

Purpose: UDIVL places the result of the unsigned integer division S1/S2 in FIRST(DEST) and the unsigned integer remainder S1 *rem* S2 in SECOND(DEST).

Restrictions: None

Exceptions: INT_OVFL, INT_Z_DIV

Precision: For UDIVL, S2, FIRST(DEST), and SECOND(DEST) all have the same precision as the modifier. S1 has a precision *twice* that of the modifier and must align accordingly. FIRST(DEST) and SECOND(DEST) must align together to form a single entity with twice that precision.

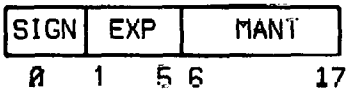
The following sets RTA to the unsigned quotient-remainder of 377377 (octal) divided by 777 (octal):

```
UDIVL.Q RTA,#377377,#777        UDIVL:=377776 (two QWs)
```

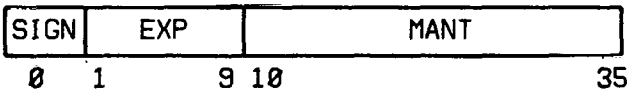
2.3 Floating Point Arithmetic

2.3.1 Floating Point Data Format

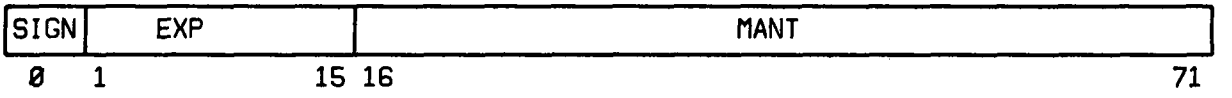
Floating point data can occur in three of the four standard precisions: halfword, singleword, or doubleword. The floating point representation is made up of three fields: SIGN, EXP, and MANT.



Halfword floating point format



Singleword floating point format



Doubleword floating point format

SIGN is 1 if the floating point number is negative.

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

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

<SIGN><hidden bit>.<MANT>

The complete mantissa consists of the concatenation of <hidden bit> and the MANT field. The entire mantissa is normalized to obey the following:

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

or

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Integrity of Floating Point Arithmetic

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

$$1.0*A=A$$
$$A+(-B)=A-B$$
$$A*B=B*A$$
$$A+B=B+A$$

2.3.3 Floating Point Exception Values

Besides zero, five floating point numbers have special meanings. The positive floating point number with the greatest magnitude (in a given precision) is called *OVF* (overflow). The two's-complement of OVF is called *MOVF* (minus overflow). The smallest positive floating point number is called *UNF* (underflow). The largest negative floating point number is called *MUNF* (minus underflow). The floating point number with the sign bit set to 1 and all other bits set to 0 is called *NAN* (not a number); all floating point instructions consider it illegal.

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

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

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

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

2.3.4 Comparing Floating Point Values

Another happy consequence of the floating point format is the ability to compare floating point numbers as if they were signed integers, without decoding the format. Thus, the architecture does not provide a separate set of test and branch instructions for floating point numbers. Instead, a single set serves for both signed integers and floating point numbers.

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

2.3.5 Floating Point Rounding Modes

During floating point operations, rounding of the result may be necessary. The FIX instruction includes a modifier that specifies how it rounds; all other floating point instructions which round their results do so according to the field *RND_MODE* in the *USER_STATUS* register. Instructions RRNDMD and WRNDMD (Section 2.3) read and write that field.

Let F be the magnitude of the difference between a true floating point result, R, and the greatest

representable floating point number N which is less than or equal to R, expressed as a fraction of the least-significant representable bit of R. The bits of RND_MODE have the following functions (reversals of rounding direction accumulate):

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

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

RND_MODE (octal)	Function	Modifier for FIX
0	Floor	FL
1	Diminished magnitude	DM
4	Ceiling	CL
5	Augmented magnitude	
12	Stable	ST
14	Half rounds toward positive infinity (PDP-10 FIXR)	HP
15	Approximate PDP-10 FLTR rounding	

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

2.3.6 Floating Point Exception Handling

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

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

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

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

FLT_REP Floating-point result cannot be represented exactly within the allowed mantissa (and must therefore be rounded). This bit signals a condition that may happen most of the time in ordinary floating point arithmetic.

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

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

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

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

FLT_OVFL_MODE<0:1>

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

FLT_UNFL_MODE<0:1>

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

FLT_NAN_MODE

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

2.3.7 Propagating Floating Point Exceptions

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

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

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

Unary operations

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

Addition (A+B)

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

Multiplication (A*B)

A ↓ B→	MOVF	-Y	MUNF	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

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

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

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

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

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

2.3.8 Floating Point Arithmetic

FADD — Floating point add

FADD . {H,S,D} **TOP**

Purpose: DEST:=S1+S2.

Restrictions: None.

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

FSUB **Floating point subtract**

FSUB . {H,S,D} **TOP**
FSUBV . {H,S,D} **TOP**

Purpose: FSUB calculates DEST:=S1-S2.

FSUBV, the reverse form, calculates S2-S1.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

FMULT — Floating point multiply

FMULT . {H,S,D} **TOP**

Purpose: DEST:=S1*S2.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

FMULTL — Floating point multiply, long result

FMULTL . {H,S} **TOP**

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

Restrictions: None

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

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

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

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

FDIV — Floating point divide

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

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

FDIVV swaps the roles of S1 and S2.

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

FDIVL — Floating point divide, long dividend

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

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

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

FRECIP — Floating point reciprocal

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

The following instruction reciprocates 2.0:

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

FSC — Floating point scale

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

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

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

Restrictions: None

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

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

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

```
FSC.S RTA,#1      ;RTA:=RTA*2↑(1)=2.0*RTA
```

FIX — Convert floating point to fixed (integer)

FIX . {FL,CL,DM,HP,ST,US} . {Q,H,S,D} . {H,S,D} **XOP**
VFIX . {H,S,D} . {H,S,D} **V:=V**

Purpose: FIX converts the floating point number specified by OP2 into an integer and stores it in OP1. The first modifier specifies which of the rounding modes (explained in Section 2.3.5) to use in the conversion:

FL	Floor
CL	Ceiling
DM	Diminished magnitude
HP	Half rounds toward positive infinity
ST	Stable
US	Whichever mode USER_STATUS.RND_MODE specifies

VFIX converts each element of the vector beginning with OP2 to an integer and stores the result in the corresponding element of the vector beginning with OP1. Instead of specifying rounding modes via a modifier, it always uses the rounding mode specified in USER_STATUS; the additional cost of executing a WRNDMD instruction to change the rounding mode is negligible for vectors of reasonable length.

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

Restrictions: None

Exceptions: INT_OVFL

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

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

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

FLOAT — Convert to floating point

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

Purpose: FLOAT converts the integer specified by OP2 into a floating point number and stores it in OP1.

VFLOAT converts each element of the vector beginning with OP2 to a floating point number and stores the result in the corresponding element of the vector beginning with OP1.

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

Restrictions: None

Exceptions: FLT_OVFL. (This can occur only in the cases of FLOAT.H.S and FLOAT.H.D. For all other conversions, the floating point format can express the corresponding integer with--at worst--only the loss of the least significant bits.)

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

The following loads RTA with the floating point value 1.0:

```
        FLOAT.S.S RTA,#1        ;RTA:=1.0
```

FTRANS — Floating point translate

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

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

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

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

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

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

Restrictions: None

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

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

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

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

FNEG — Floating point negate

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

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

```
FNEG.H RTA,#-1          ;RTA:=MUNF, signal FLT_UNFL
FNEG.H RTA,#677777      ;RTA:=OVF, signal FLT_OVFL
FNEG.H RTA,#100000      ;RTA:=NAN, signal FLT_NAN
```

FABS — Floating point absolute value

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

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

FMIN — Floating point minimum

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

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

FMAX — Floating point maximum

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

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

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

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN

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

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

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

RRNDMD, WRNDMD — Read/write rounding mode

RRNDMD **XOP**
WRNDMD **XOP**

Purpose: RRNDMD sets OP1 to USER_STATUS.RND_MODE. WRNDMD sets USER_STATUS.RND_MODE to OP1. In both instructions, OP2 is unused. For WRNDMD, if OP1 contains bits outside the field that specifies rounding modes, the result is undefined. See Section 2.3.5 for a description of rounding modes.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following jumps to ISFLOOR if floor rounding is specified by USER_STATUS. Otherwise, it selects ceiling rounding:

```
FLOOR=0
CEILING=4
RRNDMD RTA
SKP.EQL.S RTA,#FLOOR,ISFLOOR
WRNDMD #CEILING
```

2.4 Complex Arithmetic

Certain instructions operate on halfword or singleword complex numbers in either signed integer or floating point format. A complex number actually consists of two consecutive integers or floating point numbers; the one at the lower memory or register address is the real part and the one at the higher address is the imaginary part. Thus, a halfword complex number occupies two halfwords or one singleword (and must align as a singleword) while a singleword complex number occupies two singlewords.

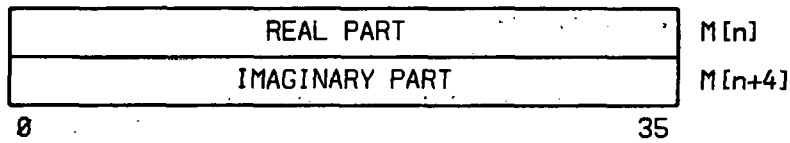


Figure 2-1
A singleword complex number

CMAG — Complex magnitude

CMAG . {H,S}	**XOP**
FCMAG . {H,S}	**XOP**
VCMAG . {H,S}	**V:=V**
VFCMAG . {H,S}	**V:=V**

Purpose: Compute the scalar magnitude of a complex number.

CMAG regards the complex number as a pair of signed integers, while FCMAG regards it as a pair of floating point numbers.:

```
OP1 := SquareRoot(FIRST(OP2)↑2 + SECOND(OP2)↑2)
```

VCMAG and VFCMAG are vector versions of CMAG and FCMAG. Assuming that "i" increments by the precision of the modifier, they compute:

```
FOR i := 0 TO SIZEREG-1 DO
    OP1[i] := SquareRoot(FIRST(OP2[2*i])↑2 +
        SECOND(OP2[2*i])↑2)
```

Restrictions: None

Exceptions: INT_OVFL (for CMAG and VCMAG); FLT_NAN, FLT_OVFL, and FLT_UNFL (for FCMAG and VFCMAG)

Precision: For CMAG and FCMAG, OP1, FIRST(OP2), and SECOND(OP2) have the precision specified by the modifier. FIRST(OP2) and SECOND(OP2) must align together to form an entity having twice that precision.

For VCMAG and VFCMAG, the elements of all three vectors have the precision specified by the modifier.

The following finds the length of the hypotenuse of a right triangle whose sides have lengths of 3 and 4:

```
CMAG.S  RTA, [3 ? 4]     ; RTA := 5
```

CADD — Complex add

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

Purpose: Add complex numbers, where FIRST(S1) and FIRST(S2) hold the real parts of the numbers and SECOND(S1) and SECOND(S2) hold the imaginary parts.

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

CADD deals with signed integers while FCADD deals with floating point numbers.

Restrictions: None

Exceptions: CARRY and INT_OVFL (for CADD); FLT_OVFL, FLT_UNFL and FLT_NAN (for FCADD)

Precision: FIRST(DEST), SECOND(DEST), FIRST(S1), SECOND(S1), FIRST(S2), and SECOND(S2) have the precision specified by the modifier. Each FIRST must align with the corresponding SECOND to form an entity with twice that precision.

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

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

CSUB — Complex subtract

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

Purpose: Subtract complex numbers, where FIRST(S1) and FIRST(S2) hold the real parts of the numbers and SECOND(S1) and SECOND(S2) hold the imaginary parts.

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

CSUB deals with signed integers while FCSUB deals with floating point numbers.

Restrictions: None

Exceptions: CARRY and INT_OVFL (for CSUB); FLT_OVFL, FLT_UNFL and FLT_NAN (for FCSUB)

Precision: FIRST(DEST), SECOND(DEST), FIRST(S1), SECOND(S1), FIRST(S2), and SECOND(S2) have the precision specified by the modifier. Each FIRST must align with the corresponding SECOND to form an entity with twice that precision.

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

```
CSUB.S  RTA,[4 ? 5],[3 ? 12.]   ; RTA := 1; RTA1 := -7
```

CMULT — Complex multiply

CMULT . {H,S} **TOP**
FCMULT . {H,S} **TOP**

Purpose: Multiply complex numbers, where FIRST(S1) and FIRST(S2) hold the real parts of the numbers and SECOND(S1) and SECOND(S2) hold the imaginary parts.

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

The instruction actually finishes the computation before altering DEST or NEXT(DEST), so operands may overlap without harm.

CMULT deals with signed integers while FCMULT deals with floating point numbers.

Restrictions: None

Exceptions: INT_OVFL (for CMULT); FLT_NAN, FLT_OVFL, and FLT_UNFL (for FCMULT)

Precision: FIRST(DEST), SECOND(DEST), FIRST(S1), SECOND(S1), FIRST(S2), and SECOND(S2) have the precision specified by the modifier. Each FIRST must align with the corresponding SECOND to form an entity having twice that precision.

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

```
CMULT.S RTA,[4 ? 5],[3 ? 12.]    ; RTA := -48; RTB := 63
```

2.5 Mathematics

SQRT — Square root

FSQRT . {H,S,D} **XOP**
VFSQRT . {H,S,D} **V:=V**

Purpose: Compute the principal square root in floating point: OP1:=SquareRoot(OP2).

VFSQRT performs FSQRT on each element of vector OP2 and places the results in vector OP1.

The implementation is guaranteed to be monotonic--that is, if x≥y then SQRT(x)≥SQRT(y). Attempting to take the square root of a negative number invokes FLT_NAN, which will result in either a FLT_NAN_TRAP hard trap or NAN, depending on the setting of USER_STATUS.

Restrictions: None

Exceptions: FLT_NAN

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

The following leaves the square root of 25 in RTA:

```
FSQRT.S RTA,#25.0       ; RTA := 5.0
```

FLOG — Floating point logarithm (base 2)

FLOG . {H,S,D} **XOP**
VFLOG . {H,S,D} **V:=V**

Purpose: FLOG computes the base 2 logarithm of OP2 and stores the result in OP1. The results are guaranteed to be monotonic--that is, if x≥y then FLOG(x)≥FLOG(y).

VFLOG performs FLOG on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_OVFL, FLT_UNFL, FLT_NAN. Taking the logarithm of a non-positive number invokes FLT_NAN, resulting in either NAN or a FLT_NAN_TRAP hard trap, depending on the setting of USER_STATUS.

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

The following leaves RTA set to the base 2 logarithm of 32:

```
FLOG.S  RTA,#32.0        ; RTA := 5.0
```

Using the rule that $\log_b z = \log_2 z / \log_2 b$, the following instructions compute the base 10 logarithm of 1000.0:

```
FLOG.S  RTB,#10.0        ; RTB := base 2 log of 10.0
FLOG.S  RTA,#1000.0      ; RTA := base 2 log of 1000.0
FDIV.S  RESULT,RTA,RTB   ; RESULT := 3.0
```

FEXP — Floating point exponential (base 2)

FEXP . {H,S,D} **XOP**
VFEXP . {H,S,D} **V:=V**

Purpose: Raise 2.0 to a power: FEXP computes OP1:=2.0↑OP2. VFEXP performs FEXP on each element of OP2 and places the result in the corresponding element of OP1. The results are guaranteed to be monotonic--that is, if x≥y then FEXP(x)≥FEXP(y).

Restrictions: None

Exceptions: FLT_NAN, FLT_OVFL, FLT_UNFL

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

Using the rule that $x \uparrow y = 2 \uparrow (y * \log_2 x)$, the following raises 81.0 to the power 0.25:

```
FLOG.S  RTA,#81.0
FMULT.S RTA,#0.25       ; RTA := 0.25 * FLOG(81.0)
FEXP.S  RTB,RTA         ; RTB := 3.0
```

FSIN — Floating point sine

FSIN . {H,S,D} **XOP**
VFSIN . {H,S,D} **V:=V**

Purpose: FSIN computes OP1:=Sine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

VFSIN performs FSIN on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_NAN

Precision: Both operands have the precision specified by the modifier.

The following computes the sine of an angle expressed in degrees:

```
MOV.S.S ANGLE,#30.0          ; 30 degrees
FDIV.S  RTA,ANGLE,#360.0     ; convert to cycles
FSIN.S  RTA                  ; RTA := 0.5
```

FCOS — Floating point cosine

FCOS . {H,S,D} **XOP**
VFCOS . {H,S,D} **V:=V**

Purpose: FCOS computes OP1:=Cosine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

VFCOS performs FCOS on each element of OP2 and places the result in the corresponding element of OP1.

Restrictions: None

Exceptions: FLT_NAN

Precision: Both operands have the precision specified by the modifier.

The following computes the cosine of an angle expressed in degrees:

```
MOV.S.S ANGLE,#60.0              ; 60 degrees
FDIV.S  RTA,ANGLE,#360.0         ; convert to cycles
FCOS.S  RTA                      ; RTA := 0.5
```

FSINCOS — Floating point sine and cosine

FSINCOS . {H,S,D} **XOP**

Purpose: Computes FIRST(OP1):=Cosine(OP2) and SECOND(OP1):=Sine(OP2). OP2 specifies the angle in cycles--that is, a "1.0" corresponds to 360 degrees or 2*PI radians.

Note that because the cosine appears in the first anyword of the pair and the sine in the second, the result can be used as a complex number.

Restrictions: None

Exceptions: FLT NAN

Precision: FIRST(OP1), SECOND(OP1), and OP2 have the precision specified by the modifier. FIRST(OP1) and SECOND(OP1) must align together to form an entity having twice that precision.

The following computes both the sine and the cosine of an angle expressed in degrees;

```
MOV.S.S ANGLE,#60.0              ; 60 degrees
FDIV.S  RTA,ANGLE,#360.0         ; convert to cycles
FSINCOS.S RTA                    ; RTA := 0.866...; RTA1 := 0.5
```

FATAN — Floating point arctangent

FATAN . {H,S,D} **TOP**
FATANV . {H,S,D} **TOP**
VFATANV . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: FATAN computes DEST:=Arctangent(S1/S2). Expressing the tangent as a quotient instead of a single value allows the instruction to determine the correct quadrant for the result, which is expressed in cycles--that is, "1.0" corresponds to 360 degrees or 2*PI radians.

FATANV, the reverse form, swaps the roles of S1 and S2.

VFATAN performs FATAN on each pair of elements, one from vector OP1 and the other from vector OP2, and places the result in the corresponding element of either vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=Arctangent(OP1[i]/OP2[i])
    ELSE
        SR0@[i]:=Arctangent(OP1[i]/OP2[i])
```

Restrictions: None

Exceptions: FLT_NAN

Precision: All three operands have the precision specified by the {H,S,D} modifier.

The following computes an arctangent in degrees:

```
FATAN.S RTA,#1.0,#1.0
FMULT.S RTA,#360.0      ; RTA := 45.0 degress
```

V2DSQ, V3DSQ — Vector 2- or 3-dimensional distance squared

V2DSQ . {SR,OP1} . {H,S,D} **V:=VV**
VF2DSQ . {SR,OP1} . {H,S,D} **V:=VV**
V3DSQ . {SR,OP1} . {H,S,D} **V:=VVV**
VF3DSQ . {SR,OP1} . {H,S,D} **V:=VVV**

Purpose: Compute the sum of squares of a series of coordinate pairs or triples.

V2DSQ and VF2DSQ deal with coordinate pairs, where the vector beginning with OP1 holds the first coordinate of each pair and the vector beginning with OP2 holds the second. Depending on the first modifier, these instructions put the result back in vector OP1 or in the vector pointed to by SR0. V2DSQ deals with integers, while VF2DSQ deals with floating point numbers:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP} THEN OP1[i]:=OP1[i]↑2 + OP2[i]↑2
    ELSE SR0@[i]:=OP1[i]↑2 + OP2[i]↑2
```

V3DSQ and VF3DSQ deal with coordinate triples, where the vector beginning with OP1 holds the first coordinate of each triple, the vector beginning with OP2 holds the second, and the vector pointed to by SR0 holds the third. Depending on the first modifier, these instructions put the result back in vector OP1 or in the vector pointed to by SR1. V3DSQ deals with integers, while VF3DSQ deals with floating point numbers:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN OP1[i]:=OP1[i]↑2 + OP2[i]↑2 + SR0@[i]↑2
    ELSE SR1@[i]:=OP1[i]↑2 + OP2[i]↑2 + SR0@[i]↑2
```

Restrictions: None

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

Precision: Each element of each vector has the precision specified by the second modifier.

The following example illustrates the use of V2DSQ:

```
MOV.S.S SIZEREG,#3      ; Specify length of vectors
MOVP.P.A SR0,RESULT     ; Set up SR0 to point to result
V2DSQ.SR.S [1 ? 2 ? 3],[4 ? 5 ? 6]
                        ; RESULT now holds [17. ? 29. ? 45.]
```

V2DIS, V3DIS — Vector 2- or 3-dimensional distance

V2DIS . {SR,OP1} . {H,S,D} V:=VV
VF2DIS . {SR,OP1} . {H,S,D} V:=VV
V3DIS . {SR,OP1} . {H,S,D} V:=VVV
VF3DIS . {SR,OP1} . {H,S,D} V:=VVV

Purpose: Compute the square root of the sum of squares for a series of coordinate pairs or triples.

V2DIS and VF2DIS operate on coordinate pairs, where the vector beginning with OP1 contains the first coordinate of each pair and the vector beginning with OP2 contains the second. Depending on the first modifier, the resulting vector goes back into OP1 or into the vector pointed to by SR0. V2DIS deals with integers while VF2DIS deals with floating point numbers:

```
FOR i:=1 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=SquareRoot(OP1[i]↑2 + OP2[i]↑2)
    ELSE
        SR0@[i]:=SquareRoot(OP1[i]↑2 + OP2[i]↑2)
```

V3DIS and VF3DIS operate on triples, with the vector beginning at OP1 containing the first coordinate of each triple, the vector beginning at OP2 containing the second, and the vector pointed to by SR0 containing the third. Depending on the first modifier, the result goes back into the vector starting at OP1 or into the vector pointed to by SR1:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier is OP1} THEN
        OP1[i]:=SquareRoot(OP1[i]↑2 + OP2[i]↑2 + SR0@[i]↑2)
    ELSE
        SR1@[i]:=SquareRoot(OP1[i]↑2 + OP2[i]↑2 + SR0@[i]↑2)
```

Restrictions: None

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

Precision: Each element of each vector has the precision specified by the second modifier.

Suppose X_DISP and Y_DISP represent a drawing as a series of line segments, describing each segment as a pair of displacements in the X and Y directions from the endpoint of the preceding segment. The following program fragment converts this data to represent each segment as an angle and magnitude:

```
; Obtain a vector of angles
```

```
        MOV.S.S SIZEREG,#0
        MOVP.P.A RTA,X_DISP
        MOVP.P.A RTB,Y_DISP
NEXT:   FATAN.S ANGLE[SIZEREG]↑2,RTA[SIZEREG]↑2,RTB[SIZEREG]↑2
        ISKP.LSS SIZEREG,LENGTH,NEXT
                                        ; Now SIZEREG = length of vector
        VF2DSQ.OP1.S X_DISP,Y_DISP      ; X_DISP becomes a vector
                                        ; of magnitudes
```

VDOT — Dot product

VDOT . {H,S,D} **S:=VV**
VFDOT . {H,S,D} **S:=VV**

Purpose: Compute the dot product of two vectors:

```
RTA:=0;
FOR i:=0 TO SIZEREG-1 DO
      RTA:=RTA + OP1[i] * OP2[i]
```

To avoid overflow and underflow problems, the processor accumulates the sum with as much precision as it can, regardless of the {H,S,D} modifier. If that modifier is "H", the result goes into RTA as a singleword, and if the modifier is "S", RTA is a doubleword. If the modifier is "D", however, the result is still a doubleword.

Restrictions: None

Exceptions: INT_OVFL (for VDOT); FLT_OVFL, FLT_UNFL, and FLT_NAN (for VFDOT).

Precision: The elements of each vector have the precision specified by the modifier. RTA has twice that precision unless the modifier is D, in which case RTA is a doubleword.

Suppose that singleword vector V contains the results from sampling a voltage waveform at 100 Hz for one second. The following computes the RMS voltage:

```
MOV.S.S SIZEREG,#100.    ; Put length in SIZEREG
VFDOT.S V,V              ; Sum of squares
FDIV.D  RTA,#100.0       ; Mean
FSQRT.D RTA,RTA          ; Root
```

CONV — Convolution

CONV . {H,S,D} **V:=VV**
FCONV . {H,S,D} **V:=VV**

Purpose: Compute the convolution of two vectors. OP1 and OP2 are the initial elements of the vectors, SIZEREG defines the length of vector OP1, and SR1 defines the range of integration (and therefore the length of vector OP2). The result appears in the vector pointed to by SR0:

```
FOR i:=0 TO SIZEREG-1 DO
    BEGIN
    SR0@[i]:=0;
    FOR j:=0 TO SR1-1 DO
        SR0@[i]:=SR0@[i] + OP2[j] * OP1[SR1-1 + i - j]
    END
```

Restrictions: None

Exceptions: INT_OVFL (for CONV); FLT_OVFL, FLT_NAN (for FCONV)

Precision: SR1 and the elements of each of the vectors have the precision specified by the second modifier.

Convolve A with B and store the result in C:

```
MOVP.P.A R0,C          ; SR0 points to destination
MOV.S.S SIZEREG,#100.  ; A is 100 elements long
MOV.S.S R1,#10.        ; B is 10 elements long
CONV.S A,B
```

RFLT2 — Second order recursive filter

RFLT2 . {H,S,D} V:=V
FRFLT2 . {H,S,D} V:=V

Purpose: Apply a second order recursive filter to the vector whose first element is OP2 and leave the results in the vector whose first element is OP1. The instruction obtains the coefficients of the filter from the five element vector pointed to by SR0. The result is actually two elements shorter than SIZEREG indicates, since it begins at OP1[2] instead of OP1[0]. The user must initialize the first two elements of the OP1 vector to start the recursion properly.

```
FOR i:=0 TO SIZEREG - 3 DO
    OP1[i+2]:=SR0@[0] * OP1[i]
        + SR0@[1] * OP1[i+1]
        + SR0@[2] * OP2[i]
        + SR0@[3] * OP2[i+1]
        + SR0@[4] * OP2[i+2]
```

Restrictions: None

Exceptions: INT_OVFL (for RFLT2); FLT_OVFL, FLT_UNFL, and FLT_NAN (for FRFLT2)

Precision: The coefficients and the elements of each vector have the precision specified by the modifier.

The following example filters the signal in vector SENSE_IN:

```
MOVP.P.A SR0,COEFFICIENTS       ; Pointer to five coefficients
MOV.S.S RESULT,[1.73476 ? 1.73476]
                                ; Initialize the recursion
MOV.S.S SIZEREG,#1000.          ; Specify length of SENSE_IN
FRFLT2.S RESULT,SENSE_IN
```

INTRAN In-place square matrix transpose

INTRAN . {H,S,D} **V:=V**

Purpose: Transpose a square two-dimensional matrix without moving the matrix to a different area of memory. (The TRANSP instruction can operate on a matrix which is not square, but must move the matrix to a new, non-overlapping area of memory as it does so.)

OP1 is the first element of the matrix, which must be stored in row major order (second subscript varying more rapidly than the first). OP2 gives the number of rows (which is, of course, the same as the number of columns) in the matrix, and must be a multiple of 8 for halfword precision (or a multiple of 4 for singlewords, or a multiple of 2 for doublewords).

Restrictions: None

Exceptions: None

Precision: Every element of the matrix has the precision specified by the modifier. OP2 is a singleword.

To transpose the following matrix:

```
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

one could use the INTRAN instruction like this:

```
DSPACE
                ; Expressions separated by "?" assemble
                ; successive singlewords in memory
FOURBY:         0 ? 1 ? 2 ? 3 ? 4 ? 5 ? 6 ? 7 ? 8
                9 ? 10 ? 11 ? 12 ? 13 ? 14 ? 15

ISPACE
        INTRAN.S FOURBY,#4.
                ; Now FOURBY = 0 ? 4 ? 8 ? 12 ? 1 ? 5 ? 9 ? 13
                ;              ? 2 ? 6 ? 10 ? 14 ? 3 ? 7 ? 11 ? 15
```

TRANSP — Matrix transpose

TRANSP . {H,S,D} **V:=V**

Purpose: Transpose a two-dimensional matrix, moving it to a different, non-overlapping area of memory in the process. (The INTRAN instruction transposes a matrix without moving it, but requires that the matrix be square.)

The instruction expects the matrix to be stored in row major order with its first element at OP2. The result of the transposition appears in row major order with its first element at OP1.

Registers R0 and R1 respectively specify the number of rows and columns in the source matrix. Registers R2 and R3 specify the number of columns to ignore between each row in the source and destination matrices respectively. To transpose an entire matrix, one sets R2 and R3 to zero; to transpose a submatrix, one sets R2 and R3 to skip over the columns that lie outside the submatrix.

The number of rows (and the number of columns) in the source matrix must be a multiple of 8 for halfword precision (or 4 for singlewords, or 2 for doublewords.)

Restrictions: None

Exceptions: None

Precision: All elements of the source and destination matrices have the precision specified by the modifier. R0, R1, R2, and R3 are singlewords.

To transpose the following matrix:

```
0  1  2  3
4  5  6  7
```

use the TRANSP instruction like this:

```
; Assume the matrix is stored as a series of doublewords
; in the following order: 0 1 2 3 4 5 6 7
ISPACE
        MOV.S.S %R0,#2          ; Number of rows
        MOV.S.S %R1,#4          ; Number of columns
        MOV.S.S %R2,#0          ; Do not skip anything
        MOV.S.S %R3,#0
        TRANSP.S NEWPLACE,TWOBY4
; The result is a series of doublewords in the following
; order: 0 4 1 5 2 6 3 7
```

As an example of how to use R2 and R3 to transpose a submatrix, suppose we have the following matrices (in Pascal notation):

```
VAR A: ARRAY [0..ARows, 0..ACols-1] OF INTEGER;
    B: ARRAY [0..BRows, 0..BCols-1] OF INTEGER;
```

and we want to transpose the submatrix of A whose origin is A[Ax,Ay] and whose size is SRows by SCols, storing the result in the submatrix of B whose origin is B[Bx,By]. Assuming the submatrices are proper (that is, they truly fit within A and B) we can use the following instructions:

```
MOV.S.S %R0,SRows       ; Number of rows in submatrix
MOV.S.S %R1,SCols       ; Number of columns in submatrix
MOV.S.S %R2,ACols
SUB.S %R2,SCols         ; Skip (ACols-SCols) columns between source rows
MOV.S.S %R3,BCols
SUB.S %R3,SCols         ; Skip (BCols-SCols) columns between dest rows

MOVP.P.A RTA,A[Ay]↑2
ARRIND.RTA 4*ACols,Ax   ; RTA:=ADDRESS(A[Ax,Ay])

MOVP.P.A RTB,B[By]↑2
ARRIND.RTB 4*BCols,Bx   ; RTB:=ADDRESS(B[Bx,By])
TRANSP.S (RTB),(RTA)
```

MATMUL — Matrix multiply

MATMUL . {H,S,D} **V:=VV**
FMATMUL . {H,S,D} **V:=VV**

Purpose: Multiply two 2-dimensional matrices stored in memory in row major order. OP1 is the first singleword of a 9-singleword vector which describes the two source matrices and the destination matrix.

Word	Meaning
0	Number of rows in source matrix 1
1	Number of columns in source matrix 1
2	Number of columns in source matrix 2
3	Number of columns to skip between rows of source matrix 1
4	Number of columns to skip between rows of source matrix 2
5	Number of columns to skip between rows of destination matrix
6	Pointer to origin of source matrix 1
7	Pointer to origin of source matrix 2
8	Pointer to origin of destination matrix

OP1[3]↑2, OP1[4]↑2, and OP1[5]↑2 are used when multiplying submatrices. To multiply entire matrices, one ordinarily sets these to zero.

Like VFDOT, FMATMUL and MATMUL accumulate results internally in the greatest feasible precision regardless of the precision of the result.

Restrictions: None

Exceptions: INT_OVFL (for MATMUL); FLT_NAN, FLT_OVFL, and FLT_UNFL (for FMATMUL)

Precision: Every element of each matrix has the precision specified by the modifier. OP1 is the first element of a vector of 9 singlewords.

The following example multiplies the two matrices shown and stores the result in matrix D:

```
A= 1  2  3      B= 1  2
   3  2  1         3  3
                   2  1
```

```
MOV.S.S %R0,#2          ; Rows in source matrix 1
MOV.S.S %R1,#3          ; Columns in source matrix 1
MOV.S.S %R2,#2          ; Columns in source matrix 2
MOV.S.S %R3,#0
```

```
MOV.S.S %R4,#0
MOV.S.S %R5,#0
MOVP.P.A %R6,A          ; Pointer to source matrix 1
MOVP.P.A %R7,B          ; Pointer to source matrix 2
MOVP.P.A %R8,D          ; Pointer to destination matrix
MATMUL.S %R0
```

As an example of how to multiply submatrices, assume we have the following matrices (in Pascal notation):

```
VAR A: ARRAY [0..ARows-1, 0..ACols-1] OF REAL;
    B: ARRAY [0..BRows-1, 0..BCols-1] OF REAL;
    D: ARRAY [0..DRows-1, 0..DCols-1] OF REAL;
```

and that we want to multiply the submatrix whose origin is at A[Ax,Ay] with the submatrix whose origin is at B[Bx,By], storing the result in D[Dx,Dy]. The submatrix of A has R rows by S columns and the submatrix of B has S rows by T columns. Assuming further that the submatrices are proper (that is, they fit inside the corresponding matrices), we can use the following code:

```
MOV.S.S DESC,R          ; Number of rows in source matrix 1
MOV.S.S DESC+4*1,S      ; Number of columns in source matrix 1
MOV.S.S DESC+4*2,T      ; Number of columns in source matrix 2
MOV.S.S DESC+4*3,ACols
SUB.S DESC+4*3,S        ; Skip (ACols-S) columns between rows in matrix 1
MOV.S.S DESC+4*4,BCols
SUB.S DESC+4*4,T        ; Skip (BCols-T) columns between rows in matrix 2
MOV.S.S DESC+4*5,DCols
SUB.S DESC+4*5,T        ; Skip (DCols-T) columns between rows in
                        ; destination

MOVP.P.A RTA,A[Ay]↑2
ARRIND.RTA ACols,Ax
MOVP.P.P DESC+4*6,RTA   ; Pointer to A[Ax,Ay]

MOVP.P.A RTA,B[By]↑2
ARRIND.RTA BCols,Bx
MOVP.P.P DESC+4*7,RTA   ; Pointer to B[Bx,By]

MOVP.P.A RTA,D[Dy]↑2
ARRIND.RTA DCols,Dx
MOVP.P.P DESC+4*8,RTA   ; Pointer to D[Dx,Dy]
FMATMUL.S DESC
```

FFT **In-place complex FFT and inverse FFT**

CFFT . {H,S}	**V:=V**
FCFFT . {H,S}	**V:=V**
CFFTV . {H,S}	**V:=V**
FCFFTV . {H,S}	**V:=V**

Purpose: Compute the fast Fourier transform (FFT) or inverse fast Fourier transform of a vector of complex numbers.

CFFT and FCFFT compute the FFT, with CFFT operating on complex signed integers and FCFFT on complex floating point numbers.

CFFTV and FCFFTV compute the inverse FFT, with CFFTV operating on complex signed integers and FCFFTV on complex floating point numbers.

For all four instructions, OP1 designates the first element of the vector to be transformed. In each case, the instruction puts its results back into the original source vector. The number of elements in the vector must be a power of 2; OP2 contains that power (i. e., the base 2 logarithm of the number of elements). If OP2 is not positive, the instruction leaves the vector untouched.

If the source vector exceeds the maximum allowable length, an FFT_TOO_LONG soft trap occurs. (This limit is implementation-dependent; see Section 1.12.) If desired, one can provide a software trap handler that operates transparently to the user on vectors of arbitrary size, transforming a lengthy vector by repeatedly applying the instruction to subvectors.

The last step of the FFT algorithm is a "scrambling" operation which swaps elements of the vector whose indices within the vector are bit reversals of each other. (For example, in a 16-element vector where indices range from 0 to 15, this scrambling would swap element 12 with element 3 because reversing the bits of the four-bit binary representation of 12 yields 3. Similarly, the scrambling would swap element 1 with element 8, and so on.) Because this step represents a considerable fraction of the time required for the total FFT, the architecture does not incorporate it in the FFT instructions themselves, but provides a separate instruction called BADREV to perform it.

Similarly, "scrambling" is the first step of the complete inverse FFT algorithm, but it is omitted from the inverse FFT instructions, which expect their source arrays to be scrambled.

Thus, a complete FFT would require the CFFT instruction (for example) followed by the BADREV instruction. A complete inverse FFT would require the BADREV instruction followed by (for example) the CFFTV instruction.

Providing a separate instruction for swapping elements saves time in many applications where one wants to transform a signal, operate on it, and transform it back. Because the FFT instructions produce a scrambled result and the inverse FFT instructions expect a scrambled input, one can

simply omit to unscramble and rescramble between them--provided the operations that take place between the FFT and inverse FFT instructions preserve the scrambled order.

Restrictions: None

Exceptions: INT_OVFL, (for CFFT and CFFTV); FLT_OVFL, FLT_UNFL, and FLT_NAN (for FCFFT and FCFFTV)

Precision: Every element of the vector has the precision specified by the modifier. OP2 is a singleword.

Consider a simple filtering operation where one transforms the input signal, multiplies it by a vector of selected filter coefficients, and transforms it back. One could write:

```
MOVP.P.A RTA,COEFFIC     ; Point to filter coefficients
CFFT.S INPUT,LOGSIZE     ; FFT
BADREV.D INPUT
V"SX".S OUTPUT,INPUT     ; Filter signal using coefficients
BADREV.D OUTPUT,LOGSIZE
CFFTV.S OUTPUT,LOGSIZE   ; Inverse FFT
```

But by scrambling the coefficient vector itself (an operation which need be performed only once no matter how many signals are to be passed through the same filter),

```
BADREV.D COEFFIC
```

one can remove both BADREV operations from the preceding sequence:

```
MOVP.P.A RTA,COEFFIC     ; Point to scrambled coefficients
CFFT.S INPUT,LOGSIZE     ; FFT
V"SX".S OUTPUT,INPUT     ; Filter using scrambled coefficients
CFFTV.S OUTPUT,LOGSIZE   ; Inverse FFT
```

The following example uses the FCFFT, BADREV, and INTRAN instructions together to perform a two-dimensional FFT:

```
;2DFFT - Two dimensional complex FFT
; half-word floating-point
;Transform complex 2D array whose origin is in ORG
;Size of array is 2↑LOGSIZE by 2↑LOGSIZE
;
; Called via JSR PC,2DFFT
;
2DFFT:  SHF.LF.S RTA,#1,LOGSIZE ;Get number of rows (and columns)
        MOV.S.S ESIZE,RTA       ;Save number of elements in rows and columns
```

```
        SHF.LF.S SIZE,RTA,#2    ;Convert to half-word complex size and save
        MOVP.P.P T,ORG          ;Initialize row pointer to first row
        MOV.S.S RTA,ESIZE       ;Loop counter
2dfft1: FCFFT.H (T),LOGSIZE     ;Transform a row
        BADREV.S (T),LOGSIZE    ;Un-bit-reverse this row
        ADD.S T,SIZE            ;Step to next row
        DJMPZ.GTR RTA,2dfft1    ;Last row?
        INTRAN.S (ORG),ESIZE    ;Transpose array
        MOVP.P.P T,ORG
        MOV.S.S RTA,ESIZE
2dfft2: FCFFT.H (T),LOGSIZE     ;Transform a column
        BADREV.S (T),LOGSIZE    ;Un-bit-reverse this column
        ADD.S T,SIZE            ;Step to next column
        DJMPZ.GTR RTA,2dfft2    ;Last column?
        INTRAN.S (ORG),ESIZE    ;Transpose array back
        RETSR PC,(SP)           ;Return
```

BADREV — In-place bit address reversal

BADREV . {H,S,D} **V:=V**

Purpose: Within a vector, swap each pair of elements whose addresses represent bit-reversals of each other. The instruction is primarily useful in conjunction with the FFT and inverse FFT instructions.

The last step of the FFT algorithm is a "scrambling" operation which swaps elements of the vector whose indices within the vector are bit reversals of each other. (For example, in a 16-element vector where indices range from 0 to 15, this scrambling would swap element 12 with element 3 because reversing the bits of the four-bit binary representation of 12 yields 3. Similarly, the scrambling would swap element 1 with element 8, and so on.)

OP1 is the first element of the vector to be scrambled; the instruction puts the results back into the same vector. The number of elements in the vector must be a power of 2. OP2 specifies that power (or, in other words, the base 2 logarithm of the number of elements). If OP2 is not positive, the instruction leaves the vector untouched.

Restrictions: None

Exceptions: None

Precision: The elements of the vector all have the precision specified by the modifier. OP2 is a singleword.

Note that when one uses BADREV to complete an FFT operation, the precision must be twice that of the FFT instruction because the vector in question contains complex numbers and thus each data point comprises two values:

```
CFFT.S SIGNAL          ; Fourier transform leaves the vector
                       ; scrambled
BADREV.D SIGNAL        ; Undo the scrambling
```

QPART — Quicksort partition inner loop

QPART V:=V

Purpose: Pipelined processors must predict with considerable accuracy whether conditional branch instructions will alter the flow of control, or execution speed suffers. Because sorting algorithms usually contain unpredictable conditional branches, the architecture provides an instruction to perform the inner loop of the Quicksort algorithm, eliminating branches.

OP1 is a pointer to the first element of a vector of records and OP2 is a pointer to the last record in the vector. Each record consists of a singleword key followed by a singleword of data (typically a pointer to a larger amount of data).

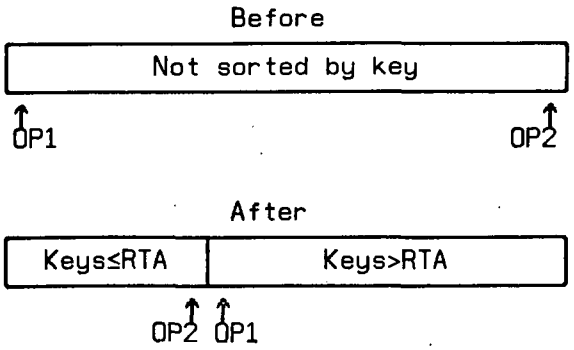
RTA contains a partitioning value.

The instruction rearranges the elements of the vector, segregating them into two groups so that all the records in one group have keys exceeding RTA and all the records in the other have keys less than or equal to RTA. Within each group, the records may still be disordered (though in moving records about to achieve the segregation, the instruction does attempt to order them locally); the instruction guarantees merely to partition the vector into two groups relative to the value in RTA.

When the instruction finishes, the first part of the vector contains the group of records with keys less than or equal to RTA, and OP2 points to the last record in that group. OP1 points to the next record, which is the first record in the group whose keys exceed RTA. RTA contains a code that reports the status of the two partitions:

0	The lower partition is sorted, but the upper one is not.
1	The upper partition is sorted, but the lower one is not.
2	Both partitions need sorting. The upper has fewer records.
3	Both partitions need sorting. The lower has fewer records.
4	Both partitions are sorted.

In simplified form, the instruction does the following:



Restrictions: None

Exceptions: None

Precision: Each element of the vector is a pair of singlewords, the first serving as a key and the second as data which the instruction moves along with the key. RTA is a singleword.

The following example illustrates how to use QPART to implement the complete quicksort algorithm:

```
; Quicksort
; Called via: JSR #-1,QUICKSORT
;  On entry :
;   LOW - pointer to first record of array to be sorted
;   HIGH - pointer to last record of array to be sorted
;    (HIGH must immediately follow LOW)
;  On exit :
;   Array between LOW and HIGH is completely sorted
QUICKSORT:
        ADJSP.UP SP,#10                 ;Reserve space to save HIGH and LOW
                                        ; later on
QUICK1: SUB.S RTB,HIGH,LOW              ;Calculate size of array - 8
        SHFA.RT.S RTB,#4                ;Get half the size
                                        ; (in double-words)
        SEXCH.D (LOW),(HIGH)            ;Swap the first, last, and middle
        SEXCH.D (LOW)0[RTB]↑3,(HIGH)    ; words of the array as necessary
        SEXCH.D (LOW),(LOW)0[RTB]↑3     ;  so first<middle<low
        MOV.S.S RTA,(LOW)0[RTB]↑3       ;Partition array around middle's value
        MOV.D.D (SP)-10,LOW             ;Save high and low pointers
        QPART LOW,HIGH                  ;Do the partitioning
        JMPA QUICK2[RTA]↑3              ;Dispatch to correct routine

QUICK2:         ;Dispatch table.
                ;It is important that all sections (except the last)
                ; be two words long

;Sort upper half only => tail recursion
        MOV.S.S HIGH,(SP)-4
        JMPA QUICK1

;Sort lower half only => tail recursion
        MOV.S.S LOW,(SP)-10
        JMPA QUICK1

;Sort upper first then lower => full recursion
```

```
        EXCH.S HIGH,(SP)-4
        JMPA QUICKSORT

;Sort lower then upper => full recursion
        EXCH.S LOW,(SP)-10
        JMPA QUICKSORT

;All sorted
        MOVP.P.A SP,(SP)-10     ;Discard the HIGH and LOW just saved
        MOV.D.D LOW,(SP)-10     ;Restore previous HIGH and LOW
        ;If LOW is the -1 value pushed by the JSR that invoked the quicksort,
        ; we're finished, so return to the caller. Otherwise, tail recursion
        ; continues sorting.
        JMPZ.GEQ.S LOW,QUICK1   ;Tail recursion
        RET (SP)-4,(SP)         ;Discard -1 and return to original caller
```

2.6 Chained Vectors

These instructions perform arithmetic on vectors, often combining two or more operations. This results in faster execution not only because it reduces the number of instructions the processor must fetch--a single multiply-and-add instruction can take the place of a multiplication followed by an addition, for example--but also because the processor can use its adder and multiplier in parallel.

Because the mnemonics for chained vector instructions explain themselves, and because the arithmetic operations are logical extensions of those for scalars, this section will not describe each instruction in detail.

Each mnemonic consists of a V followed by up to two letters defining the data type and then an equation within quotation marks:

```
V<data type>"<equation>"
```

For <data type>, a "CF" indicates complex floating point, a "C" alone indicates complex signed integer, and "F" alone indicates floating point. If <data type> is missing, the instruction deals with signed integers.

Within the equation, "X", "Y", and "Z" are the first, second, and third source vectors while "S" and "R" are the first and second source scalars. As in algebra, concatenating variables indicates multiplication.

Thus, for example, the instruction:

```
VF"X+SY",OP1
```

performs the operation:

```
FOR i:=0 TO SIZEREG-1 DO
        OP1[i]:=OP1[i] + RTA * OP2[i]
```

Two Vector Operands and One Scalar

S+X, S-X, SX

V"S+X" . {H,S,D} **V:=VS**
VF"S+X" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA + OP2[i]

V"S-X" . {H,S,D} **V:=VS**
VF"S-X" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA - OP2[i]

V"SX" . {H,S,D} **V:=VS**
VF"SX" . {H,S,D} **V:=VS**
FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA * OP2[i]

Three Vector Operands

X+Y, X-Y, Y-X, XY

V"X+Y" . {SR,OP1} . {H,S,D} **V:=VV**
VF"X+Y" . {SR,OP1} . {H,S,D} **V:=VV**

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] + OP2[i]
    ELSE SR0@[i]:=OP1[i] + OP2[i]
```

V"X-Y" . {SR,OP1} . {H,S,D} **V:=VV**
VF"X-Y" . {SR,OP1} . {H,S,D} **V:=VV**

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] - OP2[i]
    ELSE SR0@[i]:=OP1[i] - OP2[i]
```

V"Y-X" . {SR,OP1} . {H,S,D} **V:=VV**
VF"Y-X" . {SR,OP1} . {H,S,D} **V:=VV**

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP2[i] - OP1[i]
    ELSE SR0@[i]:=OP2[i] - OP1[i]
```

V"XY" . {SR,OP1} . {H,S,D} **V:=VV**
VF"XY" . {SR,OP1} . {H,S,D} **V:=VV**
VC"XY" . {SR,OP1} . {H,S} **V:=VV**
VFC"XY" . {SR,OP1} . {H,S} **V:=VV**

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] * OP2[i]
    ELSE SR0@[i]:=OP1[i] * OP2[i]
```

Three Vector Operands and One Scalar

X+SY, SX+Y, SY-X, SX-Y, SX+SY, SX-SY, S+XY, S-XY

V"X+SY" . {SR,OP1} . {H,S,D} V:=VVS
VF"X+SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] + RTA * OP2[i]
    ELSE SR0@[i]:=OP1[i] + RTA * OP2[i]
```

V"SX+Y" . {SR,OP1} . {H,S,D} V:=VVS
VF"SX+Y" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA * OP1[i] + OP2[i]
    ELSE SR0@[i]:=RTA * OP1[i] + OP2[i]
```

V"SY-X" . {SR,OP1} . {H,S,D} V:=VVS
VF"SY-X" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA * OP2[i] - OP1[i]
    ELSE SR0@[i]:=RTA * OP2[i] - OP1[i]
```

V"SX-Y" . {SR,OP1} . {H,S,D} V:=VVS
VF"SX-Y" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA * OP1[i] - OP2[i]
    ELSE SR0@[i]:=RTA * OP1[i] - OP2[i]
```

V"SX+SY" . {SR,OP1} . {H,S,D} V:=VVS
VF"SX+SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA * (OP1[i] + OP2[i])
    ELSE SR0@[i]:=RTA * (OP1[i] + OP2[i])
```

V"SX-SY" . {SR,OP1} . {H,S,D} V:=VVS
VF"SX-SY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA * (OP1[i] - OP2[i])
    ELSE SR0@[i]:=RTA * (OP1[i] - OP2[i])
```

V"S+XY" . {SR,OP1} . {H,S,D} V:=VVS
VF"S+XY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA + (OP1[i] * OP2[i])
    ELSE SR0@[i]:=RTA + (OP1[i] * OP2[i])
```

V"S-XY" . {SR,OP1} . {H,S,D} V:=VVS
VF"S-XY" . {SR,OP1} . {H,S,D} V:=VVS

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=RTA - (OP1[i] * OP2[i])
    ELSE SR0@[i]:=RTA - (OP1[i] * OP2[i])
```

Two Vector Operands and Two Scalars

S+RX

V"S+RX" . {H,S,D} **V:=VSS**
VF"S+RX" . {H,S,D} **V:=VSS**
VC"S+RX" . {H,S} **V:=VSS**
VFC"S+RX" . {H,S} **V:=VSS**

FOR i:=0 TO SIZEREG-1 DO OP1[i]:=RTA + RTB * OP2[i]

Four Vector Operands

X+YZ

V"X+YZ" . {SR,OP1} . {H,S,D} V:=VVV
VF"X+YZ" . {SR,OP1} . {H,S,D} V:=VVV

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] + OP2[i] * SR0@[i]
    ELSE SR1@[i]:=OP1[i] + OP2[i] * SR0@[i]
```

2.7 Data Moving

MOV — Logical move

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

Purpose: OP1:=OP2. If OP2 has greater precision than OP1, the low-order bits of OP2 are used. If OP2 has smaller precision than OP1, it is zero-extended to the left. This is best thought of as a "logical" or "unsigned" move operation. No condition bits (e.g., carry or integer overflow) are affected. Note that the TRANS instruction can be used to perform sign-extended or truncated integer moves.

It is preferable to use FTRANS rather than MOV on floating point numbers, because the former will execute faster on most implementations.

Restrictions: None

Exceptions: None

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

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

```
MOV.Q.Q RTA,RTA+3
```

The next example shows how MOV extends an integer with zeroes rather than sign bits:

```
MOV.H.Q RTB,#-1          ; RTB := 000777 octal
TRANS.H.Q RTB,#-1        ; RTB := 777777 octal
```

MOVMQ — Move many quarterwords

MOVMQ.{2 .. 32, 64} **XOP**

Purpose: Moves a series of quarterwords beginning with OP2 into the series of quarterwords beginning with OP1, so that OP1:=OP2, NEXT(OP1):=NEXT(OP2), and so on. The modifier specifies how many quarterwords to move. If the source and destination regions overlap, the result is undefined. Unlike vector instructions, MOVMQ can access the registers.

Restrictions: None

Exceptions: None

Precision: This instruction deals with quarterwords for both source and destination precisions.

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

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

MOVMS — Move many singlewords

MOVMS . { 2 .. 32 } **XOP**

Purpose: Moves a series of singlewords beginning with OP2 into the series of singlewords beginning with OP1, so that OP1:=OP2, NEXT(OP1):=NEXT(OP2), and so on. The modifier specifies how many singlewords to move. If the source and destination regions overlap, the result is undefined. Unlike vector instructions, MOVMQ can access the registers.

Restrictions: None

Exceptions: None

Precision: This instruction deals with singlewords for both source and destination precisions.

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

```
MOVMS.28 SAVEBK,RTA
```

The following clears the registers:

```
MOVMS.32 %R0,#0
```

VINI — Vector initialize

VINI . {Q,H,S,D} **V:=S**

Purpose: Initialize each element of a vector OP1 to match the scalar OP2.

Restrictions: None

Exceptions: None

Precision: The elements of the vector OP1, like the scalar OP2, have the precision specified by the modifier.

The following stores in each element of A the value in R0:

```
VINI.S A,R0
```

VREV — Vector reverse

VREV . {H,S,D} **V:=V**

Purpose: Reverse a vector end-for-end by swapping the first element with the last, the second element with the next-to-last, and so on. OP2 is the first element of the source vector and OP1 is the first element of the destination. Either OP1 and OP2 must be identical or the two vectors must not overlap at all; otherwise, the result of the instruction is undefined.

Restrictions: None

Exceptions: None

Precision: The elements of the two vectors have the precision specified by the modifier.

The following stores in DOWN the reverse of the vector in UP:

```
MOV.S.S SIZEREG,#5
VTRANS.S.S UP,[1 ? 1 ? 3 ? 4 ? 5]
VREV.S  DOWN,UP ; DOWN := 5, 4, 3, 2, 1
```

EXCH — Exchange

EXCH . {Q,H,S,D} **XOP**
VEXCH . {Q,H,S,D} **V:=V**

Purpose: EXCH exchanges OP1 with OP2; VEXCH exchanges vector OP1 with vector OP2.

Restrictions: None

Exceptions: None

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

The following swaps RTA and RTB:

```
EXCH.S RTA,RTB
```

One can contrive a situation where the result depends on two rules: the processor prefetches operands, and XOP instructions store OP1 after storing OP2:

```
MOV.S.S RTA,#5
MOV.S.S RTA1,#6
MOV.S.S RTB,#7
EXCH.D RTA,RTA1 ; RTA:=6; RTA1:=7; RTB:=6
                ; (first RTA1:=5 and RTB:=6; then
                ; RTA:=6 and RTA1:=7)
```

SEXCH, USEXCH — Signed and unsigned sorted exchange

SEXCH . {Q,H,S,D} **XOP**
USEXCH . {Q,H,S,D} **XOP**

Purpose: If OP1 > OP2 then exchange OP1 with OP2. The instruction requires read and write access to both OP1 and OP2 even if the inequality is false and no exchange takes place. SEXCH treats the operands as signed integers, whereas USEXCH treats them as unsigned integers.

Restrictions: None

Exceptions: None

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

The following swaps RTA and RTB only if RTA > RTB:

```
SEXCH.S RTA,RTB
```

SLR — Save and load register

SLR . { R0 .. R31 } **XOP**

Purpose: Loosely speaking, the instruction saves the contents of the register specified by the modifier in OP1 and then loads that register with OP2.

More precisely, note that the processor prefetches operands and that XOP instructions store into OP1 last. Thus SLR effectively does the following:

```
TEMP1:=Rn
TEMP2:=OP2
Rn:=TEMP2
OP1:=TEMP1
```

As illustrated below, one can contrive situations where this behavior makes a difference.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords. The modifier must be a multiple of 4 within the range 0 . . 124.

The first instruction moves RTA into RTB and zeros RTA. The second and third instructions show what happens when one of the operands is the register specified in the instruction. The fourth shows what happens when the operands are the same.

```
SLR.RTA RTB,#0   ;RTB:=RTA, RTA:=0
SLR.RTA RTA,F    ;essentially a NOP
                 ; (TMPR:=REG; TMP2:=OP2; REG:=TMP2; OP1:=TMPR)
                 ; (TMPR:=RTA; TMP2:=F; RTA:=TMP2; RTA:=TMPR)

SLR.RTA F,RTA    ;effectively MOV F,RTA
                 ; (TMPR:=RTA; TMP2:=RTA; RTA:=TMP2; F:=TEMPR)

SLR.RTA F,F      ;effectively EXCH RTA,F
                 ; (TMPR:=RTA; TMP2:=F; RTA:=TMP2; F:=TMPR)
```

SLRADR — Save and load register with address

SLRADR . { R0 .. R31 } **XOP**

Purpose: Loosely speaking, the instruction saves in OP1 the register specified by the modifier and then loads the register with ADDRESS(OP2).

Because the processor prefetches operands, and because XOP instructions store into OP1 last, it is more precise to say that:

```
TEMP1:=Rn
Rn:=ADDRESS(OP2)
OP1:=TEMP1
```

As illustrated below, one can concoct examples where this behavior makes a difference.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords. The modifier must be a multiple of 4 in the range 0 . . 124

The first instruction moves RTA into RTB and puts ADDRESS(F) in RTA. The second shows what happens when the first operand is the register specified in the instruction. The third shows what happens when the operands are the same.

```
SLRADR.RTA RTB,F        ;RTB:=RTA, RTA:=ADDRESS(F)
SLRADR.RTA RTA,F        ;effectively a NOP
                        ; (TMP:=REG; REG:=ADDRESS(OP2); OP1:=TMP)
                        ; (TMP:=RTA; RTA:=ADDRESS(F); RTA:=TMP)

SLRADR.RTA F,F          ;same as MOV F,RTA; MOVP.P.A RTA,F
```

ARRIND Array index

ARRIND . {RTA,RTB} XOP

Purpose: RTA := (RTA + OP1 * OP2) Modulo 2^{31} or RTB := (RTB + OP1 * OP2) Modulo 2^{31}. The instruction uses RTA (or RTB) to accumulate an array index.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

Given the following fragment of a Pascal program:

```
TYPE DECADE = 0 .. 9;
VAR
   I, J: DECADE;
   TABLE: ARRAY [DECADE, DECADE] OF INTEGER;
BEGIN
...
TABLE[I,J]:=25;
```

...one might implement the assignment statement with the following code:

```
MOV.S.S RTA,J
ARRIND.RTA #10.,I        ; index is 10 * I + J
MOV.S.S TABLE[RTA]↑2,#25.        ; TABLE[I,J] := 25
```

MOVP — Move pointer

MOVP . {P,R} . {P,R,A} **XOP**

Purpose: Move pointer, optionally transforming it.

This instruction deals with three kinds of "pointers", as the modifiers "P", "R", and "A" indicate. "P" specifies true pointer format, with tag and address. "R" specifies an untagged relative address, simply a signed displacement (in quarterwords) from the address of the pointer itself. "A" specifies the virtual address of the operand instead of the operand itself.

Thus there are six cases:

MOVP.P.P Treat OP2 as a tagged pointer, validate a copy of it according to the rules of Section 1.8.3 (possibly altering the tag or invoking the BAD_P_VALIDATION hard trap) and store the resulting tagged pointer in OP1. A pointer with a fault or reserved tag will cause a BAD_POINTER_TAG hard trap, but a pointer with a NIL or gate tag will not.

MOVP.R.P Treat OP2 as a tagged pointer and validate a copy of it according to the rules of Section 1.8.3 (possibly altering the tag or invoking the BAD_P_VALIDATION hard trap). If the resulting tag is NIL, store the validated pointer in OP1. Otherwise, if the result is a tag for the current ring, subtract ADDRESS(OP1) from the address field within OP2, and store the result in OP1. Otherwise, a BAD_P_VALIDATION hard trap occurs.

This instruction need not check bounds because checking will occur whenever the pointer is converted back to "P" form.

MOVP.P.R If OP2 has a NIL tag, move it to OP1 without change. Otherwise, add OP2 to ADDRESS(OP2) and perform segment bounds checking. Store the address in OP1 along with the tag appropriate to the ring containing OP2.

MOVP.R.R OP2:=OP2+ADDRESS(OP2)-ADDRESS(OP1)

MOVP.P.A Store into OP1 the ADDRESS(OP2) along with the tag appropriate to the ring containing OP2.

MOVP.R.A Store ADDRESS(OP2) - ADDRESS(OP1) into OP1.

In every case, the operand corresponding to the "R" modifier must not be a register, or an ILLEGAL_OPERAND_MODE hard trap will occur. Neither operand may be a constant, or an ILLEGAL_CONSTANT hard trap will occur.

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

The following makes register R0 point to location DATA:

```
MOVP.P.A R0,DATA
```

VALIDP **Validate pointer**

VALIDP **XOP**

Purpose: Validate the pointer OP1 with respect to the ring containing OP2. The address for OP2 is computed following the usual address validation rules, but OP2 itself is not actually fetched. (Thus this operation might cause an OUT_OF_BOUNDS trap, but not a PAGE_FAULT trap.) Then, OP1 is validated and moved to itself using the address validation level of OP2 instead of that of OP1 to derive the new tag. If the tag of OP1 is a ring tag and the number of the ring is less than the validation level of OP2, trap; if the tag of OP1 is a fault or reserved tag, a trap also occurs.

If the tag of OP1 is a user tag and the validation level of OP1 is equal to the validation level of OP2 then preserve the tag.

If the tag of OP1 is a user tag and the validation level of OP1 is greater than the validation level of OP2 then change it to a ring tag corresponding to the validation level of OP1.

Sections 1.8.2 and 1.8.3 describe the address and pointer validation mechanisms.

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

Suppose a process executing in ring 3 has called a routine executing in ring 1, passing it a parameter in register R27. The routine in ring 1 could use the return address saved on the stack--which by definition specifies the caller's ring of execution--to assure that the pointer in R27 is trustworthy. That return address is within the save area pushed by CALLX during the gate crossing (Section 2.12.2) at (SP)-12:

```
VALIDP R27,(SP)-12
```

BASEPTR — Base pointer

BASEPTR — XOP

Purpose: Store in OP1 a pointer to the beginning of the segment containing OP2. (The instruction stores ADDRESS(OP2) in OP1 and then sets to zero the low order SEGSIZE+PGSIZE bits of OP1, where SEGSIZE is the base 2 logarithm of the number of pages in the segment and PGSIZE is the base 2 logarithm of the number of quarterwords in a page.)

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

Make BP point to the beginning of the segment containing the following instruction:

```
BASEPTR BP, .
```

RMW — Read/modify/write

RMW — TOP

Purpose: In one memory cycle (and hence indivisibly with respect to other processors in a multiprocessor system) DEST:=S2 and then S2:=S1. (More precisely, because the processor prefetches operands and because TOP instructions store DEST last, this instruction makes a temporary copy of S2, stores S1 in S2, and then stores the copy into DEST.)

Other atomic instructions are MOVCSF and MOVCSS.

Restrictions: None

Exceptions: None

Precision: S1, S2, and DEST are all singlewords.

The following illustrates the use of RMW to implement a test-and-set lock for interprocessor communication. The lock is a singleword flag which is -1 if some processor has seized the lock and 0 if the lock is free:

```
SEIZE:  RMW RTA,#-1,LOCK        ;attempt to seize lock
        JMPZ.NEQ.S RTA,SEIZE    ;busy-wait if someone else has it
        ...                     ;do ... if lock was zero (now I have it)
FREE:   MOV.S.S LOCK,#0         ;release the lock
```

MOVPHY — Move physical address

MOVPHY — XOP

Purpose: OP1:=PHYSICAL_ADDRESS(OP2). If OP2 is an immediate constant or a register, an ILLEGAL_OPERAND_MODE or ILLEGAL_CONSTANT hard trap will occur.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword.

The following loads RTA with the *physical* address of F:

```
MOVPHY RTA,F     ;RTA:=PHYSICAL_ADDRESS(F)
```

RPHYS, WPHYS — Read/write physically addressed location

RPHYS **XOP**
WPHYS **XOP**

Purpose: RPHYS reads into OP1 the contents of a memory location whose physical address is specified by the 34 low order bits of R3. WPHYS writes OP1 into a memory location whose address is specified by the 34 low order bits of R3.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. R3 is a singleword whose 34 low order bits are a physical address. OP2 is unused.

The following moves SOURCE to DESTINATION even if the mapping tables are changed following the first two instructions:

```
MOVPHY R3,SOURCE
MOVPHY R2,DESTINATION
...
RPHYS  RTA
EXCH.S R3,R2
WPHYS  RTA
```

MOVHWR — Move hardware representation

MOVHWR . {N,C} . {1,16} **XOP**

Purpose: This implementation-dependent instruction exists for use by memory diagnostics. It reads words from a block beginning with OP2 and writes them to a block beginning with OP1, bypassing the cache. Depending on the second modifier, it copies either 1 or 16 singlewords.

If the first modifier is N (for "no correction"), the instruction copies each singleword along with its associated error-correction bits into a doubleword, right-justified with leading zeros, instead of applying the error correction algorithm. If the first modifier is C ("correction"), the instruction copies source singlewords into destination singlewords, applying the correction algorithm and then discarding the error-correction bits.

Restrictions: None.

Exceptions: None

Precision: OP2 is the first element of a vector of {1,16} singlewords. For MOVHWR.N, OP1 is the first element of a vector of {1,16} doublewords; for MOVHWR.C, OP1 is the first of a vector of {1,16} singlewords.

The following example copies a vector of 16 singlewords into a vector of 16 doublewords, revealing the error-correction bits:

```
MOVHWR.N.16 DEST,SOURCE
```

2.8 Skip, Jump, and Comparison

Skip and jump instructions branch to locations other than that of the next sequential instruction. Skip instructions branch within a short range while jumps branch anywhere in the 2^{29} singleword address space.

Many skips or jumps occur only if a condition specified by a modifier to the instruction is true. An *arithmetic condition* (ACOND) can be any of the following :

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

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

The SKP instruction may use a *logical condition* (LCOND) as well. The LCONDs are:

LCOND = {NON,ALL,ANY,NAL}

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

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

Combining the ACONDs and the LCONDs gives the arithmetic and logical conditions (ALCONDs):

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

SKP — Skip on condition

SKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ,NON,ALL,ANY,NAL} . {Q,H,S,D} **SOP**

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 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: None

Precision: The precision of OP1 and OP2 is specified by the second modifier.

The following instructions compute the function "IF RTA is Odd THEN BEGIN RTA:=3*RTA+1 END; RTA:=RTA/2;" repeatedly while RTA>1. Note that FASM determines the SW offset automatically from the JUMPDEST operand:

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

ISKP — Increment, then skip on condition

ISKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **SOP**

Purpose: OP1:=OP1+1. CARRY is not affected. Then if OP1 ACOND OP2 (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to a location that is within -8 . . 7 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 and OP2 are both singlewords.

The following is a typical loop of the form, "FOR I:=M TO N DO ...". The inner part of the loop must not exceed 8 singlewords when assembled:

```
        MOV.S.S I,M
LOOP:
        ...
        ISKP.LEQ I,N,LOOP
```

DSKP — Decrement, then skip on condition

DSKP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} SOP

Purpose: OP1:=OP1-1. CARRY is not affected. Then if OP1 ACOND OP2 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to a location that is within -8 . . 7 singlewords of the current PC. If the comparison is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 and OP2 are both singlewords.

The following instructions search an array of N singlewords starting at TABLE for the largest index I such that TABLE[I]=I. Assume that TABLE[0] contains 0 to ensure loop termination, and that N singlewords follow this entry. In the following, I must be a register. Note that since the loop is one instruction long the singleword skip offset is zero. The "-4" added to the base address TABLE compensates for the fact that the address calculation occurs *before* the decrementation operation, but the skip condition is tested *after* the decrementation operation. In turn, "N+1" is used instead of "N" in the initialization to compensate for this compensation:

```
        MOV.S.S I,#<N+1>
LOOP:   DSKP.NEQ I,<TABLE-4>[I]↑2,LOOP
```

JMP — Jump on condition

JMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} JOP

Purpose: If FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR, EQL, GEQ, LSS, NEQ, LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: None

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords which are together treated as a doubleword.

The following loop searches down a chain of pointers for a specified tail pointer FPTR. Let P be a register and HEAD the address of the first link in the chain. Note that NEXT(P) is implicitly used by this routine to hold the comparison operand:

```
        MOV.D.D P,#<[HEAD ? FPTR]>      ;initialize P and NEXT(P)
                                        ;(this is an assembler literal
                                        ; whose address becomes a constant)
LOOP:   MOV.S.S P,(P)
        JMP.NEQ P,LOOP
```

JMPZ — Jump on condition relative to zero

JMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} JOP

Purpose: If OP1 ACOND 0 is true (where ACOND∈{GTR, EQL, GEQ, LSS, NEQ, LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: None

Precision: OP1 has the precision specified by the second modifier.

By using the indexed constant addressing mode (Section 1.6.2), a programmer can use the JMPZ instruction to compare the contents of a register against any integer constant, not just against zero. For example, the following jumps to AWAY iff RTA≤1:

```
JMPZ.LEQ.S #[-1](RTA),AWAY
```

JMPA — Jump always

JMPA — JOP

Purpose: Jump unconditionally to JUMPDEST. For a simple jump to a label, the SJMP instruction is often more compact, but JMPA allows indexing and indirect addressing, usually at the expense of an extra singleword.

Restrictions: None

Exceptions: None

Precision: None

The following instruction jumps to the RTA-th address stored in a list of indirect pointers that begins at JVECTS:

```
JMPA  JVECTS [RTA] ↑2@
```

IJMP **Increment, then jump on condition**

IJMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: FIRST(OP1):=FIRST(OP1)+1. CARRY is not affected. Then if FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords which together are treated as a doubleword.

The following is a typical loop of the form, "FOR I:=M TO N DO . . .". The inner part of the loop may be any length when assembled:

```
        MOV.D.D I,[M ? N]              ;M,N are assembly literals
LOOP:
        ...
        IJMP.LEQ I,LOOP
```

IJMPZ — Increment, then jump on condition relative to zero

IJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: OP1:=OP1+1. CARRY is not affected. Then if OP1 ACOND 0 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 is a singleword.

The following increments N and jumps to AWAY if N=0:

```
IJMPZ.EQL N,AWAY
```

IJMPA — Increment and jump always

IJMPA — JOP

Purpose: OP1:=OP1+1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Restrictions: None

Exceptions: INT_OVFL may be set by the incrementing operation.

Precision: OP1 is a singleword.

The following is an extremely inefficient way to add RTA into RTB, assuming that integer overflow traps are disabled. However, it shows off the IJMPA instruction:

```
LOOP:   DSKP.EQL RTA,#-1        ;decrement RTA; skip next instruction if -1
        IJMPA RTB,LOOP          ;otherwise increment RTB and loop
```

DJMP — Decrement, then jump on condition

DJMP . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: FIRST(OP1):=FIRST(OP1)-1. CARRY is not affected. Then if FIRST(OP1) ACOND SECOND(OP1) is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: FIRST(OP1) and SECOND(OP1) are both singlewords which together are treated as a doubleword.

The following is a typical loop of the form, "FOR I:=M DOWNTO N DO...". The inner part of the loop may be any length when assembled:

```
        MOV.D.D I,[M ? N]              ;M,N are assembly literals
LOOP:
        ...
        DJMP.GEQ I,LOOP
```

DJMPZ — Decrement, then jump on condition relative to zero

DJMPZ . {GTR,EQL,GEQ,LSS,NEQ,LEQ} **JOP**

Purpose: OP1:=OP1-1. CARRY is not affected. Then if OP1 ACOND 0 is true (where ACOND∈{GTR,EQL,GEQ,LSS,NEQ,LEQ}), control is transferred to the location specified by JUMPDEST. If the condition is false, control is transferred to the next instruction.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 is a singleword.

The following decrements N and jumps to AWAY if N=0:

```
DJMPZ.EQL N,AWAY
```

DJMPA — Decrement and jump always

DJMPA **JOP**

Purpose: OP1:=OP1-1. CARRY is not affected. Jump unconditionally to JUMPDEST.

Restrictions: None

Exceptions: INT_OVFL may be set by the decrementing operation.

Precision: OP1 is a singleword.

The following decrements N and jumps to AWAY:

```
DJMPA N,AWAY
```

SJMP **Simple jump**

SJMP **HOP**

Purpose: Unconditionally jump anywhere in the address space.

The HOP format performs a PC-relative jump using a 29 bit unsigned displacement field. Because the address calculation "wraps around" if it exceeds the maximum virtual address, it can reach any singleword in the virtual address space.

While SJMP never occupies more than 1 singleword, it allows only a direct memory address reference. One must use JMPA for any other addressing mode, such as indexing or indirect addressing.

Restrictions: None

Exceptions: None

Precision: None

Go to CRUNCH:

```
SJMP CRUNCH
```

MOVCSF, MOVCSS — Move conditionally, skip on failure/success

MOVCSF . {Q,H,S,D} **SOP**
MOVCSS . {Q,H,S,D} **SOP**

Purpose: For MOVCSF, IF OP1=OP2 THEN OP1:=%R3 ELSE GOTO DEST.

For MOVCSS, IF OP1=OP2 THEN BEGIN OP1:=%R3; GOTO DEST END.

In a multiprocessor system, these instructions are atomic (that is, they finish work on OP1 before any other processor can alter that operand). Another atomic instruction is RMW.

Restrictions: None

Exceptions: None

Precision: OP1, OP2, and %R3 have the precision specified by the modifier.

Singleword LOCK represents a lock, which holds -1 if unlocked and 0 if locked. The following sequence seizes the lock, using busy-waiting if the lock is not free:

```
;;; Seize the lock stored in location LOCK.
        MOV.S.S %R3,#-1          ;Prepare the value -1 to be stored.
LOOP:   MOVCSF.S LOCK,#0,LOOP    ;Store -1 when LOCK holds 0.
```

The following code sequence atomically turns on bit 35 of word F01.

```
;;; Turn on bit 35 of word F01.
LOOP:   MOV.S.S RTA,F01          ;Pick up a copy of the former value of F01.
        OR.S %R3,RTA,#2          ;Turn on bit 35, creating the new value in %R3.
        MOVCSF.S F01,RTA,LOOP    ;Store the new value if the value has not
                                 ;changed since we began.
```

The following code sequence leaves in %R3 a unique number; no two callers will ever be returned the same number even if they run this routine simultaneously from different processors. The location UNIQUE holds a number, whose value is increased by one atomically to get the new unique value.

```
;;; Return a unique value in %R3.
LOOP:   MOV.S.S RTA,UNIQUE       ;Get the old value of UNIQUE.
        ADD.S.S %R3,RTA,#1       ;The new value should be one greater.
        MOVCSF.S UNIQUE,RTA,LOOP        ;Store the new value if the value
                                 ;of UNIQUE has not changed in the meantime.
```

The following code sequence atomically adds a new element to a singly linked list. The pointer to the first list element is stored in location HEAD; the first word of each element contains a pointer to the next element. Register %R3 contains a pointer to a new element to be added to the head of the list.

```
;;; Add the element in %R3 to the list.
LOOP:   MOV.S.S RTA,HEAD          ;Pick up the pointer to the former first
                                  ;element of the list.
        MOV.S.S (%R3),RTA         ;Make the new element point to it.
        MOVCSF.S HEAD,RTA,LOOP    ;Store the new pointer if the old one
                                  ;has not changed.
```

CMPSF, UCMPSF — Signed/unsigned compare and set flag

CMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} **TOP**
UCMPSF . {GTR,EQL,GEQ,LSS,NEQ,LEQ} . {Q,H,S,D} **TOP**

Purpose: If S1 *condition* S2 then DEST := -1 else DEST := 0, where *condition* is the first modifier. CMPSF performs a two's complement signed comparison whereas UCMPSF performs an unsigned comparison.

Restrictions: None

Exceptions: None

Precision: S1 and S2 have the same precision as the modifier. DEST is a singleword.

Let X, Y, and Z be singlewords, with Y=NEXT(X). The following code implements setting RTA to X if Z≥0 and to Y otherwise. It uses indexing rather than a conditional jump or skip. Such use of indexing can often make more effective use of instruction pipelining than jumping or skipping:

```
CMPSF.GEQ.S RTA,Z,#0
MOV.S.S RTA,Y[RTA]↑2     ;indexing with flag result
```

CMPSF.LSS can be used to produce an extended-sign word for a number. TRANS or FTRANS can be used to sign-extend a number to one of the four standard precisions, but this trick is useful in dealing with numbers of very large precision:

```
CMPSF.LSS.S RTA,NUM,#0  ;all bits of RTA get the sign bit of NUM
```

Though instructions CMPSF.{NON,ALL,ANY,NAL} do not exist, their effect can be obtained by an AND or ANDCT followed by a CMPSF.EQL or CMPSF.NEQ:

```
ANDCT.S RTA,ARG1,ARG2    ;this behaves as would the fictional
CMPSF.EQL.S RTA,#0       ; Instruction  CMPSF.ALL RTA,ARG1,ARG2
```

BNDSF — Bounds-check and set flag

BNDSF . {B,MIN,M1,0,1} . {Q,H,S,D} **TOP**

Purpose: Check S2 against the bounds specified by the first modifier and by S1. If S2 is within bounds then DEST := -1 else DEST := 0. The following table explains the first modifier:

Modifier	Meaning
B ("both")	FIRST(S1) ≤ S2 ≤ SECOND(S1)
MIN	MINNUM ≤ S2 ≤ S1
M1	-1 ≤ S2 ≤ S1
0	0 ≤ S2 ≤ S1
1	1 ≤ S2 ≤ S1

Restrictions: None

Exceptions: None

Precision: DEST is a singleword. S2 has the precision specified by the second modifier. If the first modifier is B, then FIRST(S1) and SECOND(S1) have the same precision as S2 and must align together to form a single entity with twice that precision; otherwise, S1 has the same precision as S2.

This first example shows a standard way to use BNDSF:

```
BNDSF.0.S RTA,#LIMIT,X          ; 0  ≤ X ≤ #LIMIT
```

This second example shows how to use a constant addressing mode to obtain a different kind of check. This makes use of the rule that a singleword instruction which expects a FIRST/SECOND operand pair will expand a constant to twice the specified precision and use half for the FIRST part and half for the SECOND part:

```
BNDSF.B.S RTA,#[LIMIT ? !0],X   ; #LIMIT ≤ X ≤ 0
```

BNDTRP — Bounds check and trap on failure

BNDTRP . {B,MIN,M1,0,1} . {Q,H,S,D} **XOP**

Purpose: Check OP2 against the bounds specified by the first modifier and by OP1. If OP2 is out of bounds then a BOUNDS_CHECK soft trap will occur. The following table explains the first modifier:

Modifier	Meaning
B ("both")	FIRST(OP1) ≤ OP2 ≤ SECOND(OP1)
MIN	MINNUM ≤ OP2 ≤ OP1
M1	-1 ≤ OP2 ≤ OP1
0	0 ≤ OP2 ≤ OP1
1	1 ≤ OP2 ≤ OP1

Restrictions: None

Exceptions: None

Precision: OP2, the upper bound, and the lower bound all have the precision specified by the second modifier. If the first modifier is B, then the instruction uses FIRST(OP1) and SECOND(OP1); thus, each has the precision specified by the second modifier, but both must align to form an entity with twice that precision.

The following instruction traps if |RTA|>2.0:

```
BNDTRP.B.S [-2.0 ? 2.0],RTA
```

STRCMP String compare

STRCMP . {RTA,RTB} XOP

Purpose: Consider OP1 and OP2 to be vectors of quarterwords--in other words, strings of characters--whose quarterword length is specified by SIZEREG. Signed comparison is used, and each quarterword character is compared separately. The result of the comparison is computed as shown in the following table and is stored into {RTA,RTB}. The result values are designed to have two useful properties. First, the result (as a signed integer) bears the same relation to zero that STRING1 does to STRING2. Second, the value can be used as an index into the string no matter what the result, because indexing arithmetic "wraps around" the address space.

Condition	Result
STRING1 = STRING2	0
STRING1 > STRING2	n
STRING1 < STRING2	-2^{35}+n (i.e. MINNUM+n)

(n is the position of the first character to differ)

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are quarterword vectors, and thus may designate registers. RTA and RTB are single words.

The following sets RTA to the result of comparing the eighty-character blocks at X and Y.

```
MOV.S.S %SIZEREG,#80.
STRCMP.RTA X,Y
```

The following illustrates a more general sort of comparison. Assume that XLENGTH contains the length of a string beginning at X and YLENGTH that of string at Y. For the purposes of this comparison we will imagine that appended to the two strings are infinitely many imaginary characters defined to be "less than" all real characters. We will then define the result of the comparison as the result of a STRCMP performed on these extended strings. (This definition is similar to that used in some high-level languages):

```
MIN.S RTA,XLENGTH,YLENGTH       ;set RTA to minimum real length
MOV.S.S %SIZEREG,RTA
INC.S RTB,RTA                   ;save one greater in RTB for unequal case
STRCMP.RTA X,Y                  ;do comparison
JMPZ.NEQ.S RTA,DONE             ;difference found
SKP.NEQ.S XLENGTH,YLENGTH       ;done if strings are equal length
JMPA DONE
```

```
        MOV.S.S RTA,RTB                 ;RTB is index of "imaginary" character
        SKP.LEQ.S XLENGTH,YLENGTH,DONE  ;set high-order bit if necessary
        OR.S RTA,#<400000,,0>           ;or  DIBYT RTA,#1,#1  to save a word!
DONE:   ...                             ;RTA contains result
```

2.9 Shift, Rotate, and Bit Manipulation

These instructions all manipulate bits within a word, either by shifting, by rotating, or by performing bitwise logical functions. Note that a left shift (or rotate) by N is equivalent to the corresponding right shift (or rotate) by -N. The SHFA instruction, which shifts signed integers, appears in Section 2.1 with the other signed integer arithmetic instructions.

NOT — Logical bit-wise NOT

NOT . {Q,H,S,D} **XOP**
VNOT . {H,S,D} **V:=V**

Purpose: NOT computes OP1:=(¬OP2), where "¬" signifies *one's complement*

VNOT performs NOT on each element of the vector beginning with OP2 and stores the result in the corresponding element of the vector beginning with OP1.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 (or the elements of vectors OP1 and OP2) have the same precision as the modifier.

The following is an alternate to NEG RTA:

```
NOT.S RTA,#[-1](RTA)     ;RTA := -RTA
```

AND — Logical bit-wise AND

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

Purpose: AND computes DEST:=S1∧S2.

VAND performs AND on each element of vector OP1 and the corresponding element of OP2. It puts the results either back into vector OP1 or into the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] ∧ OP2[i]
    ELSE SR0@[i]:=OP1[i] ∧ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For AND, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VAND, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of AND:

```
AND.Q RTA,#3,#5          ;RTA:=1 (QW)
```

ANDTC — Logical bit-wise AND true/complement

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

Purpose: DEST:=S1∧(¬S2). Note that the "TC" in ANDTC means "True-Complement" and refers to the fact that S1 and *one's-complement*(S2) respectively are operands to the AND function. The reverse form of ANDTC is ANDCT, *not* ANDTCV.

VANDTC performs ANDTC on pairs of corresponding elements in the vectors beginning at OP1 and OP2. It puts the results back into the vector OP1 or into the vector pointed to by SR0, depending on the first modifier.

```
FOR I:=0 TO SIZEREC-1 DO
    IF {modifier OP1} THEN OP1[i]:=OP1[i] ∧ (¬OP2[i])
    ELSE SR0@[i]:=OP1[i] ∧ (¬OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For ANDTC, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VANDTC, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ANDTC:

```
ANDTC.Q RTA,#3,#5              ;RTA:=2 (QW)
```

Suppose that MASK is a mask whose "1" bits select certain (possibly non-contiguous!) bits of WORD. These bits are to be regarded as a "field", and the contents of that field *decremented* as an integer "in place" in WORD, without affecting non-selected bits of WORD. This can be done as follows:

```
AND.S RTA,WORD,MASK     ;RTA:=WORD with non-selected bits zeroed
SUB.S RTA,#1            ;zeroed bits propagate the borrow
AND.S RTA,MASK          ;mask out non-selected bits
ANDTC.S WORD,MASK       ;mask out SELECTED bits in WORD
OR.S WORD,RTA           ;merge the two results
```

ANDCT — Logical bit-wise AND complement/true

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

Purpose: ANDCT computes DEST:=(¬S1)∧S2. Note that the "CT" in ANDCT means "Complement-True" and refers to the fact that *one's-complement*(S1) and S2 respectively are operands to the AND function. The reverse form of ANDCT is ANDTC, *not* ANDCTV.

VANDCT performs ANDCT on pairs of elements from the vectors beginning at OP1 and OP2. It puts the results back into the vector OP1 or into the vector pointed to by SR0, depending on the first modifier.

```
FOR I:=0 TO SIZEREG-1 DO
    IF {modifier OP1} THEN OP1[i]:=(¬OP1[i]) ∧ OP2[i]
    ELSE SR0@[i]:=(¬OP1[i]) ∧ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For ANDCT, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VANDCT, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ANDCT:

```
ANDCT.Q RTA,#3,#5                ;RTA:=4 (QW)
```

OR — Logical bit-wise OR

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

Purpose: OR computes DEST:=S1∨S2.

VOR performs OR on pairs of elements from the vectors OP1 and OP2, putting the results into vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] ∨ OP2[i]
    ELSE SR0@[i]:=OP1[i] ∨ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For OR, S1, S2, and DEST all have the precision specified by the modifier {Q,H,S,D}. For VOR, the elements of the vectors all have the precision specified by the modifier {H,S,D}.

The following instruction illustrates the effect of OR:

```
OR.Q RTA,#3,#5          ;RTA:=7 (QW)
```

ORTC — Logical bit-wise OR true/complement

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

Purpose: ORTC computes DEST:=S1∨(¬S2). Note that the "TC" in ORTC means "True-Complement" and refers to the fact that S1 and *one's-complement*(S2) respectively are operands to the OR function. The reverse form of ORTC is ORCT, *not* ORTCV.

VORTC performs ORTC on pairs of elements of the vectors OP1 and OP2, putting the results in either vector OP1 or the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=OP1[i] ∨ (¬OP2[i])
    ELSE SR0@[i]:=OP1[i] ∨ (¬OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For ORTC, S1, S2, and DEST all have the precision specified by the second modifier. For VORTC, the elements of the vectors all have the precision specified by the second modifier.

The following instruction illustrates the effect of ORTC:

```
ORTC.Q RTA,#3,#5          ;RTA:=773 (QW)
```

Suppose that MASK is a mask whose one-bits select certain (possibly non-contiguous!) bits of WORD. These bits are to be regarded as a "field", and the contents of that field *incremented* as an integer "in place" in WORD, without affecting non-selected bits of WORD. This can be done as follows:

```
ORTC.S RTA,WORD,MASK      ;RTA:=WORD with non-selected bits set to "1"
ADD.S RTA,#1              ;"1" bits propagate the carry
AND.S RTA,MASK            ;mask out non-selected bits
ANDTC.S WORD,MASK         ;mask out SELECTED bits in WORD
OR.S WORD,RTA             ;merge the two results
```

ORCT — Logical bit-wise OR complement/true

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

Purpose: ORCT computes DEST:=(¬S1)∨S2. Note that the "CT" in ORCT means "Complement-True" and refers to the fact that *one's-complement*(S1) and S2 respectively are operands to the OR function. The reverse form of ORCT is ORTC, *not* ORCTV.

VORCT performs ORCT on pairs of elements of vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=(¬OP1[i]) ∨ OP2[i]
    ELSE SR0@[i]:=(¬OP1[i]) ∨ OP2[i]
```

Restrictions: None

Exceptions: None

Precision: For ORCT, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VORCT, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of ORCT:

```
ORCT.Q RTA,#3,#5          ;RTA:=775 (QW)
```

NAND — Logical bit-wise NAND

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

Purpose: NAND computes DEST:=¬(S1∧S2).

VNAND performs NAND on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=¬(OP1[i]) ∧ OP2[i]
    ELSE SR0@[i]:=¬(OP1[i] ∧ OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For NAND, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VNAND, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of NAND:

```
NAND.Q RTA,#3,#5          ;RTA:=776 (QW)
```

NOR — Logical bit-wise NOR

NOR . {Q,H,S,D} **TOP**
VNOR . {SR,OP1} . {H,S,D} **V:=VV**

Purpose: NOR computes DEST:=¬(S1∨S2), where "¬" signifies *one's complement.*

VNOR performs NOR on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=¬(OP1[i] ∨ OP2[i])
    ELSE SR0@[i]:=¬(OP1[i] ∨ OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For NOR, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VNOR, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of NOR:

```
NOR.Q RTA,#3,#5          ;RTA:=770 (QW)
```

XOR **Logical bit-wise XOR**

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

Purpose: XOR computes DEST:=(S1∧¬(S2)) ∨ (¬(S1)∧S2), where "¬" represents the one's complement operation.

VXOR performs XOR on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, depending on the first modifier:

```
FOR i:=0 TO SIZEREG-1 DO
    IF {modifier OP} THEN OP1[i]:=ExclusiveOR(OP1[i],OP2[i])
    ELSE SR0@[i]:=ExclusiveOR(OP1[i],OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For XOR, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VXOR, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of XOR:

```
XOR.Q RTA,#3,#5          ;RTA:=6 (QW)
```

The following code exchanges the two words QUUX and ZTESCH. (A better way to do this is with the EXCH instruction, but this example demonstrates an interesting information-preserving property of XOR.)

```
XOR.S QUUX,ZTESCH
XOR.S ZTESCH,QUUX
XOR.S QUUX,ZTESCH
```

EQV — Logical bit-wise equivalence

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

Purpose: EQV computes DEST:=(S1∧S2) ∨ (¬(S1)∧(¬S2)), where "¬" represents the one's complement operation.

VEQV performs EQV on pairs of elements of the vectors OP1 and OP2, putting the results either in vector OP1 or in the vector pointed to by SR0, according to the first modifier:

```
FOR i:=0 TO SIZEREG 1 DO
    IF {modifier OP} THEN OP1[i]:=EQV(OP1[i],OP2[i])
    ELSE SR0@[i]:=EQV(OP1[i],OP2[i])
```

Restrictions: None

Exceptions: None

Precision: For EQV, S1, S2, and DEST all have the precision specified by the {Q,H,S,D} modifier. For VEQV, the elements of the vectors all have the precision specified by the {H,S,D} modifier.

The following instruction illustrates the effect of EQV:

```
EQV.Q RTA,#3,#5          ;RTA:=771 (QW)
```

The following code exchanges the two words QUUX and ZTESCH. (A better way to do this is with the EXCH instruction, but this example demonstrates an interesting information-preserving property of EQV.)

```
EQV.S QUUX,ZTESCH
EQV.S ZTESCH,QUUX
EQV.S QUUX,ZTESCH
```

SHFA — Shift arithmetically

SHFA . {LF,RT} . {Q,H,S,D} **TOP**
SHFAV . {LF,RT} . {Q,H,S,D} **TOP**
VSHFA . {LF,RT} . {H,S,D} **V:=VS**

Purpose: SHFA computes DEST:=S1 arithmetically shifted {left,right} by S2. Shifts to the (true) left introduce "0" bits; shifts to the (true) right replicate the sign bit and discard bits shifted out the low end. This is equivalent to a multiplication or division by a power of two, where it is understood that such a division rounds towards negative infinity. 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 to a right shift by -S1. If the absolute value of S2 exceeds the width of the anyword being shifted, an ILLEGAL_SHIFT_ROTATE hard trap occurs.

SHFAV swaps the roles of S1 and S2.

VSHFA performs SHFA on each element of the vector beginning at OP2 and stores the results in the corresponding elements of OP1. RTA specifies how far to shift each element.

Restrictions: None

Exceptions: INT_OVFL (the instruction behaves exactly as would a multiplication by a power of 2)

Precision: For SHFA, S2 is a singleword, and DEST and S1 have the precision specified by the second modifier.

For SHFAV, S1 is a singleword, and DEST and S2 have the precision specified by the second modifier.

For VSHFA, the elements of vectors OP1 and OP2 have the precision of the modifier and RTA is a scalar singleword.

The following two instructions illustrate the difference between SHF.RT and SHFA.RT:

```
SHF.RT.Q RTA,#-1,#1     ;RTA:=377
SHFA.RT.Q RTA,#-1,#1    ;RTA:=777
```

SHF — Logical shift

SHF . {LF,RT} . {Q,H,S,D} **TOP**
SHFV . {LF,RT} . {Q,H,S,D} **TOP**
VSHF . {LF,RT} . {H,S,D} **V:=VS**

Purpose: SHF computes DEST:=S1 logically shifted {left,right} by S2. Bits shifted in are "0" bits; bits shifted out are lost. Note that a left shift by S2 is identical to a right shift by -S2. If the absolute value of S2 exceeds the width of the anyword being shifted, an ILLEGAL_SHIFT_ROTATE hard trap occurs.

SHFV, the reverse form, behaves identically except that it swaps the roles and precisions of S1 and S2.

VSHF performs SHF on each element of the vector beginning with OP2 and stores the results in the corresponding elements of the vector beginning with OP1. RTA specifies the number of bit positions by which to shift.

Restrictions: None

Exceptions: None

Precision: For SHF, S2 is a singleword; DEST and S1 have the precision specified by the second modifier. For SHFV, S1 is a singleword; DEST and S2 have the precision of the second modifier. For VSHF, RTA is a singleword; the elements of OP1 and OP2 have the precision specified by the modifier.

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

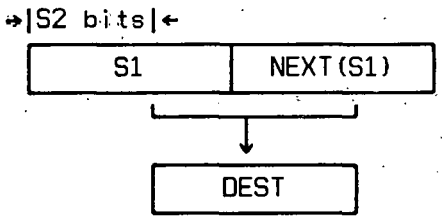
```
SHF.LF.Q RTA,#-1,#1      ;RTA:=-2 (QW)
```

DSHF — Extended logical shift

DSHF . {LF,RT} . {Q,H,S} **TOP**
DSHFV . {LF,RT} . {Q,H,S} **TOP**

Purpose: Just as a programmer can use the ADDC instruction repeatedly to add numbers of arbitrarily great precision, the programmer can use the DSHF instruction repeatedly to shift an arbitrarily long string of bits. Ordinary logical shift instructions are difficult to chain in this fashion because they shift zeros into the word. DSHF solves the problem by shifting in bits from the adjacent word in memory instead.

More precisely, DSHF.LF concatenates S1 and NEXT(S1), logically shifts the resulting double precision entity left by S2 bits and stores in DEST the high order 9, 18, or 36 bits (corresponding to Q, H, or S precisions). DSHF.RT logically shifts the entire entity right by S2 bits and stores in NEXT(DEST) the low order 9, 18, or 36 bits.



DSHF.LF DEST,S1,S2

Careful use of DSHF even permits in-place shifting--that is, leaving the result of the shifting in the original memory locations: right shifts must start at the right end of the series of words, and long left shifts must start at the left end.

An ILLEGAL_SHIFT_ROTATE hard trap occurs if the absolute value of S2 exceeds the width of the anyword being shifted.

DSHFV, the reverse form, swaps the roles of S1 and S2.

See also the vector instruction VDSHF.

Restrictions: None

Exceptions: None

Precision: For DSHF, operands S1, NEXT(S1), and DEST (or NEXT(DEST)) all have the precision specified by the modifier. S2 is a singleword. S1 and NEXT(S1) need not be aligned specially: using DSHF.H, for example, S1 must be a properly aligned halfword, but S1 and NEXT(S1) together need not be a properly aligned singleword.

For DSHFV, the same is true except that the roles of S1 and S2 are swapped.

The following illustrates the result of shifting a long operand:

```
        MOV.H.H   %R8,#123456
        DSHF.LF.Q RTA,%R8,#1     ;RTA:=247 (high-order QW of RTA)
```

Suppose that a 30-word block of bits MARKERS is to be logically shifted in place three bits to the left. While using VDSHF provides better performance, the following example illustrates the use of DSHF within an explicit loop:

```
        MOV.S.S RTB,#0              ;RTB indexes MARKERS from left to right
LOOP:   DSHF.LF.S MARKERS[RTB]↑2,#3  ;produce one result word
        ISKP.LSS RTB,#29.,LOOP      ;increment RTB and loop if < 29.
        SHF.LF.S MARKERS+29.*4,#3           ;do 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.S.S RTB,#29.                    ;RTB indexes MARKERS from right to left
LOOP:   DSHF.RT.S MARKERS[RTB]↑2,#3  ;produce one result word
        DSKP.GTR RTB,#0,LOOP        ;decrement RTB and loop if > 0
        SHF.RT.S MARKERS,#3         ;do the last word in single precision
```

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

VDSHF — Lengthwise vector logical shift

VDSHF . {LF,RT} V:=VS

Purpose: Logically shift an arbitrarily long series of bits. OP2 is the first word of the source vector, OP1 is the first word of the destination vector, SIZEREG gives the length of the vector in singlewords, and RTA specifies how far to shift the bits.

If the source and destination vectors overlap at all, they must coincide completely, or the result is undefined. An ILLEGAL_SHIFT_ROTATE hard trap occurs if the absolute value of RTA is greater than 36.

VDSHF.RT does not alter the first word of the vector, and VDSHF.LF does not alter the last word. This allows the programmer to use a scalar shift or rotate instruction to finish the operation, and thereby obtain a logical shift, arithmetic shift, or rotation. This also permits chaining of VDSHF instructions.

This instruction accomplishes the same task as a loop that applies the scalar DSHF instruction to a series of words, one at a time (see the example under the discussion of DSHF). For all but the shortest series of bits, the vector version will execute more rapidly, but the scalar version gives a choice of precisions.

Restrictions: None

Exceptions: None

Precision: The elements of both vectors are singlewords in terms of alignment (though the instruction can operate on larger sections of the vector to achieve greater speed). RTA and SIZEREG are singlewords.

This is a simple illustration of VDSHF and SHF combined to perform a logical shift:

```
MOV.S.S SIZEREG,#3        ; Length of vector is 3 singlewords
MOV.S.S RTA,#19.          ; Shift by 19 bit positions
VTRANS.S.S SOURCE,[1,,2 ? 3,,4 ? 5,,6]
                          ; "a,,b" tells FASM to put a in
                          ;   the left halfword, b in the right
VDSHF.LF DEST,SOURCE                    ; Result is
SHF.LF.S <SOURCE-4*1>[SIZEREG]↑2,RTA    ;    [4,,6 ? 8.,,10. ? 12.,,0]
```

ROT — Logical rotate

ROT . {LF,RT} . {Q,H,S,D} **TOP**
ROTV . {LF,RT} . {Q,H,S,D} **TOP**

Purpose: ROT computes DEST:=S1 rotated {left,right} by S2. Rotation introduces bits shifted out of one end into the other end, so that no bits are lost. An ILLEGAL_SHIFT_ROTATE hard trap occurs if the absolute value of S2 exceeds the width of the anyword being shifted.

ROTV, the reverse form, rotates S2 left or right by S1 bits.

Restrictions: None

Exceptions: None

Precision: For ROT, S2 is a singleword. DEST and S1 have the precision specified by the second modifier.

For ROTV, S1 is a singleword; DEST and S2 have the precision of the second modifier.

The following illustrates a right rotation by a positive amount:

```
ROT.RT.Q RTA,#1,#1      ;RTA:=400 (QW)
```

BITRV — Bit reverse

BITRV . {Q,H,S,D} **TOP**
BITRVV . {Q,H,S,D} **TOP**

Purpose: BITRV reverses the order of the S2 low-order bits of S1, and zero-extends the result into DEST. An ILLEGAL_SHIFT_ROTATE hard trap occurs if S2 is negative or exceeds the word width.

BITRVV reverses the order of the S1 low-order bits of S2 instead.

Restrictions: None

Exceptions: None

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

For BITRVV, S2 and DEST have the precision of the modifier; S1 is a singleword.

The following reverses all nine bits of its operand:

```
BITRV.Q RTA,#[123],#9.   ;RTA:=624 (QW)
```

BITEX — Bit extract

BITEX . {Q,H,S,D} **TOP**
BITEXV . {Q,H,S,D} **TOP**

Purpose: BITEX extracts the bits of S1 selected by the "1" bits of S2. It squeezes these selected bits to the right, zero-extends them, and stores them into DEST.

BITEXV, the reverse form, swaps the roles of S1 and S2.

Restrictions: None

Exceptions: None

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

The following extracts alternate bits from the operand:

```
BITEX.Q RTA,#[765],#[525]        ;RTA:=37 (QW)
```

BITCNT — Bit count

BITCNT . {Q,H,S,D} **XOP**
VBITCNT . {H,S,D} **V:=V**
LBITCNT . {H,S,D} **S:=V**

Purpose: BITCNT computes OP1:=*number of "1" bits in* OP2. This instruction is useful for counting the number of elements in a Pascal set.

VBITCNT performs BITCNT on each element of the vector beginning at OP2 and stores the results in the corresponding elements of the vector beginning at OP1.

LBITCNT counts all the "1" bits in all elements of the vector OP2 and stores the resulting total in singleword OP1.

Restrictions: None

Exceptions: None

Precision: For BITCNT, OP1 is a singleword and OP2 has the same precision as the modifier. For VBITCNT, the elements of vector OP1 are singlewords and those of OP2 have the same precision as the modifier. For LBITCNT, OP1 is a singleword and the elements of vector OP2 have the precision specified by the modifier.

The following sets RTA to -1 if RTA has odd parity, 0 otherwise:

```
        BITCNT.S RTA,RTA
        AND.S RTA,#1
        NEG.S RTA
```

The parity of an arbitrarily long block of bits can be obtained by using 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 singlewords (N>2), this code sets RTA (flag-style) if TABLE has odd parity:

```
        XOR.S RTA,<TABLE+4*(N-1)>,<TABLE+4*(N-2)>       ;RTA gets XOR of two words
        MOV.S.S RTB,#[N-4*3]                            ;RTB counts all other words
LOOP:   XOR.S RTA,TABLE[RTB]↑2          ;XOR in next word
        DSKP.GEQ RTB,#0,LOOP            ;loop until all words done
        BITCNT.S RTA,RTB                ;count result as before
        AND.S RTA,#1
        NEG.S RTA
```

A non-zero integral power of two always has a two's-complement representation with exactly one

bit set. Assuming that HUNOZ contains a positive singleword integer, this code jumps to TWOPOWER if HUNOZ is an exact power of two:

```
BITCNT.S RTA,HUNOZ       ;RTA←1 if HUNOZ is a power of two
DJMPZ.EQL RTA,TWOPOWER   ;jump to TWOPOWER if RTA-1 is zero
```

If zero is to be considered a power of two, DJMPZ.EQL can be changed to DJMPZ.LEQ.

BITFST — Bit number of first "1" bit

BITFST . {Q,H,S,D} **XOP**
LBITFST . {H,S,D} **S:=V**

Purpose: For BITFST, if OP2=0 then OP1:=-1 else OP1:=*bit number of the leftmost "1" bit in* OP2. This instruction is useful for finding the index of the first element of a Pascal set.

LBITFST finds the first "1" bit in vector OP2 and puts its number--or, if there are no "1" bits in the vector, a zero--into scalar singleword OP1.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. For BITFST, OP2 has the same precision as the modifier. For LBITFST, each element of OP2 has the same precision as the modifier.

The following sets RTA to floor(log2(RTA)) with RTA assumed to be a non-zero unsigned singleword integer:

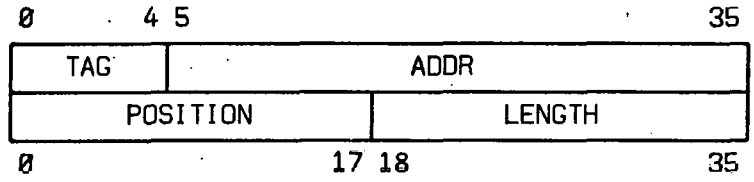
```
BITFST.S RTA,RTA
SUBV.S RTA,#35.
```

This piece of code constructs a byte pointer in (doubleword) RTA to the smallest byte containing all the one-bits in HUNOZ:

```
BITFST.S RTA,HUNOZ       ;number of leading ''0'' bits
BITRV.S RTA1,HUNOZ,#36.  ;reverse HUNOZ into RTA1
BITFST.S RTA1            ;number of trailing ''0'' bits
ADD.S RTA1,RTA           ;number of surrounding ''0'' bits
SUBV.S RTA1,#36.         ;length of smallest containing byte
MOV.H.S RTA1,RTA         ;put position in high halfword of RTA1
MOVP.P.A RTA,HUNOZ       ;make pointer to HUNOZ in RTA
```

2.10 Byte Manipulation

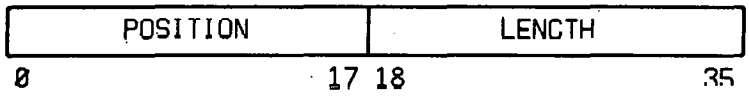
Bytes, byte pointers, and byte selectors: A byte is simply a field of zero or more bits within a singleword or doubleword. The native mode architecture does not tie the concept of a byte to the representation of a character. Instead, it lets the programmer specify the position and width of a byte by constructing a byte pointer:



The TAG and ADDR fields comprise a pointer (as described in Section 1.8.1), and are subject to the validation checking described in Section 1.8.2. They must point to an aligned singleword in memory--that is, ADDR must be a multiple of 4. The POSITION field gives the bit number within the singleword or doubleword at which the byte begins, and must lie within the range 0 . . 35 for singlewords or 0 . . 71 for doublewords. The LENGTH field gives the number of bits within the byte, and must lie within the range 0 . . 36 for singlewords or 0 . . 72 for doublewords. A singleword byte instruction requires each byte operand to lie within an aligned singleword. A doubleword byte instruction requires each byte operand to lie within the doubleword specified by TAG and ADDR.

If the POSITION and OFFSET fields of a byte pointer violate any of those rules, an ILLEGAL_BYTE_PTR hard trap occurs.

Immediate byte instructions use an operand to specify the singleword or doubleword containing a byte, and thus can access a byte within a constant or register as well as in memory. They use a simplified version of the byte pointer, called a *byte selector*, eliminating the TAG and ADDRESS fields:



One useful consequence of the format for byte pointers is the ability to compare them as if they were ordinary doublewords (provided that one knows the tag fields of the pointers match). The comparison will reveal which byte is higher in memory or, if the two bytes begin at the same position of the same word, which byte is longer.

LBYT — Load unsigned byte

LBYT . {S,D} **XOP**

Purpose: The instruction copies the byte specified by byte pointer OP2 and stores it, right justified in a field of zeros, in OP1.

Restrictions: None

Exceptions: None

Precision: OP1 has the precision specified by the modifier. OP2 is a byte pointer. The byte which OP2 points to must obey the length and alignment rules for the precision specified by the modifier.

The following sets RTA to the exponent field of the singleword floating point number X (the exponent field is 9 bits wide and starts at bit 1 of the word):

```
LBYT.S RTA,[TAG+X ? 1,,9. ]
```

LIBYT — Load immediate unsigned byte

LIBYT . {S,D} — TOP

Purpose: The instruction copies from S1 the byte specified by byte selector S2 and stores it, right justified in a field of zeros, in DEST.

Restrictions: None

Exceptions: None

Precision: S1 and DEST have the same precision as the modifier. S2 is a byte selector.

The following sets RTA to the exponent field of the singleword floating point number X (the exponent field is 9 bits long and starts at bit 1 of the word):

```
LIBYT.S RTA,X,#[1,,9.]
```

LSBYT — Load signed byte

LSBYT . {S,D} **XOP**

Purpose: The instruction copies the byte specified by byte pointer OP2, sign-extends it, and stores it in OP1.

Restrictions: None

Exceptions: None

Precision: OP1 has the precision specified by the modifier. OP2 is a byte pointer. The byte specified by OP2 must obey the length and alignment rules for the precision specified by the modifier.

The following uses RTB as a byte pointer, setting RTA to the signed value of the sign and exponent fields of the singleword floating point number X:

```
MOVP.P.A RTB,X          ; Set address part of pointer
MOV.S.S RTB1,#[0,,10.]  ; Set position, length parts
LSBYT.S RTA,RTB
```

LISBYT — Load immediate signed byte

LISBYT . {S,D} TOP

Purpose: The instruction copies from S1 the byte specified by byte selector S2, sign-extends it, and stores it in DEST.

Restrictions: None

Exceptions: None

Precision: S1 and DEST have the same precision specified by the modifier. S2 is a byte selector.

The following sets RTA to the signed value of the sign and exponent fields of the singleword floating point number X. Notice that a short constant can be used, because the position field of the byte selector is zero:

```
LISBYT.S RTA,X,#10.      ; Same as #<0,,10.>
```

DBYT — Deposit byte

DBYT . {S,D} **XOP**

Purpose: The instruction copies the appropriate number of low-order bits from OP2 and stores them in the byte specified by byte pointer OP1.

Restrictions: None

Exceptions: None

Precision: OP1 is a byte pointer. The byte specified by OP1 must obey the length and alignment rules of the precision specified by the modifier. OP2 has the precision specified by the modifier.

The following sets the mantissa of the singleword floating point number X to the twenty-six low order bits of RTA (the mantissa is 26 bits long and begins at bit 10:

```
DBYT.S [TAG+X ? 10.,,26.],RTA
```

DIBYT — Deposit immediate byte

DIBYT . {S,D} TOP

Purpose: The instruction copies the appropriate number of low-order bits from S1 and stores them in the byte within DEST specified by byte selector S2.

Restrictions: None

Exceptions: None

Precision: S1 and DEST have the precision specified by the modifier. S2 is a byte selector.

The following sets the exponent field of the singleword floating point number in RTA to zero. (The exponent field is 9 bits long and begins at bit 1):

```
DIBYT.S RTA,#0,#[1,,9.]
```

ADJBP — Adjust byte pointer

ADJBP . {C,A,Z} TOP

Purpose: This instruction assumes S1 is a byte pointer which points to one byte in a series of packed bytes. It copies that byte pointer from S1, adjusts it to point to an earlier or later byte in the series, and stores the new pointer in DEST. S2 specifies how many bytes forward (or, if S2 is negative, how many bytes backward) to move the pointer.

The modifier specifies one of three different ways to pack bytes with respect to singleword boundaries.

If the modifier is "C", the instruction positions bytes continuously, one after another, splitting a byte across a singleword boundary when necessary. The pointer S1 must specify LENGTH ≤ 72 and (LENGTH + POSITION) ≤ 72.

If the modifier is "A", the instruction positions bytes continuously, except that it will leave bits "unused" if necessary to prevent a byte from being split across a singleword boundary. It maintains the same alignment of bytes (that is, the same pattern of bytes and unused bits) in each singleword. The pointer S1 must specify a byte which does not cross a singleword boundary, and whose length does not exceed 36 bits.

If the modifier is "Z", the instruction positions bytes beginning at the bit-zero (high-order) end of each singleword. No byte ever crosses a singleword boundary, and if 36 is not evenly divisible by the byte length, then the "leftover" bits all appear at the low-order end of the word. It is illegal for the byte pointer S1 to point to a byte which crosses a word boundary or whose length exceeds 36 bits. It may point to a byte whose position within the word suggests that the bytes are not bit-zero aligned; if so, the instruction will impose bit-zero alignment if S2 causes it to point to a different singleword.

Given that ADDRESS, POSITION, and LENGTH are fields of the byte pointer, and DIV and MOD indicate integer division and modulo in the Pascal language sense rather than the S-1 native mode assembly language sense, the algorithms for this instruction are:

ADJBP.C

```
ADDR := ADDR + ((POSITION + S2*LENGTH) DIV 36) * 4;
POSITION := (POSITION + S2*LENGTH) MOD 36;
```

ADJBP.A

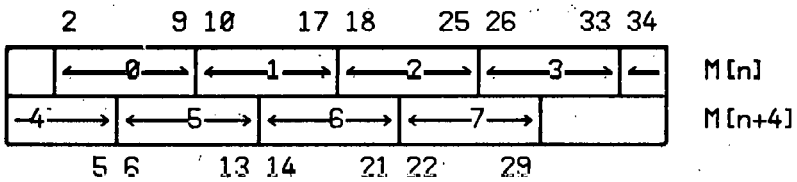
```
BP := POSITION DIV LENGTH;                          (* BYTE NUMBER *)
BPW := BP + ((36-POSITION) DIV LENGTH);             (* BYTES PER WORD *)
ADDR := ADDR + ((S2+BP) DIV BPW) * 4;
POSITION := POSITION + ((S2+BP) MOD BPW) * LENGTH;
```

ADJBP.Z

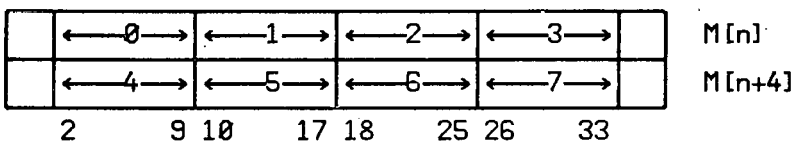
```
BPW := 36 DIV LENGTH;          (* Bytes per word *)
BP := POSITION DIV LENGTH;     (* Byte number *)
IF S2 > 0 THEN
  BF := (36-POSITION-LENGTH) DIV LENGTH  (* Bytes after *)
ELSE BF := BP;                           (* Bytes before *)
IF ABS(S2) > BF
  THEN BEGIN       (* New byte pointer points to different word *)
    ADDRESS := ADDRESS + ( (S2+BPW-BF-1) DIV BPW ) * 4;
    POSITION := ((S2+BPW-BF-1) MOD BPW) * LENGTH;
    END
  ELSE             (* New byte pointer still points to same word *)
    POSITION := POSITION + S2*LENGTH;
```

To show the effect of the three different modifiers, assume that RTA is a byte pointer to an 8-bit byte beginning at bit 2 of singleword M[n]. Executing the instruction "ADJBP.{C,A,Z} RTA,#1" eight times will cause it to point to eight successive bytes in memory, as shown in the drawings:

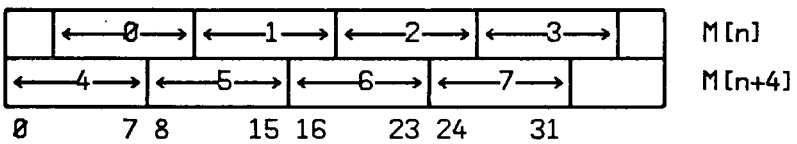
ADJBP.C



ADJBP.A



ADJBP.Z



Restrictions: None

Exceptions: None

Precision: S1 and DEST are byte pointers. S2 is a singleword.

The following advances the byte pointer at BP by one byte:

```
ADJBP.C BP,#1
```

Suppose that TABLE is a vector of NBYTES four-bit bytes, packed nine per singleword. 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 doubleword indexing, because byte pointers are two words long:

```
        BNDSF.0.S RTA,#[NBYTES-1],RTB   ;RTA:=-1 if index okay, else 0
        ADJBP.A RTA,<BPTRS+10>[RTA]↑3,RTB ;get ptr to desired byte, or null ptr
        LBYT.S RTA,RTA                  ;load byte into RTA
        ...

BPTRS:  TABLE ? 0,,4            ;byte pointer to beginning of TABLE
        TABLE ? 0,,0            ;zero-length byte pointer
```

2.11 Stack Manipulation

A stack is specified by any two consecutive singlewords. The architecture interprets these singlewords as a *stack-pointer* and a *stack-limit*. While this pointer/limit pair may reside in memory or in registers, the stack itself always resides in memory. The architecture supports both stacks which grow upward in memory toward higher addresses and stacks which grow downward in memory toward lower addresses. Instructions which manipulate stacks generally specify either "UP" or "DOWN" as a modifier, indicating the direction in which they consider the stack to grow.

For upward-growing stacks, the first of the two consecutive singlewords of the pointer/limit pair is the stack-pointer and the second is the stack-limit. For downward-growing stacks, the first is the limit and the second is the pointer. When an upward-growing stack and a downward-growing stack share the same segment of memory, this allows the same pointer/limit pair to serve both stacks: the pointer of the upward-growing stack is the limit of the downward-growing stack, and vice versa.

For upward-growing stacks, the stack-pointer specifies the next *free* singleword on the stack, so that a push operation first stores the item and then increments the pointer. For downward-growing stacks, the pointer specifies the top item of the stack, so that a push operation first decrements the pointer and then stores the new item.

For upward-growing stacks, the stack-limit points to the first singleword *beyond* the end of the stack. For downward-growing stacks, the stack-limit points to the last singleword into which one may legally store an item.

The processor compares SP with SL using signed 36-bit arithmetic and invokes the STACK_OVERFLOW hard trap on any instruction that would cause the stack to overflow.

Registers %R30 (called SP) and %R31 (called SL) specify a *particular* upward-growing stack for implicit use by interrupts, traps, and linkage instructions such as JSR and ALLOC. The instructions in this section can operate on that stack, but usually they operate on additional stacks specified by other stack pointer/limit pairs.

Note that both the stack pointer and the stack limit are truly pointers, and thus undergo the pointer validation described in section 1.8.2.

ADJSP — Adjust designated stack pointer

ADJSP . {UP,DN} **XOP**

Purpose: Adjust the size of an {upward-growing, downward-growing} stack. The instruction assumes that FIRST(OP1) and SECOND(OP1) are a stack pointer/limit pair, and adjusts the stack pointer to point to operand OP2.

The pointer itself is subject to segment bounds checking during ADJSP. If the instruction would make the stack pointer exceed the stack limit, a STACK_OVERFLOW hard trap will occur.

Restrictions: None

Exceptions: None

Precision: FIRST(OP1), SECOND(OP1) and OP2 are singlewords.

The following throws away the top 4 singleword stack elements of the upward-growing stack designated by the stack pointer/limit pair SPL:

```
ADJSP.UP SPL, (SPL)-4*4
```

PUSH — Push onto designated stack

PUSH . {UP,DN} . {Q,H,S,D} — XOP

Purpose: Push OP2 onto the upward-growing or downward-growing stack designated by stack pointer/limit pair FIRST(OP1) and SECOND(OP1).

If the instruction would cause the stack pointer to pass the stack limit (that is, OP1+{1,2,4,8}>NEXT(OP1) for PUSH.UP or NEXT(OP1)-{1,2,4,8}<OP1 for PUSH.DN) a STACK_OVERFLOW hard trap will occur. Similarly, causing the stack pointer to cross a segment boundary results in an OUT_OF_BOUNDS hard trap.

Restrictions: None

Exceptions: None

Precision: FIRST(OP1) and SECOND(OP1) are singlewords. OP2 has the precision of the modifier.

The following pushes RTA on the stack designated by stack pointer/limit pair SPL:

```
PUSH.UP.S SPL,RTA
```

POP — Pop from designated stack

POP . {UP,DN} . {Q,H,S,D} — XOP

Purpose: From the upward-growing or downward-growing stack designated by pointer/limit pair FIRST(OP2) and SECOND(OP2), pop the top value (whose precision is specified by the second modifier) and store that value in OP1.

A STACK_OVERFLOW hard trap occurs if the instruction would make the stack pointer pass the stack limit, and an OUT_OF_BOUNDS hard trap occurs if it would make the stack pointer cross a segment boundary.

Restrictions: None

Exceptions: None

Precision: FIRST(OP2) and SECOND(OP2) are singlewords; OP1 has the precision of the modifier.

The following pops the top halfword on an upward-growing stack into RTA. Let SPL be the pointer/limit doubleword designating the stack:

```
POP.UP.H RTA,SPL
```

PUSHADR — Push address onto designated stack

PUSHADR . {UP,DN} **XOP**

Purpose: Compute a tagged pointer to OP2 and push that pointer onto an upward-growing or downward-growing stack specified by stack pointer/limit pair FIRST(OP1) and SECOND(OP1).

If the instruction would cause the stack pointer to pass the stack limit (that is, OP1+4>NEXT(OP1) for PUSH.UP or NEXT(OP1)-4<OP1 for PUSH.DN) a STACK_OVERLOW hard trap will occur. Similarly, causing the stack pointer to cross a segment boundary results in an OUT_OF_BOUNDS hard trap.

Restrictions: None

Exceptions: None

Precision: FIRST(OP1) and SECOND(OP1) are singlewords.

The following pushes a pointer to WHIRR onto the stack specified by a pointer at %R25 and a limit at %R26:

```
PUSHADR.UP %R25,WHIRR
```

2.12 Routine Linkage and Traps

These instructions provide call and return mechanisms for subroutines, coroutines, trap handlers, and interrupt handlers. (Additional instructions WTDBP and RTDBP, used to specify the locations for trap and interrupt vectors, appear in Section 2.15.)

The architecture provides several complete sets of call and return instructions with varying degrees of sophistication. They include:

JSR, ALLOC, RETSR, RET

Jump to and return from simple subroutines. JSR calls the subroutine, pushing a single parameter on the stack; ALLOC may be used to save registers and allocate space upon the stack; and RETSR returns from the subroutine, restoring the parameter. Alternatively, RET returns but discards the parameter pushed by JSR and, if desired, a number of words preceding it on the stack.

CALL, JSP, ENTRY, UNCALL

Call and return from an internal procedure, using a stack frame. CALL calls the procedure, ENTRY builds the stack frame, and UNCALL returns from the procedure, dropping back to the preceding stack frame. JSP is useful when the chain of procedure calls permits calls to share a stack frame.

CALLX, ENTRY, RETGATE, UNCALL

Call and return from an external procedure, using a stack frame. CALLX calls the procedure and ENTRY builds the stack frame. If the call crossed a ring boundary, the procedure returns with RETGATE rather than with UNCALL.

TRPSLF, RETUS

Cause a trap to one of the vectors for the current address space, and return from the corresponding trap handler. See Section 1.9.3 for details. RETUS is also used to return from the handler of a soft trap.

TRPEXE, RETFS

Cause a trap to the executive and return from the corresponding trap handler. See Section 1.9.3 for details. RETFS is also used to return from the handler of a hard trap or interrupt.

JCR Jump between coroutines without using the stack.

JMPCALL, JMPRET

These are simple jump instructions which are considered to be call and return instructions for purposes of call tracing.

The following instructions will invoke the CALL_TRAP hard trap when the call tracing mechanism in PROCESSOR_STATUS is enabled:

CALL
CALLX
JCR
JMPCALL
JMPRET
JSP
JSR
RET
RETGATE
RETSR
UNCALL

2.12.1 The Stack Frame Convention

All of the linkage instructions use registers R30 and R31 as stack pointer (SP) and stack limit (SL). The CALL/JSP/ENTRY/UNCALL family of instructions establish a stack frame convention which further defines R28 to be a *closure pointer* (CP), defines R29 to be a *frame pointer* (FP), and defines a stack frame consisting of three singlewords called SF.EP, SF.FLAGS, and SF.RET_ADDR. FP points to SF.EP for the current procedure.

CP The closure pointer points to the stack frame for the procedure which is immediately global to the one which is currently executing. In Pascal, this is the procedure (or main program) inside which the currently executing procedure was declared. This pointer establishes the static scope of a language.

FP The frame pointer points to the stack frame for the currently executing procedure.

Though the stack frame need contain only three singlewords, we'll present a more elaborate example that contains the following:

SF.CP The closure pointer that points to the stack frame of the procedure which statically encloses the current one.

SF.PREV_FP The frame pointer which points to the stack frame of the procedure which called the current one.

SF.EP An entry pointer, which points to the first singleword of code for the current procedure. This permits the placing of debugging and runtime information between the physical beginning of the procedure and the first instruction.

SF.FLAGS A word of flags which is zeroed on entry to the routine.

SF.RET_ADDR The return address, a pointer to an instruction within the current procedure. When the current procedure calls another one, this pointer specifies where to resume execution when the other procedure returns.

To illustrate the stack frame convention, consider the following fragment of a Pascal program:

```
PROCEDURE A;
   VAR A1, A2, A3;
   PROCEDURE C;
      VAR C1, C2, C3;
      BEGIN
      ...
      END (* C *);
   PROCEDURE B;
      VAR B1, B2, B3;
      BEGIN
      C;
      ...
      END (* B *);
   BEGIN
   B;
   ...
   END (* A *);
```

Suppose that someone calls procedure A, which calls procedure B, which in turn calls procedure C. We stop the processor some time after C begins to execute, but before it has called any further procedure. Following the stack frame convention, Figure 2-2 shows the appearance of the stack and the code frame.

The CALL and CALLX instructions save SF.RET_ADDR within the stackframe, and the ENTRY instruction saves SF.EP and clears SF.FLAGS. The remaining portions of the stack frame must be handled by a sequence of individual instructions. In Figure 2-2, for example, the instructions required for procedure B to call procedure C might look like:

Within procedure B:

```
CALL CP,C                              ; Call C, giving it the same
NI: ...                                ; CP as B because both are
                                       ; nested in A. The address NI
                                       ; is saved as SF.RET_ADDR
                                       ; within the stack frame of B
```

At the beginning of procedure C:

```
<information for runtime debugging>
C:ALLOC.2 CP,(SP)4*<3+SizeOfLocals>    ; Push the CP and FP, allocate
```

```
                                        ; 3 SWs for the rest of the
                                        ; stack frame, allocate more for
                                        ; the local variables
ENTRY (SP)-4*<3+SizeOfLocals,C          ; Make SF.EP point to C,
                                        ; clear SF.FLAGS, make FP
                                        ; point to SF.EP

<Code of procedure C>
UNCALL (FP)4*-1,(FP)4*-2                ; Return, retrieving B's FP
                                        ; from our frame and popping
                                        ; our frame from the stack
```

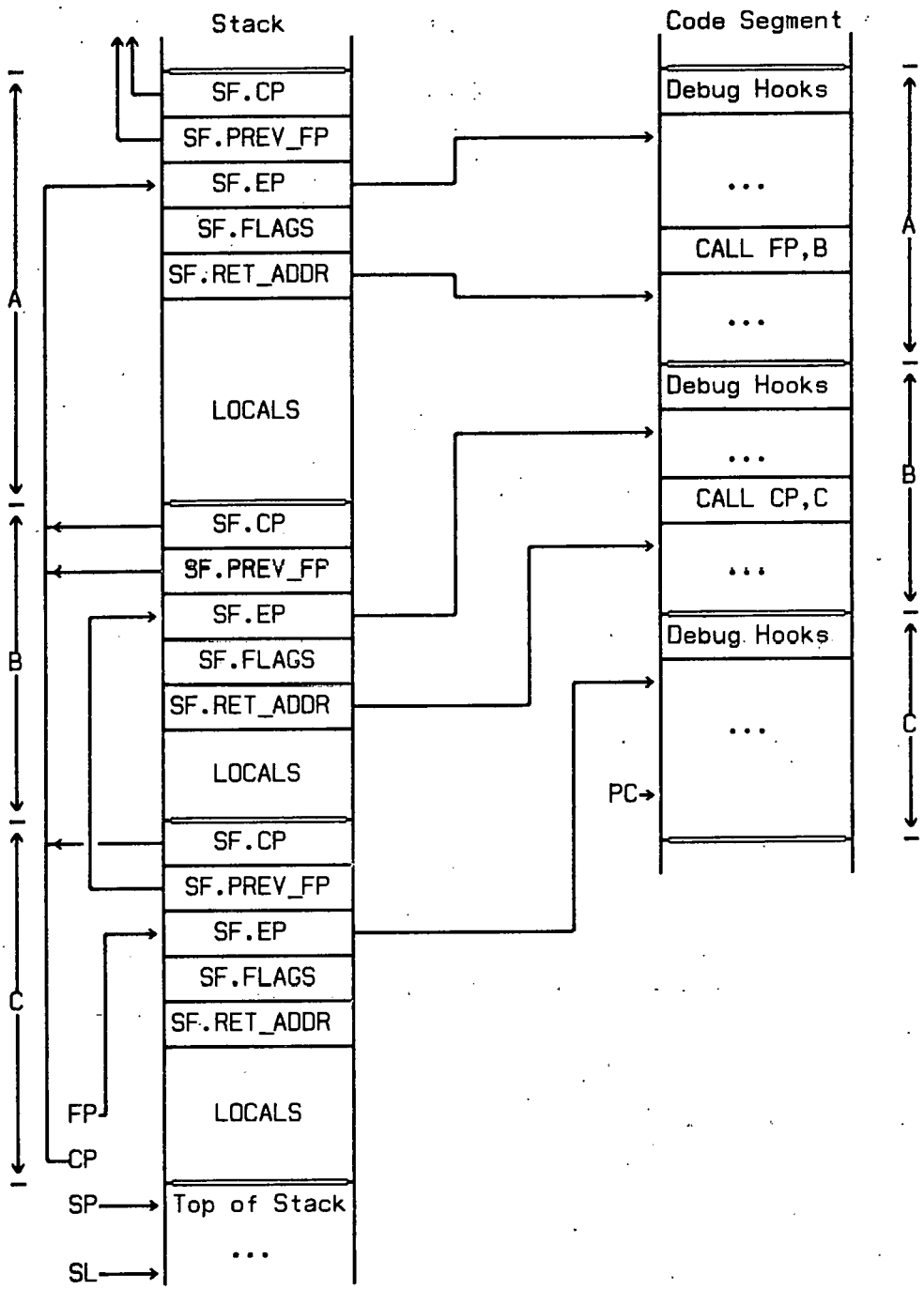


Figure 2-2
Stack Frame Illustration

If procedure C expected parameters, the sequence could easily be changed to use the modifier and OP1 of the ALLOC instruction to push additional registers onto the stack preceding SF.CP and SF.FP:

```
ALLOC.<Parms+2> CP-4*Parms,(SP)4*<3+SizeOfLocals>
```

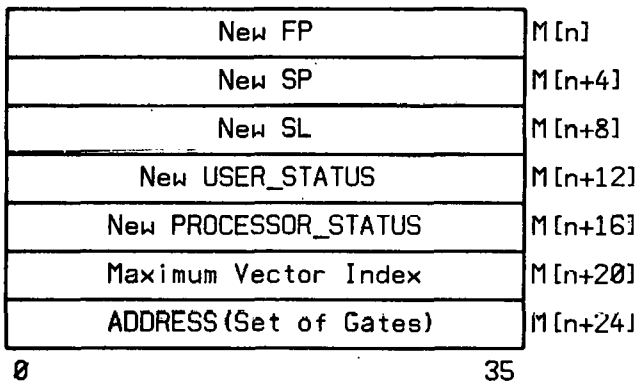
2.12.2 Cross-ring Calls

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

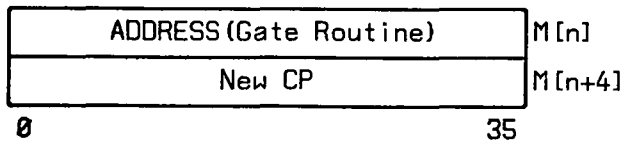
To achieve this, the architecture provides an instruction called CALLX, a special kind of pointer called a *gate pointer*, and a vector of entry points called *gates*. When the CALLX instruction employs a ring or user pointer to specify the called procedure, it cannot--due to the validation mechanism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

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

the ring in question:

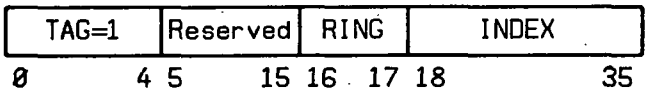


The "set of gates" pointed to by the gate descriptor corresponds to a set of trap vectors. The gates are numbered consecutively beginning at 0, and each has the following format:



A gate routine is actually the routine to be called by the CALLX instruction, but here it roughly corresponds to a trap handler. When the CALLX instruction executes using a gate pointer, the following sequence occurs:

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

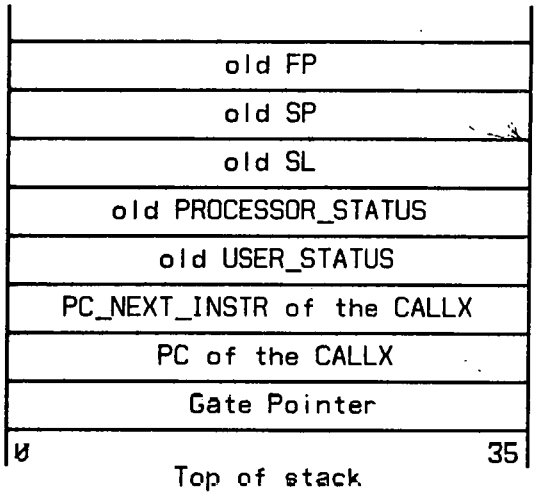


If RING=3, a GATE_INDEX_TOO_BIG hard trap occurs (there are no gates into ring 3 because the address validation mechanism does not prevent any ring from making ordinary calls into ring 3). Otherwise, the processor consults the gate descriptor for the specified ring. If INDEX is greater than the maximum vector index specified by that gate descriptor, a GATE_INDEX_TOO_BIG hard trap occurs. Otherwise, the processor uses INDEX to select the specified gate from the set of gates pointed to by the gate descriptor. Note that the pointer-and-index mechanism for finding the proper gate is subject to address validation.

2. Save FP, SP, SL, PROCESSOR_STATUS, and USER_STATUS internally. Load FP, SP, SL and USER_STATUS with the new values specified in the gate descriptor. If the ring specified by RING is privileged, load PROCESSOR_STATUS with the value specified in the gate descriptor.

3. Push the current state onto the SP stack specified by the new PROCESSOR_STATUS and SP found in the gate descriptor. The act of pushing this information onto the stack is atomic, and any interrupts will remain pending until it is complete. A hard trap can result, however--if, for example, the SP would cross a segment boundary, exceed SL, or touch an absent page--and such a hard trap does intercede (Section 1.9.6).

The information is pushed onto the stack in the following format, known as the *save area* for the gate crossing (if the ring specified by RING is not privileged, push zero in place of PROCESSOR_STATUS):



4. Load CP with the value specified in the gate itself. Load PC with the address of the gate routine specified in the gate itself and resume execution. By thus changing the ring of execution before executing the first instruction of the called routine, the processor effectively bypasses the usual address validation mechanism and the checking of the execute bracket (STE.EB field) of the corresponding segmentito.

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

2.12.3 Routine Linkage Instructions

CALL — Call an internal procedure

CALL — JOP

Purpose: Call an internal procedure, assuming the use of the standard stack frame. First CP:=OP1, then SF.RET_ADDR:=PC_NEXT_INSTR (SF.RET_ADDR is the singleword at (FP)4*2). Then GOTO JUMPDEST, which must lie within the ring of execution.

Restrictions: None

Exceptions: None

Precision: OP1 is a memory address; OP2 is a jump destination.

Suppose a procedure named C is declared within a procedure named B. The following sequence would call C from B:

```
MOVP.P.A %R27,Parmlist          ; Pointer to parameters
CALL FP,FirstC                  ; Call C. Use B's FP as C's
                                ; CP because C is nested
                                ; within B
```

CALLX — Call an external procedure

CALLX — XOP

Purpose: Call an external procedure, assuming the use of the standard stack frame. First CP:=OP1, then SF.RET_ADDR:=PC_NEXT_INSTR (SF.RET_ADDR is the singleword at (FP)4*2). Then fetch OP2 and treat the resulting value as a pointer. If the pointer has a gate tag, perform a cross-ring call through a gate (see Section 2.12.2); otherwise, simply go to the instruction it points to and resume execution there.

If OP2 is a register or constant, an ILLEGAL←OPERAND←MODE or ILLEGAL←CONSTANT hard trap occurs.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords. The contents of OP2 must point to a singleword.

Assume that a procedure has been passed as a parameter to the current routine, and that the two singlewords at (AP)0 are a pointer to the code for that procedure, followed by its closure pointer. To invoke the procedure, the current routine would execute:

```
CALLX (AP)1*4,(AP)0*4
```

JSP — Jump and save PC

JSP JOP

Purpose: First OP1:=PC_NEXT_INSTR, then go to JUMPDEST.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following saves the return address in R0 and calls PRSTR:

```
JSP R0,PRSTR
```

ENTRY **Initialize a stack frame**

ENTRY **XOP**

Purpose: Initialize the stack frame assumed by the CALL and CALLX instructions. OP1:=ADDRESS(OP2), typically used to make SF.EP point to the first instruction of the called routine. NEXT(OP1):=0, typically used to clear SF.FLAGS. FP:=ADDRESS(OP1), typically used to make FP point to SF.EP, marking the location of the stack frame.

Note that ENTRY does not alter SP. It assumes that an instruction such as ALLOC has allocated space for the stack frame.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords.

The following sequence saves the entire register file, with the exception of SP and SL, preceding the portion of the stack frame initialized by ENTRY:

```
ALLOC.30 R0,4*<3+SizeOfLocals>  ; Save registers and allocate space
                                ; for stackframe and locals
ENTRY (SP)-4*<3+SizeOfLocals>,C ; Make SF.EP point to C, clear
                                ; SF.FLAGS, make FP point to SF.EP
```

UNCALL — Return from a call

UNCALL — XOP

Purpose: Return from a procedure called by the CALL or CALLX instruction. FP:=OP1; SP:=ADDRESS(OP2). Go to the instruction pointed to by SF.RET_ADDR. (SF.RET_ADDR is (FP)4*2 *after* OP1 has been moved to FP.)

If the instruction causes SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Use RETGATE, not UNCALL, to return from cross-ring calls.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords.

The following sequence restores the entire register file, with the exception of SP and SL, from the area of the stack preceding SF.EP, pops the stack frame, and returns to the caller:

```
MOVMS.30 R0, (FP)-4*30.          ; Restore registers
UNCALL (FP)-4*1, (FP)-4*30.      ; Restore old FP, pop all
```

RETGATE — Return from a cross-ring call

RETGATE XOP

Purpose: Return from a cross-ring call initiated by the CALLX instruction. OP1 is the first of six singlewords specifying new values for FP, SP, SL, USER_STATUS, PROCESSOR_STATUS, and PC respectively. If the new value of PC specifies a ring whose number is less than that of the ring of execution, a BAD_A_VALIDATION hard trap occurs. Otherwise, the instruction loads the new values into the FP, SP, SL, and USER_STATUS registers. If the ring of execution is privileged, it also loads the new value specified for PROCESSOR_STATUS. In any case, it loads the new value into PC and resumes execution.

Use UNCALL, not RETGATE, to return from ordinary calls.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

A cross-gate call will present the called routine with a stack like the one shown in Section 2.12.2. The callee might use the following instruction to return:

```
RETGATE (SP)-4*8.       ; Restore the caller's state from the
                        ; first 6 singlewords of the 8 singlewords
                        ; pushed by the CALLX instruction.
```

JSR — Jump to subroutine

JSR **JOP**

Purpose: Push first OP1 and then the return address onto the stack whose pointer is SP. Then transfer to JUMPDEST.

If this instruction would cause SP to pass SL, a STACK_OVERFLOW hard trap occurs; if it would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

The following pushes RTA and ADDRESS(F01) on the stack before jumping to BAZ:

```
        JSR RTA,BAZ
F01:    ...             ;return address
```

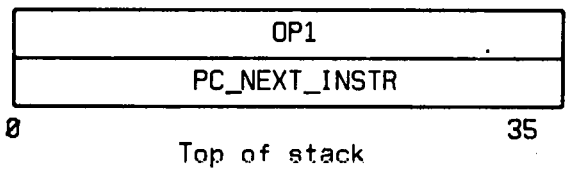


Figure 2-3
JSR Save Area Format

ALLOC — Allocate space atop stack

ALLOC . {1 .. 32} **XOP**

Purpose: This instruction pushes a specified group of singlewords onto the SP stack (the one used by the subroutine calling mechanism) and then adjusts the stack pointer, possibly allocating additional space atop the stack. Typically it is used to save registers and make room for a stack frame.

More specifically, the instruction first moves a vector of 1 . . 32 singlewords starting with OP1 to the vector pointed to by SP (if the two vectors overlap, the result is undefined). Then SP:=ADDRESS(OP2). Thus, OP2 is typically a memory location beyond the last of the words moved, though this is not required. If this instruction would cause SP to pass SL, a STACK_OVERFLOW hard trap occurs; if it would cause SP to cross a segment boundary, an OUT_OF_BOUND3 hard trap occurs.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 must be singlewords.

The following saves all the registers and reserves an additional DW on the stack as well:

```
ALLOC.32 %R0, (SP)<4*(40+2)>
```

Note that the modifier is a *decimal* number, but the numbers in the operands are *octal*. The same instruction could be written:

```
ALLOC.32 %R0, (SP)<4*(32.+2)>
```

RETSR **Return from subroutine**

RETSR **XOP**

Purpose: Return from a subroutine that was invoked by the JSR instruction. First the instruction copies ADDRESS(OP2) into SP. Then it pops the first singleword (return address) from the stack pointed to by SP and stores it in the PC. Then it pops the second singleword (typically the value of OP1 placed there by the JSR instruction) and stores it in OP1.

To be sure that RETSR is the exact reverse of JSR, the programmer must use the same OP1 in both JSR and RETSR, and assure that OP2 in the RETSR instruction is the same memory location that SP pointed to immediately after the JSR. If the subroutine does not alter SP, then OP2 should be "(SP)"; otherwise, the subroutine should save a stack marker and use it as OP2.

If the instruction would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords.

The following code calls BAZ, which returns to F01, saving and restoring RTA on the stack. Assume SP is the stack pointer:

```
        JSR RTA,BAZ
F01:    ...             ;return here

BAZ:    ...             ;called routine
        RETSR RTA,(SP)
```

Suppose that BAZ needs N words of temporary stack space while it is running. These words can be allocated using the ADJSP instruction (or ALLOC if registers must also be saved), and the RETSR instruction can automatically discard these words and pop the JSR save area as well:

```
BAZ:    ALLOC.2 %R8,(SP)<N+2>*4 ;save %R8 and %R9, and allocate N words
        ...                     ;called routine
        MOVMS.2 %R8,(SP)-<N+2>*4        ;restore registers %R8 and %R9
        RETSR RTA,(SP)-<N+2>*4  ;pop stack and return from subroutine
```

RET **Return and pop parameters**

RET **XOP**

Purpose: Return without restoring parameters. First the instruction makes SP point to OP2. Then it pops one singleword (the return address) from the stack pointed to by SP and stores it in the PC. Then it makes SP point to OP1, thereby optionally popping and discarding parameters (such as the one pushed onto the stack by the JSR instruction).

If the instruction would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords.

The following returns from a previous JSR call, throwing away the operand previously pushed on the stack by the JSR:

```
RET (SP)-4,(SP)
```

TRPSLF — Trap to self

TRPSLF . { 0 .. 63 } **XOP**

Purpose: Trap to a routine in the current address space. The operation of TRPSLF is explained in detail in Section 1.9.3; briefly, the modifier selects one of 64 trap vectors. The selected vector itself specifies a handler address and a word called TRP_PARM_DESC_SW. Within TRP_PARM_DESC_SW are two fields called TMODE1 and TMODE2 which can be set to tell the processor to evaluate the operands of the TRPSLF instruction as it would the operands of an ordinary instruction. The processor pushes the evaluated operands onto the SP stack so that the trap handler can access them and operate upon them, providing software emulation of whatever instruction is desired.

Restrictions: None

Exceptions: None

Precision: Determined by TRP_PARM_DESC_SW for each operand

The following causes a trap to the "number 0" trap routine in the current address space, passing to it the operands X and Y:

```
TRPSLF.0 X,Y
```

RETUS — Return, restoring user status

RETUS . {R,A} **XOP**

Purpose: Return from a soft trap or TRPSLF trap. This instruction uses the save area beginning at OP1 to recover the pre-trap state of the processor, and pops the stack by making SP point to OP2. (Thus, OP2 should ordinarily be the value of SP preceding the trap, and OP1 should be the first word of the save area pushed by the trap.)

The instruction loads USER_STATUS with the old USER_STATUS found in the save area. (Section 1.9 illustrates the save area format.)

Ordinarily, RETUS.R repeats the instruction that was in progress when the trap or interrupt occurred (that is, the instruction at the PC stored in the save area) whereas RETUS.A skips to the following instruction.

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

Note that the instruction does not copy REGISTER_SAVE_AREA back into the registers.

If the instruction would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Restrictions: None

Exceptions: None

Precision: Both operands are singlewords.

The following example shows how to use the RETUS.A instruction as a one-word trap handler that ignores the trap and resumes execution at the instruction following the one that caused the trap. The pseudoregister (SP)-4 obtains the old SP from the last singleword of the save area. The operand ((SP)-4)0 thus indicates the singleword pointed to by the old SP. Because the SP stack grows upward, SP always points to the free location atop the stack, and thus in this case it also designates the first word of the save area:

```
RETUS.A ((SP)-4)0,((SP)-4)0
```

TRPEXE — Trap to executive

TRPEXE . { 0 .. 63 } **XOP**

Purpose: Trap to an executive routine. The operation of TRPEXE is explained in detail in Section 1.9.3; briefly, the modifier selects one of 64 trap vectors. The selected vector itself specifies a handler address and a word called TRP_PARM_DESC_SW. Within TRP_PARM_DESC_SW are two fields called TMODE1 and TMODE2 which can be set to tell the processor to evaluate the operands of the TRPSLF instruction as it would the operands of an ordinary instruction. The processor pushes the evaluated operands onto the SP stack so that the trap handler can access them and operate upon them, providing software emulation of whatever instruction is desired.

Restrictions: None

Exceptions: None

Precision: Determined by TRP_PARM_DESC_SW for each operand

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
```

RETFS — Return, restoring full status

RETFS . {R,A} **XOP**

Purpose: Return from a hard trap, interrupt, or TRPEXE trap. This instruction first pops the stack used by the trap handler by making SP point to OP2 and then recovers the pre-trap context of the processor from the save area pointed to by OP1. (Thus, OP2 should ordinarily be the value of SP for the trap handler's stack preceding the trap, and OP1 should be the first word of the save area pushed by the trap. The value of SP for the task interrupted by the trap is assumed to exist unaltered in the register file used by that task.)

To recover the pre-trap context, the instruction loads USER_STATUS and PROCESSOR_STATUS with the old USER_STATUS and the old PROCESSOR_STATUS found in the save area. (Section 1.9 illustrates the save area format.)

Ordinarily, RETFS.R repeats the instruction that was in progress when the trap or interrupt occurred (that is, the instruction at the PC stored in the save area) whereas RETFS.A skips to the following instruction.

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

When the instruction state is non-zero, RETFS.A sets the TRACE_PEND bit to match the TRACE_ENABLE bit in the saved PROCESSOR_STATUS and the CALL_TRACE_PEND bit to match the saved CALL_TRACE_PEND bit, just as the instruction would if it were allowed to finish; thus, aborting an instruction does not erroneously disable tracing.

Note that the instruction does not copy REGISTER_SAVE_AREA back into the registers.

If the instruction would cause SP to cross a segment boundary, an OUT_OF_BOUNDS hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: Both operands are singlewords.

The following shows a trap handler for a hard trap. The pseudoregister ((SP)-4)0 specifies the last word of the save area, which contains the old SP for the trap handler's stack. Because the SP stack grows upward, SP points to the free location atop the stack, so the old SP also points to the first word of the save area pushed onto the stack by the trap:

```
(code to handle the trap without altering SP)
RETFS.R ((SP)-4)0,((SP)-4)0
```

JCR **Jump to coroutine**

JCR **XOP**

Purpose: The instruction first exchanges OP1 (usually register SP) with OP2 (usually a memory location holding a saved copy of the value of SP used by the other coroutine). Then it copies the saved "return address" from NEXT(OP2), stores PC_NEXT_INSTR in NEXT(OP2), and branches to the return address.

Restrictions: None

Exceptions: None

Precision: All operands involved are singlewords.

When each of two coroutines has its own distinct stack, the JCR instruction transfers between them without using either stack. Instead, it stores the stack pointer and program counter for the currently inactive coroutine in two consecutive singlewords pointed to by OP2. In the following example, let SAVE.AREA be the first of those two singlewords. Then the following instruction saves the stack pointer and PC for the current routine, sets up the stack pointer and PC for the other routine, and branches to it.

```
JCR SP,SAVE.AREA        ;call other coroutine
```

JMPCALL, JMPRET — Jump to call/return

JMPCALL	JOP
JMPRET	JOP

Purpose: These instructions are identical with the JMPA instruction, except that JMPCALL is considered to be a call instruction and JMPRET is considered to be a return instruction when call tracing is enabled.

2.13 Interrupts and I/O

See Sections 1.9 and 1.10 for explanations of the interrupt and input/output mechanisms.

The {B,Q,H,S} modifiers that appear on certain instructions refer to bitwise, quarterword, halfword, and singleword translations, which are likewise explained in Section 1.10.

IOR — I/O read

IOR . {Q,H,S,D} — **XOP**
VIOR . {B,Q,H,S} — **V:=V**

Purpose: Transfer from an I/O memory to main memory.

IOR transfers a scalar from OP2 (which must lie on an I/O page) to OP1 (which must lie on a non-I/O page) without translation.

VIOR transfers the vector OP2 (which must lie within an I/O page) to vector OP1 (which must lie within a non-I/O page), translating each singleword according to the modifier.

Restrictions: None

Exceptions: None

Precision: For IOR, OP1 and OP2 have the precision of the modifier. For VIOR, OP1 and OP2 are vectors of aligned singlewords regardless of the modifier, and SIZEREG specifies the number of singlewords in the destination (main memory) vector.

Assume BUFFER is a legitimate IOBUF address. To read eighty characters from the I/O memory (starting at BUFFER) to a block in memory starting at IMAGE, the following instruction sequence could be used:

```
MOV.S.S %SIZEREG,#<80./4>      ;set %SIZEREG to eighty QWs
VIOR.Q IMAGE,BUFFER            ;do read
```

IOW — I/O write

IOW . {Q,H,S,D} **XOP**
VIOW . {B,Q,H,S} **V:=V**

Purpose: Transfer from main memory to an I/O memory.

IOW transfers a scalar from OP2 (which must lie on a non-I/O page) to OP1 (which must lie on an I/O page) without translation.

VIOW transfers the vector OP2 (which must lie within a non-I/O page) to vector OP1 (which must lie within an I/O page), translating each singleword according to the modifier.

Restrictions: None

Exceptions: None

Precision: For IOW, OP1 and OP2 have the precision of the modifier. For VIOW, OP1 and OP2 are vectors of aligned singlewords regardless of the modifier, and SIZEREG specifies the number of singlewords in the source (main memory) vector.

Assume BUFFER lies within an I/O page. To transfer the four characters "S-1!" into the IOBUF starting at BUFFER the following instructions could be used:

```
MOV.S.S %SIZEREG,#<4/4>          ;make vector 4 characters long
VIOW.Q BUFFER,#["S-1!"]          ;do write
```

Because no translation is required, however, the following instruction would work just as well:

```
IOW.S BUFFER,["S-1!"]            ;copy a singleword
```

IORMW — I/O read/modify/write

IORMW — TOP

Purpose: In one memory cycle (and hence indivisibly with respect to other processors in a multiprocessor system) DEST:=S2 and then S2:=S1. (More precisely, because the processor prefetches operands and because TOP instructions store DEST last, this instruction makes a temporary copy of S2, stores S1 in S2, and then stores the copy into DEST.)

DEST and S1 must lie in main memory. S2 must lie on an I/O page.

Restrictions: None

Exceptions: None

Precision: S1, S2, and DEST are all singlewords.

The following illustrates the use of IORMW:

```
IORMW RTA,#-1,LOCK
```

VPIOR, VPIOW — Vector I/O read/write by physical address

VPIOR . {B,Q,H,S} **V:=V**
VPIOW . {B,Q,H,S} **V:=V**

Purpose: VPIOR copies a vector from OP1, which must lie on an I/O page, to the vector in main memory whose physical address is specified by the 34 low order bits of RTA.

VPIOW copies a vector from main memory, beginning at the location whose physical address is specified by the 34 low order bits of RTA, to OP1, which must lie on an I/O page.

Both instructions perform the translation specified by the modifier.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: Regardless of the modifier, OP1 is a singleword and the low order 34 bits of RTA are the physical address of a singleword. SIZEREG specifies the number of singlewords in the vector in main memory.

Copy 4000 singlewords, treated as packed 8-bit characters, from TTYMEM to BUF in main memory:

```
MOVPHY RTA,BUF
MOV.S.S SIZEREG,#4000.
VPIOR TTYMEM
```

INTIOP — Interrupt I/O processor

INTIOP — XOP

Purpose: Interrupt the I/O processor connected to the I/O memory containing OP1, and pass OP2 to the I/O processor as a parameter whose purpose is not specified by the architecture.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 are singlewords. OP1 must lie within an I/O page having WRITE_PERMIT access.

Assume BUFFER lies within an I/O page. The following instruction will interrupt the I/O processor connected to the I/O memory containing BUFFER:

```
INTIOP BUFFER,#0
```

WAIT — Wait for interrupt

WAIT **XOP**

Purpose: Cause the processor to wait for an interrupt.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are unused.

The following instruction waits for an interrupt:

```
WAIT
```

RIEN — Read interrupt enable

RIEN — XOP

Purpose: If interrupts are enabled for the I/O memory containing singleword OP2, then OP1 := -1 else OP1 := 0.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords; OP2 must lie on an I/O page.

The following jumps to DISABLED if interrupts are not enabled for the I/O memory which contains TTYMUX:

```
RIEN RTA,TTYMUX
JMPZ.EQL.S RTA,DISABLED
```

WIEN — Write interrupt enable

WIEN **XOP**

Purpose: If the low order bit of OP2 is "1", enable interrupts for the I/O memory containing OP1; otherwise, disable interrupts for that I/O memory.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords. OP1 must lie on an I/O page.

The following enables all interrupts for the I/O memory containing TTYMUX:

```
WIEN TTYMUX,#1
```

RIPND — Read interrupt pending

RIPND **XOP**

Purpose: OP1 gets the priority level of the pending interrupt for the I/O memory containing OP2. (OP1=0 indicates no interrupt is pending.)

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords. OP2 must lie on an I/O page.

The following sets RTA to the level of pending interrupt for the I/O memory containing TTYMUX:

```
RIPND RTA,TTYMUX
```

WIPND — Write interrupts pending

WIPND — XOP

Purpose: If an interrupt is pending for the I/O memory containing OP1, change the priority of the interrupt to the level specified by OP2. If not, cause an interrupt with priority specified by OP2 on behalf of the I/O memory containing OP1 (whether the interrupt occurs immediately or remains pending depends, as always, on the relative priority of the uniprocessor). If OP2=0, the instruction effectively clears any pending interrupt for the I/O memory in question. If OP2 is not a valid level, an ILLEGAL_PRIORITY hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 and OP2 are both singlewords. OP1 must lie on an I/O page.

The following clears any pending interrupt for the I/O memory containing TTYMUX:

```
WIPND TTYMUX,#0
```

2.14 Cache Handling

The S-1 uniprocessor has four caches: an instruction cache, a data cache, an instruction map cache, and a data map cache. The first two hold recently used words from address spaces, and the latter two hold recently used entries from the virtual-to-physical address mapping tables (described in Section 1.7).

If the uniprocessor accesses memory to fetch an instruction, then that access involves the instruction cache and the instruction map cache. If the access reads or writes a piece of data, then it involves the data cache and the data map cache. If the ACCESS bits for a particular page specify EXECUTE_PERMIT as well as READ_PERMIT or WRITE_PERMIT, then conceivably one could, by alternately reading (or writing) a location and executing it, cause that location to appear in both the instruction cache and the data cache; no problems need result. (In the more likely situation where the ACCESS bits are used to enforce separation of instructions and data, such a situation would not occur.)

In general, the caches employ a least recently used (LRU) algorithm to decide which cache residents to evict to make room for new residents. Not every instruction causes its operands to be regarded as used, however. I/O instructions do not update the LRU status bits for their operands, for example, since the data involved in an I/O operation is unlikely to be accessed repeatedly.

While the caches are usually invisible to software, instructions are provided to sweep them--that is, deliberately update main memory to reflect any changes in cache contents--if this is felt to improve performance. The cache sweeping instructions take ordinary operands which specify memory location on the pages to be swept; the instructions implicitly examine the *addresses* of those operands rather than the operands themselves to determine which pages to sweep.

SWPIC **Sweep instruction cache**

SWPIC . {V,P} **XOP**

Purpose: Sweep the instruction cache by removing a vector of consecutive singleword residents without writing them back to main memory. (Since access to an instruction page prevents writing, the contents of the cache cannot differ from the corresponding portions of main memory.) OP1 is the vector.

The {V,P} modifier tells the processor how to determine which locations are "consecutive". In either case, it first evaluates OP1 as it would for any ordinary memory reference. If the modifier is V, it then sweeps the vector of words whose virtual addresses follow that of OP1. If the modifier is P, it sweeps the vector of words whose physical addresses follow that of OP1.

Restrictions: Physical sweeps are legal only in privileged mode.

Exceptions: None

Precision: OP1 is a vector of singlewords. OP2 is unused.

The following sweeps all instructions from START up to but not including the following instructions:

```
MOV.S.S %SIZEREG,<.-START>      ;specify the length of the vector
SWPIC.V START                   ;sweep cache
```

SWPDC — Sweep data cache

SWPDC . {V,P} . {U,UK} **XOP**

Purpose: Sweep the data cache by writing a vector of consecutive singleword residents back to main memory. If the second modifier is U, merely update main memory; if it is UK, update main memory and then remove the specified residents from the cache ("kill" them). OP1 is the vector.

The {V,P} modifier tells the processor how to determine which locations are "consecutive". In either case, it first evaluates OP1 as it would for any ordinary memory reference. If the modifier is V, it then sweeps the vector of words whose virtual addresses follow that of OP1. If the modifier is P, it sweeps the vector of words whose physical addresses follow that of OP1.

Restrictions: Physical sweeps are legal only in privileged mode.

Exceptions: None

Precision: OP1 is a vector of singlewords. OP2 is unused.

The following updates the first 128 quarterwords in the address space, without removing them from the data cache (i.e., not killing them):

```
MOV.S.S %SIZEREG,#128.  ;specify the vector length
SWPDC.V.U 0             ;sweep cache
```

SWPIM, SWPDM, FLSHIM, FLSHDM — Sweep/flush instruction/data map cache

SWPIM	**XOP**
SWPDM	**XOP**
FLSHIM	**XOP**
FLSHDM	**XOP**

Purpose: Sweep a map cache, removing one resident, or flush a map cache, removing all residents.

SWPIM removes from the instruction map cache the entry for the page containing OP1. SWPDM removes from the data map cache the entry for the page containing OP1.

FLSHIM removes all entries from the instruction map cache. FLSHDM removes all entries from the data map cache.

None of these instructions update main memory.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: For SWPIM and SWPDM, OP1 is a singleword and OP2 is unused. For FLSHIM and FLSHDM, OP1 and OP2 are unused.

The following kills the instruction map entry for the first page in the user's address space:

```
SWPIM 0
```

The following kills the data map entry for the page containing the memory location pointed to by RTA:

```
SWPDM (RTA)
```

2.15 Context (Map, Register Files, and Status Registers)

This section describes a number of instructions which an operating system can use to set up the proper environment for a task. They manipulate the user and processor status registers, the multiple sets of user registers, the mapping system, and the origin of trap, interrupt, and gate vectors. Sections 1.2.3, 1.4, 1.7, and 1.9 explain details of these features of the architecture.

The logical conditions (LCONDs) mentioned in this section are described at the beginning of Section 2.8.

Address Space IDs: In a multiprogramming environment, it is likely that various tasks will alternately use the same virtual address space but different portions of the physical address space--in other words, that the operating system could keep multiple tasks in various regions of physical memory and switch between them by changing the virtual-to-physical address mapping tables. The operating system would have to sweep the map caches before switching from one task to the next to prevent the new task from being affected by mapping information left in the caches by the old one. To obviate this time-consuming process, the operating system can specify via the SWITCH instruction a different code, called an address space ID, for each task. The caching mechanism combines this code with virtual address references made by that task, rendering them unique from virtual address references made by other tasks. Thus, for example, a reference to virtual address 1000 in ring 3 with address space ID 5 is distinct from a reference to virtual address 1000 in ring 3 with address space ID 20; the mapping information for both of these may reside in cache simultaneously and can provide two different address transformations. It is the responsibility of the operating system never to specify the same ID for two different tasks which use the same address space unless it sweeps the map caches between instances of the two tasks.

SWITCH **Switch context**

SWITCH **XOP**

Purpose: OP1 is a vector describing the state of a task to be run. The instruction loads the appropriate internal registers with the information from this vector and resumes execution (restarting an interrupted instruction if INSTRUCTION_STATE so demands.)

The vector contains the following information:

Singleword	Information
0	DSEGP
1	Address space ID for ring 0
2	Address space ID for ring 1
3	Address space ID for ring 2
4	Address space ID for ring 3
5	PROCESSOR_STATUS
6	USER_STATUS
7	PC
8	SIZE of INSTRUCTION_STATE
9...	INSTRUCTION_STATE

Address space IDs are explained in Section 2.15. The DSEGP is explained in Section 1.7.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is the first element of a vector of singlewords. OP2 is unused.

Start executing the task described in the vector beginning at NextTask:

```
SWITCH NextTask
```

WASJMP — Write address space and jump

WASJMP — JOP

Purpose: OP1 is a vector describing a particular mapping of four virtual address spaces onto the physical address space. The instruction loads the DSEGP and address space IDs from this vector, thereby causing the address translation mechanism to adopt this mapping, and resumes execution at JUMPDEST (where JUMPDEST is translated according to the newly established mapping).

The vector contains the following information:

Singleword	Information
0	DSEGP
1	Address space ID for ring 0
2	Address space ID for ring 1
3	Address space ID for ring 2
4	Address space ID for ring 3

Address space IDs are explained in Section 2.15.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is the first element of a vector of singlewords.

Tell the address translation mechanism to use the mapping specified by NewMap, and resume execution at NewProcess:

```
WASJMP NewMap,NewProcess
```

RRFILE **Read register file identity**

RRFILE **XOP**

Purpose: OP1:=PROCESSOR_STATUS.REGISTER_FILE, right justified and padded with zeros. This instruction tells *which* register file is in use.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

Set RTA to the number (in the range 0 .. 15) of the current register file:

```
RRFILE  RTA
```

WRFILE — Write register file identity

WRFILE **XOP**

Purpose: PROCESSOR_STATUS.REGISTER_FILE:=OP1. This instruction *chooses* which register file to use. If OP1 is not within the range 0 . . 15 the consequences are undefined.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

Select register file number 2:

```
WRFILE  #2
```

RREGFILE — Read register file

RREGFILE — XOP

Purpose: OP2 is a singleword specifying a register file. The instruction copies the entire register file into vector OP1, which is 32 singlewords long.

If OP2 is ouside the range 0 . . 15, an ILLEGAL_REGISTER hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a vector of 32 singlewords. OP2 is a singleword.

Push register file 7 onto the stack pointed to by ANSP:

```
ADJSP.UP ANSP, (ANSP)<32.*4>
RREGFILE (ANSP)-4,#7
```

WREGFILE — Write register file

WREGFILE — XOP

Purpose: OP1 is a singleword specifying a register file. The instruction copies vector OP2, which is 32 singlewords long, into that register file.

If OP1 is ouside the range 0 . . 15, an ILLEGAL_REGISTER hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP2 is a vector of 32 singlewords. OP1 is a singleword.

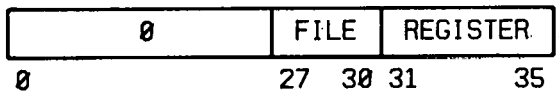
Initialize register file 7 using 32 singlewords popped from the stack pointed to by ANSP:

```
WREGFILE #7,(ANSP)<-32.*4>
ADJSP.UP ANSP,(ANSP)<-32.*4>
```

RREG — Read register

RREG — XOP

Purpose: OP2 is a singleword specifying a register within a particular register file. The instruction copies that register into OP1. The format of OP2 is:



where FILE is in the range 0 . . 15 and REGISTER is in the range 0 . . 31. If OP2 is invalid, an ILLEGAL_REGISTER hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: Both operands are singlewords.

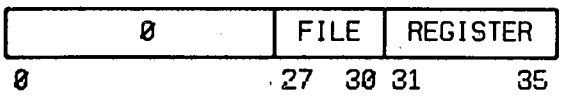
Copy the version of %R4 in register file 7 into the current RTA:

```
RREG RTA,#<32.*7+4>
```

WREG — Write register

WREG — XOP

Purpose: OP1 is a singleword specifying a register within a particular register file. The instruction copies OP2 into that register. OP1 has the following format:



where FILE is in the range 0 . . 15 and REGISTER is in the range 0 . . 31. If OP1 is invalid, an ILLEGAL_REGISTER hard trap occurs.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: Both operands are singlewords.

Copy the current register %R3 into the version of register %R3 in register file 7 (note that this involves register 3, not the PC):

```
WREG #<32.*7+3>,%R3
```

RPS **Read processor status**

RPS **XOP**

Purpose: OP1:=PROCESSOR_STATUS

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following copies PROCESSOR_STATUS into RTA:

```
RPS RTA
```

WFSJMP — Write full status and jump

WFSJMP — JOP

Purpose: USER_STATUS:=FIRST(OP1); PROCESSOR_STATUS:=SECOND(OP1). Note that an ILLEGAL_STATUS hard trap will occur if an illegal value of USER_STATUS or PROCESSOR_STATUS is specified.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: FIRST(OP1) and SECOND(OP1) are singlewords.

The following sets USER_STATUS to FIRST(NEWPST), sets PROCESSOR_STATUS to SECOND(NEWPST) and jumps to BRAZIL:

```
        WFSJMP NEWPST,BRAZIL
```

RUS — Read user status

RUS **XOP**

Purpose: OP1:=USER_STATUS. OP2 is unused.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword.

The following loads RTA from USER_STATUS:

```
RUS RTA
```

JUS — Jump on selected user status bits

JUS . {NON,ALL,ANY,NAL} **JOP**

Precision: If USER_STATUS LCOND OP1 (where LCOND∈{NON,ALL,ANY,NAL}) is true, control is transferred to the location specified by JUMPDEST.

Restrictions: None

Exceptions: None

Precision: All operands concerned are singlewords.

Let ERRORS be a mask for several bits in USER_STATUS. The following jumps to ZIP if any of these bits are set:

```
JUS ERRORS,ZIP
```

JUSCLR **Jump on selected user status bits and clear**

JUSCLR . {NON,ALL,ANY,NAL} **JOP**

Purpose: OP1 is a mask for selecting bits from USER_STATUS. The instruction first tests those bits using the condition specified by the modifier. Then it clears those bits. Finally, if the test yielded true, the processor jumps to JUMPDEST.

Formally:

```
TEMP:=USER_STATUS;
(* ¬ represents one's complement *)
USER_STATUS:=USER_STATUS∧(¬OP1);
If TEMP {NON,ALL,ANY,NAL} OP1 THEN GOTO JUMPDEST;
```

Note that an ILLEGAL_STATUS hard trap will occur if clearing the specified bits would produce an illegal value for USER_STATUS.

Restrictions: None

Exceptions: None

Precision: All operands are singlewords.

Let ZDIV be the mask for the INT_Z_DIV bit in USER_STATUS. The following jumps to YOW and clears this bit if it is set:

```
JUSCLR.ALL ZDIV,YOW
```

WUSJMP — Write user status and jump

WUSJMP — JOP

Purpose: USER_STATUS:=OP1. Control is then transferred to the location specified by JUMPDEST. Note that an ILLEGAL_STATUS hard trap will occur if an illegal value of USER_STATUS is specified.

Restrictions: None

Exceptions: None

Precision: All operands concerned are singlewords.

The following sets the USER_STATUS to NEWUS and jumps to AWAY:

```
WUSJMP NEWUS,AWAY
```

SETUS — Set specified user status bits

SETUS — XOP

Purpose: USER_STATUS:=USER_STATUSvOP1. OP2 is unused. Note that an ILLEGAL_STATUS hard trap will occur if an illegal value of USER_STATUS is specified.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following sets the low order bit in USER_STATUS:

```
SETUS #1
```

CLRUS — Clear specified user status bits

CLRUS **XOP**

Purpose: USER_STATUS:=USER_STATUS∧*one's-complement*(OP1). Note that an ILLEGAL_STATUS 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.

Restrictions: None

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused (OD2 must equal zero).

The following clears the low order bit in USER_STATUS:

```
CLRUS #1
```

RTDBP, WTDBP — Read and write TDBP

RTDBP **XOP**
WTDBP **XOP**

Purpose: These instructions read and write the trap descriptor base pointer, the register which specifies the origin of a table which in turn specifies the origins of each set of trap, interrupt, and gate vectors.

RTDBP loads into OP1 the 34-bit physical address stored in TDBP. WTDBP loads into TDBP the rightmost 34 bits of OP1.

The effect of altering the trap descriptor table without executing a WTBP instruction is undefined.

Restrictions: Illegal in user mode

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused.

The following specifies that the table of trap vector origins begins at the first singleword of physical memory:

```
WTDBP #0
```

2.16 Performance Evaluation

The processor has several *doubleword* counters which can be configured to count different events. A user mode program can read these counters, but only a privileged mode program can write them or alter the bits that enable them. Counter zero is always enabled, by convention, to count real-time cycles.

RCTR — Read counter

RCTR — XOP

Purpose: OP2 is a counter number. OP1 gets the contents of the counter specified by OP2.

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 is a doubleword. OP2 is a singleword.

The following sets RTA (DW) to the current real-time cycle count:

```
RCTR RTA,#0
```

WCTR — Write counter

WCTR XOP

Purpose: OP1 is a counter number. Write OP2 into the counter specified by OP1.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is a doubleword.

The following zeros the real-time cycle counter:

```
WCTR #0,#0
```

RECTR — Read enable bits for counter

RECTR **XOP**

Purpose: OP2 is a counter number. OP1 gets the contents of the enabling register for the counter specified by OP2.

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 is a doubleword. OP2 is a singleword.

The following reads the enabling bits for counter COUNT into RTA:

```
RECTR RTA,COUNT
```

WECTR — Write enable bits for counter

WECTR — XOP

Purpose: OP1 is a counter number. Write OP2 into the enabling register for the counter specified by OP1.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is a doubleword.

The following writes ENABLE into the enabling register for counter COUNT:

```
WECTR COUNT,ENABLE
```

2.17 Miscellaneous

NOP — No operation

NOP — XOP

Purpose: NOP may have operands, but it performs no operation and stores no result. It always transfers control to the next instruction. The operand addressing calculations are carried through; while the operands themselves are not referenced, an invalid addressing mode will cause a RESERVED_ADDRESS_MODE hard trap.

Restrictions: None

Exceptions: None

Precision: OP1 and OP2 may be any precision since they are not fetched.

The following three instructions are, respectively, one, two and three word NOPs:

```
NOP #0,#0
NOP #0,#[0]
NOP #[0],#[0]
```

HALT — Halt this processor

HALT — JOP

Purpose: Halt the processor. Execution continues at JUMPDEST when the halted processor continues. HALT affects only the processor that executes it. OP1 is unused.

Restrictions: Illegal in user mode.

Exceptions: None

Precision: OP1 is unused

The first instruction continues at CONT; the second halts immediately upon continuation:

```
HALT CONT
HALT .
```

RPID — Read processor identification number

RPID **XOP**

Purpose: OP1:=PROC_ID

Restrictions: Traps if the processor is in virtual machine mode.

Exceptions: None

Precision: OP1 is a singleword. OP2 is unused

The following sets RTA to the processor ID number.

```
RPID RTA
```

3 The FASM Assembler

3.1 Commands to invoke FASM

FASM is a cross-assembler which executes on the PDP-10 and emits code for the S-1 native mode instruction set. To use it with the WAITS operating system at Stanford University, type:

```
R FASM;<output>,<listing><-<input>
```

<input> is the name of the file containing assembly source language. The file extension defaults to ".S1" if omitted.

<output> is the file FASM puts relocatable code into. The file extension defaults to ".LDI" if omitted.

<listing> is the file FASM puts its listing into. If you omit the file extension, FASM assumes ".LST".

Alternatively, type the following and FASM will suppress the listing:

```
R FASM;<output><-<input>
```

Or type the following and FASM will suppress the listing, putting relocatable code in a file whose name matches that of <input> but whose extension is ".LDI":

```
R FASM;<input>
```

Or type the following and the program will prompt with "*" and wait for the rest of the command

line.

```
R FASM
```

It is possible to segment the input into several files. To assemble files IN1, IN2 and IN3, for example, type:

```
R FASM;OUT←IN1+IN2+IN3
```

or:

```
R FASM;OUT←IN1.IN2.IN3
```

or create a file called IN containing the line "IN1+IN2+IN3" and then type:

```
R FASM;OUT←@IN
```

A file which, like IN, contains part of the command line is an *indirect file*. Within an indirect file a semicolon tells the program to ignore the rest of the line, including the carriage return and line feed. This allows the command to extend over more than one physical line, as the following example shows:

```
OUT←IN1+;
IN2+;
IN3
```

The first linefeed that is not ignored will cause the indirect file to be closed and command line processing to continue from where the indirect file was called. An indirect file may also call another indirect file (up to 10 levels).

Use the SNAIL commands LOAD and COMPILE to automatically run FASM and then optionally call FSIM. The /L switch may be used with SNAIL to force FASM to make a listing.

3.2 Preliminaries

FASM makes three passes over the input file to do a good (but not perfect) job of substituting relative-JOP instructions for generally bulkier absolute-JOP instructions. During the first pass, FASM uses only absolute jumps, setting each label to the maximum possible value it will attain. During the second pass, FASM replaces absolute jumps with relative ones where possible, provided the jump destination is in instruction space only and not external. During the third pass, FASM generates the code.

FASM accepts the superset of the ASCII character set used at the Stanford Artificial Intelligence Lab (SAIL), but wherever its syntax uses special characters from the SAIL set, it also accepts substitutes from the standard ASCII set. This section will present both choices.

Because each page of S-1 memory can be marked EXECUTE_PERMIT, READ_PERMIT, and/or WRITE_PERMIT, FASM maintains separate location counters controlled by the ISPACE, DSPACE, XSPACE, IPAGE, and DPAGE pseudo-ops explained later.

Like any assembler, FASM processes statements, each of which may define a symbol, emit an S-1 instruction, or emit a dataword.

But unlike many assemblers, which simplemindedly parse lines looking for label, opcode, and operand fields, FASM starts by scanning the text character by character, expanding macros. The resulting strings go to the portion of the assembler that recognizes assembly language constructs. Many of those constructs themselves (symbol definitions, literals, pseudo-ops, and so on) return values just as functions in a high-level language do, so the programmer may embed them in expressions with considerable flexibility.

3.3 Expressions

The primary building block of a FASM statement is the expression. An expression is made up of terms separated by operators with no embedded blanks. The simplest legal expression is a single term with no operators.

Attributes: An expression may have one or more attributes. The possible attributes are: *register*, *instruction value (IVAL)*, *data value (DVAL)*, and *external value (XVAL)*. These attributes are derived from the terms and operators that make up the expression.

A term in an expression may be a number, a symbol, a literal, a text constant or a value-returning pseudo-op.

When it encounters an expression, FASM attempts to perform the indicated operations on the specified terms. Sometimes, the value of a term is not available (for example, is undefined or is external) at the time the expression is evaluated. Sometimes this is permissible and sometimes it will cause an error. In the descriptions that follow it will sometimes be said that an expression must be defined at the time it is evaluated.

3.3.1 Operators

The following are the valid operators along with their precedences. Each is binary unless marked "(unary)".

Purpose	ASCII symbol	SAIL symbol	Precedence
Addition	+	+	1
Subtraction	-	-	1
Multiplication	*	*	2
Division	/	/	2
Bitwise OR	!	∨	3
Bitwise AND	&	∧	3
Bitwise XOR	#	≠	3
Power of 2	^	↑	4
Bitwise NOT		¬	5 (unary)
Plus	+	+	5 (unary)
Minus	-	-,∂	5 (unary)
Register attribute	%	%	5 (unary)

(Though FASM recognizes no ASCII equivalent for "¬", the programmer can achieve the effect of "¬X" by writing "<-1#X>".)

A↑B has the value of A shifted left (if B is positive) or right (if B is negative) by B bits.

The "%" symbol gives the term following it the *register* attribute (though context may override that attribute; for example, a "%5" in an expression inside a constant operand merely contributes an integer "5" to the expression which then becomes a constant.)

Each operator has a precedence which is used to determine order of association. For operations with the same precedence, association is to the left. Angle brackets <> (also known as brokets and pointy brackets) may be used to parenthesize arithmetic and logical expressions. (Parentheses "()" themselves may not be used for this purpose because they are significant for expressing various addressing modes.) A parenthesized (or rather, *broketed*) expression may take more than one line, in which case the value of the last line is used as the value of the expression. However, *all* the lines are evaluated and then all the values are thrown out except for the last one. These evaluations may have side effects like defining symbols, or executing macros, etc.

3.3.2 Numbers

A string of digits is interpreted as a number. If it contains ".", FASM assumes it is decimal. Otherwise, FASM assumes the current radix, which defaults to base 8 (octal) but may be changed with the RADIX pseudo-op. A singleword floating point number has digits on both sides of a decimal point and may be followed by an E, an optional + or –, and a one or two digit exponent, which is assumed to be a decimal number and should not have an explicit decimal point.

3.3.3 Symbols

A symbol is a one- to sixteen-character name made up from letters, numbers, and the characters ".", "_", and "$". (A symbol may actually contain more than sixteen characters, but all characters after the twelfth are ignored.) Lower-case letters are permitted, but are considered to be the same as the equivalent upper-case characters. A symbol must not look like a number; for example, 43. is an integer and 0.1 is a floating point number, whereas 0..1, 1.E5, and 2.3E.5 are symbols (because they do not quite qualify as floating point numbers).

Following the initial character of a symbol, one may enclose in quotation marks any characters which would otherwise be forbidden. The quotation marks and the otherwise forbidden characters all become part of the symbol. For example, the first of the following two lines is an arithmetic expression involving symbols "CAT", "A", and "DOG", whereas the second is a single symbol

"CAT"*A-"DOG":

```
CAT*A-DOG
CAT"*A-"DOG
```

Symbols have values and attributes. The values are 36-bit numbers which are used in place of the symbol when it appears in an expression. The attributes are: *register, instruction value (IVAL), data value (DVAL), half-killed, external value,* and *macro name.*

If a symbol is a macro name, then instead of having a value, the symbol has a macro definition associated with it. This macro definition is expanded when the symbol is seen under certain circumstances and the expansion is used in place of the symbol in the expression. (See the section on macros for more details on macro definition and expansion.)

Predefined symbols: FASM recognizes certain symbols without requiring the programmer to define them.

"."	A lone dot represents the current location counter. It is either an IVAL or a DVAL, depending upon whether ISPACE, DSPACE, IPAGE, or DPAGE is in force. Its value is the quarterword address at which the next instruction or data will be assembled. Its default attribute is IVAL and its initial value is 0 for a relocatable assembly or 10000 octal for an absolute assembly.
RTA,RTB	RTA and RTB represent %16 and %24 respectively, so their attribute is *register.*

3.3.4 Literals

A literal is any set of assembler statements enclosed in [ ] (called square brackets) and separated by "↔", "?", or linefeeds. A literal directs the assembler to assemble the statements appearing inside the square brackets and store them at some location other than the current location counter. If embedded in an expression, the entire literal returns a value: the address at which the first singleword of the literal is assembled. There are certain restrictions on just what may appear inside a literal. Certain pseudo-ops are illegal inside of literals (see the section on pseudo-ops). Currently, labels are not permitted inside a literal, although this may change in the future. The symbol "." is not affected by the fact that it is referenced from inside a literal. It will have the value it had at the point where the literal was begun even though the literal may already have assembled some statements.

Just where the literal is assembled is determined by several factors. First it is determined whether the literal is an instruction-space or a data-space literal. This is determined in the following manner. If the next characters immediately after the [that begins the literal are !I or !D, then the

literal is an instruction-space or data-space literal, respectively. If not, then the literal will be an instruction-space literal if it contains any opcodes. Otherwise it will be a data-space literal. All instruction-space literals will be assembled starting at the current location counter when a LIT pseudo-op is encountered while in instruction-space. A similar statement is true of the data-space literals. Certain other pseudo-ops cause an implicit LIT to be done first.

One typical use of a literal is to move a doubleword from data memory into register space. The following initializes %40 to the largest doubleword integer:

```
        MOV.D.D %40,BIGNUM
        DSPACE
BIGNUM: 377777,,-1
        -1
        ISPACE
        ...
```

but a more elegant way, using a literal, would be:

```
        MOV.D.D %40,[377777,,-1 ? -1]
```

Similarly, the following example uses %40 to index into a table of indirect pointers, perhaps to implement a CASE statement in Pascal:

```
        JMPA CTABL[%40]↑2@
DSPACE
CTABL:  CASE0+TAG
        CASE1+TAG
        CASE2+TAG
ISPACE
```

but a literal expresses the same structure more compactly:

```
JMP <[CASE0+TAG ? CASE1+TAG ? CASE2+TAG]>[%40]↑2@
```

3.3.5 Text Constants

An ASCII text constant is enclosed in double-quotes and has the value of the right-adjusted ASCII characters packed one to a quarterword. For example:

```
"ab"
```

is the same as the number 141142_8. If more than four characters are specified, then only the value of the last four will be used. If the trailing double-quote is missing, the assembler will stop accumulating characters when it sees the end of line. The last four characters will be used in the constant and no error message will be given.

A delimiter such as a space must precede a text constant so FASM does not consider it to be a quoted portion within a symbol.

3.3.6 Value-returning Pseudo-ops

Some pseudo-ops generate values and may be used as terms in an expression. See the descriptions of the individual pseudo-ops to learn what values they return.

3.3.7 Combining terms to make expressions

FASM determines the value of an expression simply by combining the values of the individual terms according to the operators between them.

Determining the attribute of the expression is a bit more complicated, however.

When a symbol with the register attribute appears in an expression, then the entire expression has the register attribute. At most one external symbol may appear in an expression. It does not matter how it appears in the expression; it is assumed to be added in. This causes the expression to be an XVAL. If an IVAL (DVAL) ever appears in an expression then the whole expression is an IVAL (DVAL) with one exception. An IVAL (DVAL) minus an IVAL (DVAL) is no longer an IVAL (DVAL). Note: in a relocatable assembly all relocation is done by *addition* of the I space or D space relocation or of an external symbol's value. Therefore using the negative of an IVAL, DVAL or external value will not have the right effect.

3.4 Statements

A statement can accomplish three things: define a symbol, emit an S-1 instruction, or emit a data word.

How a statement is terminated will depend upon the exact type of statement. In general, a statement is terminated with a linefeed, a ↔, a ?, or a semicolon that begins a comment. (The comment itself terminates at the next linefeed. Some statements, like symbol definitions, can also be terminated with a space or a tab.

3.4.1 Symbol Definition

A symbol may be defined to have a specific value either with the assignment statement or by declaring the symbol to be a label. The assignment statement has two forms:

```
SYMBOL←expression or SYMBOL←←expression
```

An = may be used in place of a ←. These statements define or redefine the symbol to have the value of the expression. The expression must be defined at the time the assignment statement is processed. Any attributes of the expression are passed on to the symbol (except for the *half-killed* attribute). For example, if the expression has a register value, then the symbol is given the register attribute. In addition if the second form is used (with two left-arrows) then the symbol will additionally be given the half-killed attribute. This attribute is not used by the assembler but is passed on to the debugger, where it means that the symbol should not be used in symbolic 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 almost any separator, including space and tab. Therefore it is possible to put more than one assignment statement on one line, or to put an assignment statement on the same line with other statements.

3.4.2 S-1 Instructions

An instruction is a statement that can cause the assembly of one, two or three singlewords. It is made up of an opcode with modifiers followed by a list of operands.

3.4.2.1 Operands

(Throughout the following discussion, either "#" or "?" indicates a constant, and "| ..... |", "⊂ ..... ⊃", and "[.....]" are all equivalent pairs of brackets.)

In general, an operand may be any of the following:

Register or memory reference:

`expression` If the attribute of the expression is "register", FASM interprets it as a quarterword address in the registers; otherwise, FASM interprets it as a quarterword memory reference. If an instruction requires a singleword address, FASM derives it by dividing the value of the specified label or expression by four. If an instruction requires a relative address, FASM derives it by subtracting the current location counter from whatever label or expression the programmer provides.

General constant:

`#expression` If the expression is in the range -32 . . 31 (decimal) the assembler will generate a short constant. If not, it will generate a long, sign-extended constant. (It is dangerous to use an as yet undefined symbol in this expression, because the assembler might decide to switch from one length to the other, confusing the rest of the assembly.)

Pseudoregister:

```
(register expression)expression
```

Long constants:

```
#cexpression⊃
#[expression]
#c!S ↔ expression⊃
#[!S ? expression]
```

Any of these produces an LO constant (even if the number is small enough to fit inside an SO) right justified with sign extended or compressed as necessary.

```
#cexpression ↔ !0⊃
#[expression ? !0]
```

Either of these produces an LO constant which, if the instruction using it calls for a doubleword, is left justified and extended with zeroes. The spaces around the "↔" or "?" are optional.

```
#[!0 ? expression]
#c!0 ↔ expression⊃
```

Either of these operands produces an LO constant which, if the instruction using it calls for a doubleword, is right justified and extended with zeroes. The spaces around the "↔" or "?" are optional.

Indexed constant:

```
#cexpression⊃(register expression)
#cexpression⊃[register expression]
#[expression](register expression)
#[expression][register expression]
```

An indexed constant adds a constant to the contents of a singleword register. The register expression must lie in the range 0 . . 124 and be divisible by 4.

Operand descriptor:

`!expression` Intended primarily for patching, this generates an operand descriptor (OD) that matches the low 12 bits of the result of the expression. FASM does not check to be sure such an OD is legal, and does not generate an extended word even if the OD calls for one.

Long operand variable:

```
(base)offset[index]↑shift
⊂base⊃offset(index)↑shift
base[index]↑shift
base(index)↑shift
```

This is the general syntax for a long operand (LO) variable. The processor computes the address as if by scanning the expression from left to right. It starts with the contents of the memory location or register specified by "base". Then it adds "offset", if any. Finally it takes the contents of the memory location or register specified by "index", shifts it left by the number of bits specified by "shift", and adds it to the base-offset combination to obtain the address of the operand.

If "@" appears after the entire phrase, indicating indirect addressing, the processor interprets the operand as a pointer and uses it to fetch the ultimate operand. If, on the other hand, the "@" appears after the offset, the processor uses the base+offset address to fetch a pointer from memory, and indexes from it.

The LO variable addressing modes have space use the OD for a sort of "nested" short operand (SO) variable, and they fall into three categories based on how they use this SO variable: as the base, as the index, or not at all.

DEFINITION OF TERMS:

SW_REG	<%R0 .. %R31>
LONG_DISP	31-bit signed displacement
LONG_ADDR	31-bit unsigned address
SHORT_DISP	26-bit signed displacement
SHIFT	0 .. 3 bit left shift
SHORT SHIFT	0 or 2 bit left shift
INDEX_REG	<%R3 .. %R31>
SF	-32 .. 31

USING THE SO AS THE BASE:

```
(SW_REG)LONG_DISP
(SW_REG)LONG_DISP@
(SW_REG)SHORT_DISP[SW_REG]↑SHIFT
(SW_REG)SHORT_DISP@[SW_REG]↑SHIFT
(SW_REG)SHORT_DISP[SW_REG]↑SHORT_SHIFT@
```

```
((INDEX_REG)SF)LONG_DISP
((INDEX_REG)SF)LONG_DISP@
((INDEX_REG)SF)SHORT_DISP[SW_REG]↑SHIFT
((INDEX_REG)SF)SHORT_DISP@[SW_REG]↑SHIFT
((INDEX_REG)SF)SHORT_DISP[SW_REG]↑SHORT_SHIFT@
```

USING THE SO AS THE INDEX:

```
LONG_ADDR[SW_REG]↑SHIFT
LONG_ADDR@[SW_REG]↑SHIFT
LONG_ADDR[SW_REG]↑SHORT_SHIFT@
(SW_REG)SHORT_DISP[SW_REG]↑SHIFT
(SW_REG)SHORT_DISP@[SW_REG]↑SHIFT
(SW_REG)SHORT_DISP[SW_REG]↑SHORT_SHIFT@

LONG_ADDR[(INDEX_REG)SF]↑SHIFT
LONG_ADDR@[(INDEX_REG)SF]↑SHIFT
LONG_ADDR[(INDEX_REG)SF]↑SHORT_SHIFT@
(SW_REG)SHORT_DISP[(INDEX_REG)SF]↑SHIFT
(SW_REG)SHORT_DISP@[(INDEX_REG)SF]↑SHIFT
(SW_REG)SHORT_DISP[(INDEX_REG)SF]↑SHORT_SHIFT@
```

NOT USING THE SO:

```
LONG_ADDR
LONG_ADDR@
(SW_REG)SHORT_DISP
(SW_REG)SHORT_DISP@
```

3.4.2.2 Opcodes and Modifiers

An opcode is built out of a base opcode name followed optionally by a "." and an opcode modifier and another "." and another modifier, etc. The modifiers are standard as defined in the opcode files. Numeric modifiers are in decimal *without* a decimal point.

It is also possible to use an already defined symbol as a modifier. For example, if A has been defined by A←%4 then SLR.A assembles the same way as SLR.4 does. Note that an expression may *not* be used in place of a modifier. For example, SLR.4+4 is not permitted in place of SLR.8 . Also note that if there is a conflict between a legal modifier name and a symbolic value, the legal modifier name will win. For example:

```
M1←←1
BNDTRP.M1.S XXX,YYY
```

will NOT be the same as:

```
BNDTRP.1.S XXX,YYY
```

because M1 is a legal modifier for BNDTRP and takes precedence over the lookup of the symbol M1.

Modifiers should not be omitted from instruction opcodes, with one exception: a precision modifier {Q, H, S, D} which is omitted will be assumed to be S. Modifiers should be written in the order defined by the instruction descriptions.

The opcode must be separated from the operand list by spaces or tabs.

3.4.2.3 Instruction Types

There are several basic instruction types: XOPs, TOPs, SOPs JOPs, and HOPs. For the assembler, they differ as to the number and interpretation of operands.

An XOP is (in general) a two-operand instruction. If no operands are given, then the instruction must be one (e.g. WAIT) which requires no operands, and the operand descriptors are set to zero. If exactly one operand is given then, depending upon the specific instruction, either it is used for both operands or the second operand is defaulted to be register zero (%R0). For example,

```
INC COUNT
```

is equivalent to

```
INC COUNT,COUNT.
```

A TOP is a three-operand instruction, where one of the operands is restricted. Operands may be written only in certain combinations indicated by a two-bit field called *T* within the instruction. FASM automatically sets this field based on the operands specified by the programmer. If X and Y represent two operands which are distinct from each other and from RTA and RTB, then there are four possible combinations for the operands, as the following shows:

```
SUB X,X,Y
SUB RTA,X,Y
SUB X,RTA,Y
```

```
SUB RTB,X,Y
```

Other combinations, such as the following, are illegal:

```
ADD X,Y,RTA
```

If the programmer writes only two operands for a TOP, FASM repeats the first.

An SOP is a two-operand instruction with a skip destination. Both of the operands must be present. The skip destination is written as if it were a third operand, and should be an expression which evaluates to the quarterword address of the instruction that is to be skipped to. If the skip destination is missing, then the instruction is assembled so as to skip over the next instruction, however long it is. For example,

```
ISKP.GTR %1,#100,EXIT
```

assembles a conditional skip to the label EXIT. During the last pass of the assembly, the assembler checks to see that the skip is within range. This means that the value of the skip destination operand must be within -8 . . 7 singlewords of the location of the SOP. The difference in this range is assembled into the SKP field of the instruction.

A JOP is a two-operand instruction, the second of which is the jump destination. If only one operand is specified, then which operand it is assumed to be depends upon the exact opcode. Some opcodes expect only one argument, in which case that argument is the jump destination (JMPA, for example). The opcodes JSR and JCR expect one or two operands. If only one is supplied it is assumed to be the jump destination. For other JOPs, if there is only one argument, it is assumed to be OP1 and the jump is assembled to skip over the next instruction (just as for an SOP with an omitted skip destination). The assembler will try its best to assemble the jump with the PR-bit on (using relative addressing). It even takes a whole extra pass through the source file just for this. For example,

```
IJMPZ.NEQ %20,LOOP
```

assembles a jump to location LOOP.

The only *HOP* instruction is SJMP, which expects a single operand, which should be a simple label or expression that evaluates to the quarterword address of the jump destination. FASM subtracts the current location counter from the operand value and divides by 4 to obtain the necessary singleword relative address. While compact and useful for patching, this instruction lacks the flexibility of the unconditional branch JMPA, which can use indexing or indirect addressing.

3.4.2.4 Data Words

An expression standing alone on a line (or, more precisely, an expression which by itself constitutes a statement) causes FASM to emit a singleword containing the value of the expression.

```
-1       ; A singleword with all bits set
%7+347.  ; A singleword containing 354 decimal
NAME*2   ; A singleword containing twice the value
         ;. of the symbol NAME
```

If two expressions appear on either side of ",,", FASM emits a singleword with the left halfword set to the first expression and the right halfword set to the second.

```
30,,7    ;A singleword with 30 in its left
         ; halfword and 7 in its right halfword
```

The following example illustrates a simple use of a literal. Because the literal itself returns the address of the first word it emits, FASM generates four singlewords in all. At the next "LIT" pseudo-op in data space it generates three singlewords containing 1, 2, and 4 respectively. At the current location counter, it generates a singleword containing the value returned by the literal.

```
[   1
    2
    4   ]
```

3.5 Absolute and Relocatable Assemblies

An assembly is either absolute or relocatable. Initially it is assumed that the assembly is relocatable. Certain things in the input file may cause the assembler to try to change its mind if it is not too late. The pseudo-ops ABSOLUTE and RELOCA will force absolute and relocatable respectively. A LOC will force absolute.

In a relocatable assembly, there is one instruction space and one data space. These spaces may be interleaved in the input file (by use of the ISPACE, DSPACE and XSPACE pseudo-ops) but will be separated into two disjoint spaces in the output. The data space will be output immediately after the instruction space and it is up to the linker to further relocate it to begin on a page boundary (or whatever).

Whenever a word is assembled, the attributes of the expressions involved in the assembly of that word are passed on to the word itself. The assembler outputs instructions to the linker to relocate every IVAL by adding to it the starting address of the instruction segment, and similarly for every DVAL and the starting address of the data segment. Notice that this does *not* do the right thing for the *difference* between an IVAL and a DVAL. This is because the assembler does not keep track of whether the relocation should be positive or negative.

In an absolute assembly, no relocation is done. There may be multiple instruction and data spaces. The pseudo-ops IPAGE and DPAGE cause the assembler to move the location counter to a new page boundary and switch to the indicated space. The assembler output will contain multiple spaces which occur in the same order as the IPAGE and DPAGE statements. The LOC pseudo-op may be used to set the value of the location counter to any desired absolute address (with some restrictions). It cannot be used to change spaces.

An IPAGE, DPAGE, or LOC pseudo-op may not be used in a relocatable assembly, and an ISPACE, DSPACE, or XSPACE pseudo-op may not be used in an absolute assembly.

3.6 Pseudo-ops

The following lists all the pseudo-ops in alphabetical order.

If a "." appears in front of the pseudo-op here, then the "." is mandatory; otherwise it is optional.

Certain pseudo-ops require a string of characters, denoted by ⊗ text ⊗. This indicates that FASM regards the first character (other than a blank or tab) following the pseudo-op as the delimiter for the beginning of the string, and looks for a matching character to delimit the end of the string. Thus, for example, the following produce identical strings:

```
ASCII "Now is the time"
ASCII 'Now is the time'
ASCII bNow is the timeb
```

ABSOLUTE
Forces the assembly to be absolute.

.ALSO,< conditionally assembled text > rest of program
.ELSE,< conditionally assembled text > rest of program
These pseudo-ops conditionally assemble the text in brokets depending upon the success or failure of the immediately preceding conditional. There is an assembler internal symbol called .SUCC which is set when a conditional succeeds and is cleared when one fails. .ALSO will succeed if .SUCC is set and .ELSE will succeed if it is clear. If a conditional succeeds, .SUCC is set both at the beginning and at the end of the conditionally assembled text. This enables the inclusion of conditionals within conditionals while using .ALSO or .ELSE following any outer conditional. For example,

```
IFN A-B,<IFIDN <X>,<Y>,< ...>>
.ELSE < ...>
```

Here, the .ELSE tests the success of the IFN A-B independent of whether the IFIDN succeeded or failed.

ASCII ⊗ text ⊗
Assembles *text* as ASCII characters into consecutive quarterwords, padding the last used singleword with zeros. This pseudo-op may cause more than one word to be assembled as long as it is not enclosed in any level of brokets. However, the "value" of this pseudo-op is the value of the last word it would assemble. So if it is used in an expression, the arithmetic applies only to the last word. If it is enclosed in brokets, then all but the last word are thrown away. For example,

```
1+ASCII /ABCDEFG/
```

is the same as

```
ASCII /ABCD/
<ASCII /EFG/>+1
```

but not the same as

```
1+<ASCII /ABCDEFG/>
```

which is the same as

```
1+ASCII /EFG/
```

ASCIIV ⊗ text ⊗
Is the same as ASCII except that macro expansion and expression evaluation are enabled from the beginning of text as in PRINTV. "\", "‘", and "’" may be used as in PRINTV.

ASCIZ ⊗ text ⊗
Same as ASCII except that it guarantees that at least one null character appears at the end of the string.

ASCIZV ⊗ text ⊗
Is the same as ASCIIV except it does ASCIZ.

.AUXO <filename>
Prepares the file <filename> to receive auxiliary output. Auxiliary output can be generated with the AUXPRX and AUXPRV pseudo-ops. The auxiliary output file remains open until the next .AUXO or the end of the assembly is encountered. It is probably most appropriate to do the .AUXO during just one pass of the assembly. This can be done, for example by

```
IF3,<.AUXO MSG.TXT[P,PN]>
```

AUXPRX ⊗ text ⊗
The *text* is output to the auxiliary file. An error message is generated if no auxiliary file is open.

AUXPRV ⊗ text ⊗

Is the same as AUXPRX except that macro expansion and expression evaluation are enabled from the beginning of *text* as in PRINTV. "\", "‘", and "’ may be used as in PRINTV.

BLOCK `expression`

Adds expression*4 to the location counter. That is, the expression is the number of singlewords to reserve. The expression must be defined when the BLOCK pseudo-op is encountered.

BYTE `(s1)b11,b12,b13,... (s2)b21,b22,b23,...`

The BYTE pseudo-op is used to enter bytes of data. The s-arguments indicate the byte size to be used until the next s-argument. The b-arguments are the byte values. An argument may be any defined expression. The BYTE pseudo-op may *not* evaluate to more than one word. The s-values are interpreted in decimal radix. Scanning is terminated by either ⊃ or >, so a BYTE pseudo-op may be used in an operand or in an expression. For example,

```
MOV A,#⊂BYTE (7)15,12⊃
MOV B,[1+<BYTE (7)15,12>]
```

COMMENT `⊗ text ⊗`

The *text* is totally ignored by the assembler.

DEFINE `name argument-list`

This pseudo-op is used to define a macro. See the section on macros for a description.

DPAGE

If the current space is instruction space, it does an implicit LIT, advances the location counter to the next page boundary, and sets the space to data. If the current space is data, it merely advances to the next page boundary. This pseudo-op may not appear inside of a literal or in a relocatable assembly.

DSPACE

This is 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, which may be omitted, is taken to be the starting address. This pseudo-op may not appear inside of a literal. END forces an implicit LIT to be done first for both instruction and data space. The expression must be defined when the END pseudo-op is encountered.

EXTERNAL sym1,sym2,sym3,...

This pseudo-op defines the symbols in the list to be "external" symbols. The symbols in the list must not be defined anywhere in the program. Only one external reference may be made per expression. The value of the external will be ADDED by the linker to the word containing the expression regardless of the operation the expression says to perform on the external symbol.

IF1,<conditionally assembled text> rest of program
IFN1,<conditionally assembled text> rest of program
IF2,<conditionally assembled text> rest of program
IFN2,<conditionally assembled text> rest of program
IF3,<conditionally assembled text> rest of program
IFN3,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the assembler is in pass 1, 2 or 3 for IF1, IF2 and IF3 or if the assembler is not in pass 1, 2 or 3 for IFN1, IFN2, IFN3.

IFDEF symbol,<conditionally assembled text> rest of program
IFNDEF symbol,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the symbol is defined or not for IFDEF and IFNDEF respectively.

IFE expr,<conditionally assembled text> rest of program
IFN expr,<conditionally assembled text> rest of program
IFL expr,<conditionally assembled text> rest of program
IFG expr,<conditionally assembled text> rest of program
IFLE expr,<conditionally assembled text> rest of program
IFGE expr,<conditionally assembled text> rest of program

Assembles *conditionally assembled text* if the condition is met. If the condition is not met, then the program is assembled as if the text from the beginning of the pseudo op to the matching > were not

present. For IFE the condition is "the expression has value zero," for IFN it is "the expression has non-zero value," etc. In any case the expression must not use any undefined or external symbols. The comma, < and > must be present but are "eaten" by the conditional assembly statement. In deciding which is the matching right broket, all brokets are counted, including those in comments, text and those used for parentheses in arithmetic expressions. Therefore one must be very careful about the use of brokets when also using conditional assembly. For example, the following example avoids a potential broket problem:

```
IFN SCANLSS,<
        SKP.NEQ A,#"<"          ;> MATCHING BROKET
        JMPA FOUNDLESS
>;END OF IFN SCANLSS
```

The broket in the comment is used to match the one in double quotes so that the conditional assembly brokets will match.

IFIDN <string1>,<string2>,<conditionally assembled text> rest of program
IFDIF <string1>,<string2>,<conditionally assembled text> rest of program
These are text comparing conditionals. The strings that are compared are separated by commas and optionally enclosed in brokets. If the strings are identical (different for IFDIF) then the text inside the last set of brokets is assembled as for arithmetic conditionals.

IFB <string>,<conditionally assembled text> rest of program
IFNB <string>,<conditionally assembled text> rest of program
These text testing conditionals compare the one string against the null string. They are equivalent to

```
IFIDN <string>,<>,< ... > ...
IFDIF <string>,<>,< ... > ...
```

.INSERT <filename>
Starts assembling text from the new file <filename>. When the end of file is reached in the new file, input is resumed from the previous file. .INSERTs may be nested up to a level of 10.

INTERNAL sym1,sym2,sym3,...
Defines each symbol in the list as an "internal" symbol. This makes the value of the symbol available to other programs loaded separately from the one in which this statement appears.

IPAGE

If the current space is data space, it does an implicit LIT, advances the location counter to the next page boundary and sets the space to instructions. If the current space is instructions, it merely advances to the next page boundary. This pseudo-op may not appear inside of a literal or in a relocatable assembly.

ISPACE

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.

.LENGTH ⊗ text ⊗

Has the value of the length of the string *text*. A CRLF counts as one character.

LIST

Increments listing counter. Listing is enabled when the count is positive. The count is set to one at the beginning of each pass. XLIST is used to decrement the count.

LIT

Forces all literals in the current space (instruction or data) that have not yet been emitted to be assembled starting at the current location counter. It has no effect on the literals in the "other" space. This pseudo-op may not appear inside of a literal.

LOC expression

Sets the location counter to the specified quarterword address. May not appear inside of a literal or in a relocatable assembly.

MLIST

Increments macro listing counter. Macro expansion listing is enabled when the count is positive. The count is set to one at the beginning of each pass. XMLIST is used to decrement the count.

PRINTV ⊗ text ⊗

Prints *text* on the console. It is identical to PRINTX except that macro expansion may occur within the text. '\, ', and ' may be used within the text as in macro arguments and expression evaluation. See the section on special processing in macro arguments for an explanation of \ and '' processing. Macro expansion is intially enabled at the beginning of text and may be disabled with \.

PRINTX ⊗ text ⊗

Prints *text* on the console.

.QUOTE ⊗ text ⊗

Legal only inside a macro definition. It allows the assembler to see *text* without scanning it for a DEFINE or a TERMIN.

RADIX expression

Sets the current radix to expression. The radix may not be set less than two.

RELOCA

Forces the assembly to be relocatable.

REPEAT expression,<body>

Assembles *body* concatenated with a carriage return *expression* many times. The expression must be defined at the time the REPEAT pseudo op is encountered. The expression must be non-negative. If it is zero, the body will not be assembled.

TERMIN

This pseudo-op is legal only during a macro definition. It is used to terminate a macro definition. See the section on macros for a description.

TITLE `name other_text`

Sets the title of the program to *name*. Everything else on the line is ignored.

XLIST

Decrements listing counter. Listing is enabled when the count is positive. The count is set to one at the beginning of each pass. LIST is used to increment the count.

XMLIST

Decrements macro listing counter. Macro expansion listing is enabled when the count is positive. The count is set to one at the beginning of each pass. MLIST is used to increment the count.

XSPACE

Has the effect of ISPACE if the current space is data and DSPACE if the current space is instructions. This pseudo-op may not appear inside or a literal or in an absolute assembly.

3.7 Macros

The FASM macro facility shows a strong resemblance to those of FAIL (the macro assembler for the PDP-10 developed and used at the Stanford Artificial Intelligence Laboratory) and MIDAS (the macro assembler for the PDP-10 developed and used at the M.I.T. Artificial Intelligence Laboratory), which are hereby acknowledged.

Macros are essentially procedures that can be invoked by name at almost any point in the assembly. They can be used for abbreviating repetitive tasks or for moving quantities of information from one part of the assembly to another (in fact even from one pass to another). Macro operation is divided into two parts: definition and expansion.

The macro facility does differ in an important way from those of other assemblers, however. Macro expansion in FASM is performed at the "read-next-character" level, whereas in most other assemblers it is done at symbol lookup time during expression evaluation. Due to this difference, macro expansion in FASM inherently produces "string" output rather than evaluated expressions as is sometimes the case in other assemblers. Wherever a macro call is seen, the effect can be predicted by substituting the body of the called macro in place of the call.

3.7.1 Macro Definition

Macros are defined using the DEFINE pseudo-op, which has the following format:

```
DEFINE macroname argumentlist
  body of macro definition
TERMIN
```

This will define the symbol *macroname* to be a macro whose body consists of all the characters starting after the CRLF that ends *argumentlist* and ending with the character immediately preceding the TERMIN.

3.7.1.1 The Parameter List

Basically, the parameter list is a list of formal parameters for the macro. This is similar to the list of formal parameters for a procedure in a "high" level language. The parameters are symbol names and are separated by commas. The number of macro parameters must be in the range 0 . . 64. The macro parameter list is terminated by either a ; (which begins a comment, as usual) or a CRLF.

Each macro parameter has certain attributes associated with it. In FASM these attributes are *balancedness*, *gensymmedness*, and *parenthesizedness*. From now on, it shall be said that a parameter is or is not *balanced*, is or is not *gensymmed*, and that certain pairs of parentheses can or cannot *parenthesize* a parameter. If a parameter is not balanced or gensymmed then it is said to be *normal*.

Parameter attributes are specified by enclosing a string of characters in double quotes preceding a parameter in the parameter list. The attributes specified by that string are "sticky"; that is, they apply to all following parameters until the next such string is specified. The characters B and G may appear in the string to indicate that the parameter is to be balanced or gensymmed respectively. There are four parenthesis pairs: (and), [and], < and >, and { and }. Any of these characters may appear in the string to indicate that that set of parentheses may be used to parenthesize that parameter. One final thing that may appear in the string is a statement about the *concatenation* character for the macro body. If the string !=⊗ appears, where ⊗ is any character other than CRLF, then ⊗ will be the concatenation character. If the string 0! appears, then there will be no concatenation character. Only the last statement made in the parameter list about the concatenation character will apply to the macro body.

At the beginning of the parameter list, the attributes have the following defaults: ! is the concatenation character, parameters are neither balanced nor gensymmed, and any pair of parentheses may be used to parenthesize a parameter. Whenever an attribute string is encountered, the previous set of attributes are forgotten and the new one applies to future parameters until the next string is specified.

Here are some examples of valid macro definition lines:

```
DEFINE MAC
DEFINE MAC1 A,B,C
DEFINE MAC2 "!='" A,B, "G" C
DEFINE MAC3 "([B])" A, "[0!" B
```

With these definitions, MAC has no parameters and has ! for the concatenation character. MAC1 has three normal parameters A, B and C with ! for the concatenation character. MAC2 has two normal parameters A and B and a gensymmed parameter C, and uses ' as the concatenation character. MAC3 has a balanced parameter A, for which () and [] can be used as parentheses, and a normal parameter B, for which [] can be used as parentheses. MAC3 has no concatenation character.

3.7.1.2 The Macro Body

The macro body begins at the character following the CRLF at the end of the DEFINE line and ends with the last character before the matching TERMIN. Within the macro body, FASM replaces

all delimited occurrences of formal parameters with a mark that indicates where the actual argument should be substituted. Any character that is not a symbol constituent is considered a delimiter for this purpose. The concatenation character is also considered a delimiter. However, the concatenation character is deleted wherever it occurs and will not appear in the macro body definition. The concatenation character is useful to delimit a formal parameter where, without the concatenation character, the formal parameter would not have been recognized as such. For example,

```
DEFINE MAC A,B,C
PUSH.UP.S SP,B
PUSH.UP.S SP,C
JSR A!RTN
TERMIN
```

If the arguments X, Y, and Z were substituted for the formal parameters A, B, and C, then the third line would assemble as JSR XRTN. Without the concatenation character, it would always assemble as JSR ARTN regardless of the actual value of the parameter A.

In addition to scanning for formal parameters in the macro body, FASM also scans for occurrences of the names DEFINE and TERMIN. It keeps a count of how many it has seen so that it can find the TERMIN that matches the DEFINE that began the macro definition. This allows a macro body to contain a macro definition entirely within it. For example,

```
DEFINE MAC1 A
DEFINE MAC!A
....
TERMIN
TERMIN
```

defines a macro called MAC1 which contains a complete macro definition sequence within itself.

Note that FASM does *not* recognize either comments or text constants as special cases in its search for DEFINEs, TERMINs and formal parameters. Therefore, the user must be careful when using the words DEFINE and TERMIN in those places. They *will* be counted in order to find the TERMIN that marks the end of the current definition. There is a pseudo-op called .QUOTE that can be used if it is desired to inhibit FASM from seeing a DEFINE, TERMIN, or macro parameter name. .QUOTE is like an ASCIZ statement in syntax, taking the first nonblank character after the .QUOTE as a delimiter and passing all characters up to the matching delimiter through to the macro definition. For example,

```
DEFINE MAC
....                ;how to put a .QUOTE /DEFINE/ in a comment
TERMIN
```

will define MAC's body to be

```
....                    ;how to put a DEFINE in a comment
```

3.7.2 Macro Calls

A macro call occurs whenever a macro name is recognized in a context where macro calls are permitted. When this happens, the macro call is processed in two distinct phases. The first is argument scanning and the second is macro body expansion.

3.7.2.1 Argument Scanning

Argument scanning is the process of assigning text strings to the formal parameters of a macro. These text strings come from the input stream. If a formal parameter is not assigned a string by the call, then it is assigned the null string as its value, unless the argument is defined to be gensymmed. In that case, the argument is assigned a six character string beginning with G and followed by 5 decimal digits which represent the value of an internal counter which is incremented before being converted to a text string.

Argument scanning is performed for those macros that have formal parameters. If a macro does not have any formal parameters, then the character that terminates the macro name is left to be reprocessed after the macro expansion is complete, even if it is a comma.

If the macro has formal parameters, then how the argument scan is done depends on the character immediately following the macro name. If it is a CRLF, then the argument scan is terminated and all of the formal parameters are assigned the null string or are gensymmed as appropriate. The CRLF is left to be reprocessed after the macro expansion is complete.

If the character following the macro name is a space or a tab, then all immediately following spaces and tabs are thrown out. The entire sequence of spaces and tabs can be considered to be the macro name delimiter.

If the character following the macro name is a (, then the macro call is said to be a parenthesized call; otherwise it is a normal call. A parenthesized call differs from a normal call in the way argument scanning is terminated. In a normal call, argument scanning is terminated by either CRLF (or its surrogates, ? and ↔), semicolon, or the argument terminator for the last argument (which may be a comma). If terminated by a CRLF or semicolon, the terminator is left to be

reprocessed after macro expansion is complete. In a parenthesized call, only the matching) can terminate the call. The) is not reprocessed after the macro expansion is complete. The following paragraphs will describe the syntax of macro arguments and explain how they are terminated. The phrase "... macro call terminator" refers to the character that terminated either the normal or parenthesized call, as described in this paragraph.

3.7.2.2 Macro Argument Syntax

The first macro argument begins with the first character following either the (that demarks a parenthesized call or the macro name delimiter in a normal call. This character is looked at by FASM to determine how to scan the argument.

If the first character is a left parenthesizing character that belongs to the set of characters that may be used to parenthesize the argument that is being scanned (as determined by the character string in force at the time this formal parameter was seen in the macro define line), then the argument is taken to be all characters following that open parenthesis until, but not including, the matching closed parenthesis. *Any* characters may appear between the parentheses. Only the particular type of parentheses that enclose the argument are counted in finding the matching closed parenthesis. This type of argument is called a parenthesized argument.

If the first character is a comma, then the argument is the null string; the comma is taken to be an argument separator.

If the first character is a macro call terminator, then this argument and all further arguments are not assigned strings. That is, if the arguments are gensymmed, they will be assigned unique gensymmed strings, and if they are not gensymmed they will be assigned the null string.

If the first character is not one of the above, then argument scanning depends on whether the argument is to be balanced or not. If the argument is not to be balanced, then the argument is taken to be all characters from the first character until, but not including, a comma, CRLF (or ↔ or ?), semicolon, or the macro call terminator. If the argument terminator is a comma, it is thrown out; a macro call terminator, however, will be kept to terminate the macro call. If the argument terminator is not a comma, then it is usually a macro call terminator. However, if the call is parenthesized, a CRLF or semicolon will terminate the argument but not the macro call. In this case the remainder of the line (if the terminator was a semicolon) is ignored and the CRLF is thrown out. Argument scanning continues on the next line. This allows the arguments of a parenthesized call to take multiple lines; each CRLF acts as if it were a comma (with comments thrown out) allowing the next line to continue supplying arguments.

If the argument is to be balanced, then all types of parentheses are treated the same. A count is kept of the parenthesis level. If there are no unbalanced parentheses, then a comma or macro call terminator will terminate the argument as if it were a normal argument. Also, if the parentheses are

balanced, any close parenthesis will terminate the argument and the call. If it is a parenthesized call, the close parenthesis must be a) or an error is reported. If it is not a parenthesized call, the parenthesis will be left to be reprocessed after the macro call is complete. In either case, the remaining formal parameters are assigned the null string or gensymmed as appropriate.

3.7.2.3 Special Processing in Macro Arguments

Ordinarily, macro arguments are the quoted forms of the strings that appear between delimiters within the macro call. However, it is possible to call a macro or even evaluate an expression *within* a macro argument *during* the macro argument scan.

If a macro argument is not parenthesized, then the appearance of the character \ (backslash) in the argument will enable macro calls to be recognized during the scanning of the macro argument. The appearance of a second \ will again disable this feature. If a macro call is detected during this time, then that new macro is expanded and its expansion appears as if it were written in line in the macro argument that is currently being read. Every time a new macro call is seen and macro argument scanning is started, the macro-in-argument recognition feature is disabled until re-enabled by a \. The \ character itself is discarded.

Perhaps this will be clearer if explained in terms of the actual implementation. FASM maintains a flag, called the \ flag, which when set enables macro expansion. This flag is pushed when a macro name is recognized and initialized to be off at the beginning of the argument scan. It is complemented every time a \ is seen in the input. When the entire macro call has been scanned (but expansion has not yet started) the \ flag is popped.

In fact, the \ flag has wider application than just in macro calls. It is also applicable at expression evaluation time. Normally it is set during expression evaluation, thereby allowing macros to be expanded. It is perfectly legal to use \ during expression evaluation to inhibit macro expansion.

There is a second feature, analogous to the \ feature, which allows the expression evaluator to be called during a macro argument, or in fact even at expression evaluation time. If an expression is enclosed within "`" and "'" characters, the expression evaluator is called upon to produce a value, which may possibly be null, which is then converted into a character string of digits representing that value in the current radix. The conversion always treats the value as a 36-bit unsigned integer. A null value is converted to the null string. The surrounding singlequotes act in a similar way to parentheses in arithmetic expressions, in that multiple lines may be used, but only the expression on the last line is converted. This converted string is used in place of the singlequoted expression. As in the case of \ this can occur in non-parenthesized macro arguments or in expression evaluation. The singlequote characters themselves are thrown out.

Following are some examples of the use of these features:

```
X←←1  F00'X':   JMPA F001
```

will assemble as

```
F001:   JMPA F001
```

If F00 was a macro name, it would have been expanded in the previous example. This could be inhibited with:

```
\F00\'X':       JMPA F001
```

Next consider:

```
X←←1
DEFINE MAC
X←←X+1
X!TERMIN

F00'MAC':
```

will define the label F002 while incrementing X to be 2. The next time F00'MAC': appears, the label F003: will be generated.

It is sometimes useful to extract the value of a symbol in a macro argument before the macro call changes that value:

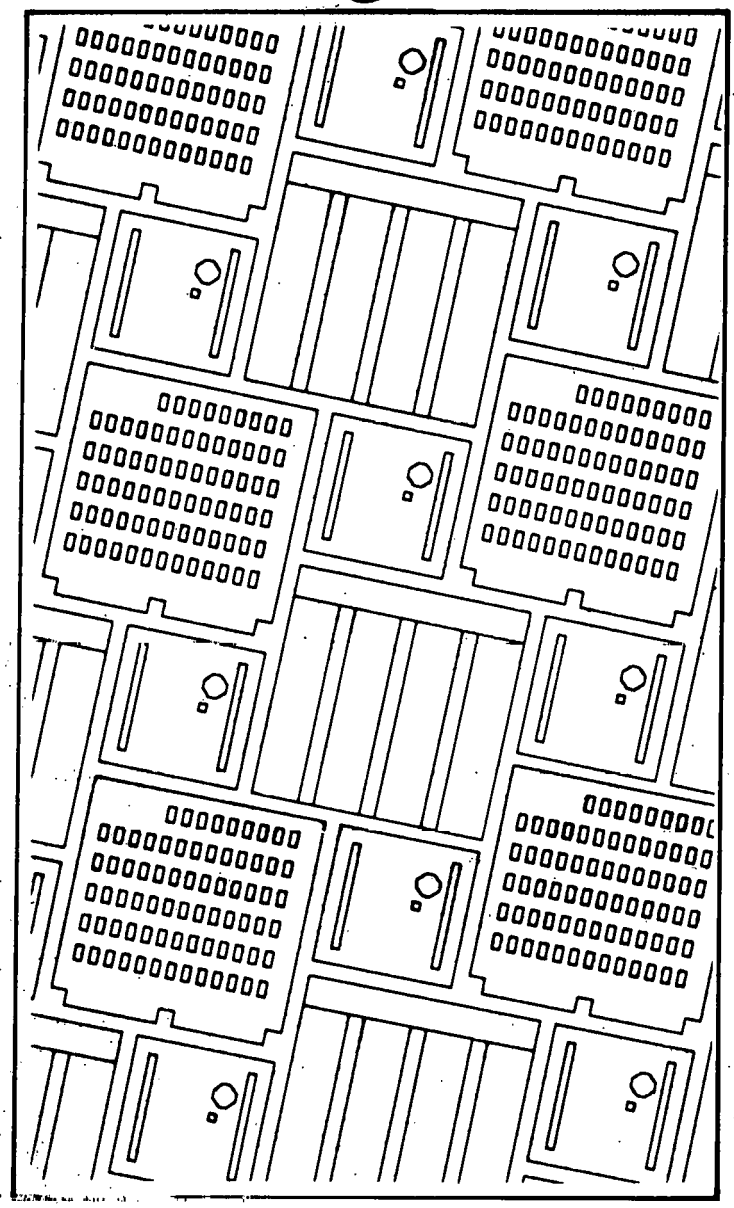
```
DEFINE MAC A
BRR←←BRR+1
A*BRR
TERMIN

MAC 'BRR'
```

will call MAC with the current value of BRR. Without the singlequotes, the string BRR would be passed to the macro and used where "A" appears, which is after BRR is incremented.

Index

5



UYK-7 Emulation (UYK7-1)

Richard Kovalcik

1 A new approach to emulation

Given the task of emulating an existing machine such as the AN/UYK-7 on a microprogrammable processor such as the S-1 Mark IIA, the conventional approach is to write microcode to perform each instruction in the UYK-7 repertoire. Though possible, this approach presents significant drawbacks:

- The underlying microarchitecture used to implement the S-1 native mode instruction set may change from one implementation to the next, even though the native mode instruction set remains exactly the same. Such a change would render a microcoded UYK-7 emulation obsolete;

- Writing and debugging microcode is widely appreciated to be more difficult, time-consuming and unreliable than programming in a high-level language or assembly language, and the result is harder to read, understand or modify.

To avoid these drawbacks, the S-1 Mark IIA emulation of the UYK-7 instruction set uses S-1 native mode binary machine code instead of microcode. A *"post-compiler"*—a novel type of software construct written in a high level language with some assembly language subroutines—transforms UYK-7 binary machine code into S-1 binary machine code, typically while also expressing a loader function. This transubstantiated UYK-7 code then executes in a special runtime environment which provides a high-level emulation of UYK-7 input/output and protection hardware.

Though the post-compiler typically emits several S-1 instructions while transubstantiating each UYK-7 instruction, the resulting emulation will execute a factor of two to four faster than the standard UYK-7 implementation, because it does make optimum use of the pipelined hardware in the S-1 processor.

Upon first encountering the post-compiler concept, one might protest that a machine-coded "object" emulation would necessarily execute more slowly than its microcoded counterpart, and probably much more so. In the case of a pipelined processor such as the S-1 system, this belief is largely fallacious. The relevant measure of the throughput of an emulator is the number of cycles needed

for it to execute a given block of emulated code, *not* the number of instructions to be fetched in the process.

An S-1 processor such as the Mark IIA must execute various sequences of operations for some number of microcycles in order to emulate scalar instructions such as those of the UYK-7; S-1 native mode almost always specifies such sequences as efficiently as possible. Furthermore, as long as an S-1 processor's "parsing" of the emulation specification generated by the post-compiler does not require more than this former number of cycles, there will be no penalty for using S-1 native mode as the input language, as the parsing and execution processes proceed simultaneously in different portions of the processor. S-1 processors are so designed that this is generally true; operand calculation and operation execution are more cycle-intensive than are the instruction-fetching and -decoding processes.

Of course, it is not always true that gains cannot be made through use of microcode. For slightly enhanced post-compiler performance, one may, for instance, 'fine-tune' by very selectively microcoding certain frequently-used processes corresponding to executing particular UYK-7 instructions or calculating certain UYK-7 operands. The frequent use of an extensive sequence of S-1 native mode instructions might also tempt one to substitute for convenience and enhanced readability of the resulting code a single new S-1 instruction, whose microcoded expression might execute slightly faster than the instruction sequence it replaced.

The magnitude of the principal drawback of the post-compiler approach to the emulation problem depends on how similar the emulated and emulating architectures are. The biggest single complication occurs if the machine to be emulated permits code to modify itself, as the UYK-7 indeed does. In such cases, the emulation process must be able to detect and 'repair' the transubstantiated UYK-7 code as each code-modifying instruction is executed; the S-1 Mark IIA post-compiler for the UYK-7 therefore does so.

Other problems arise from the use of one's complement arithmetic in the UYK-7, rather than the two's complement arithmetic used in the S-1, in executing a UYK-7 instruction which causes the next instruction to repeat, and from a UYK-7 instruction which executes a single instruction located at a remote address. The S-1 Mark IIA post-compiling emulator package for the UYK-7 successfully addresses each of these complications.

2 Implementation of the compiler

2.1 Details of implementation

The Mark IIA post-compiler for the UYK-7 is written in Pascal, supplemented with small S-1 assembly language routines for bit manipulation and for running the S-1 code generated by the post-compiler. The basic functions of its major modules are discussed below.

A series of tables is used to correlate UYK-7 code with S-1 code. One table contains the starting address of the S-1 code corresponding to every UYK-7 half-word. If a UYK-7 half-word does not constitute an instruction because the preceding instruction occupies a full word, then a meaningless starting address is stored. Another table contains the length of the S-1 code sequence corresponding to each UYK-7 half-word. Again, if a UYK-7 half-word does not constitute an instruction, then a meaningless length is stored.

To satisfactorily address the problem of self-modifying code, the post-compiler stores in memory a "marked" copy of the original UYK-7 program, where each 32 bit UYK-7 word appears in the 32 low order bits of a 36 bit S-1 singleword whose high order ("pure") bit is set. When the UYK-7 program tries to modify itself, it will store a 32 bit value which is zero-extended to 36 bits, thus clearing the "pure" bit. The next time that instruction is to be executed, the sequence of S-1 code which emulates that instruction discovers the pure bit is not set, and calls the post-compiler to dynamically recompile the instruction and replace that code sequence with an updated one before the emulation proceeds.

If, when the post-compiler is called to recompile a UYK-7 instruction which has been modified, the new S-1 code sequence does not fit in the area occupied by the outmoded one, a patch is generated. This is done by placing the new code sequence in a special patch area of the S-1 processor's memory, followed by a jump back to the start of the S-1 code sequence representing the next

UYK-7 instruction. A jump to the start of the patch is inserted in place of the outmoded code sequence, and the two tables mentioned above modified to contain the new starting address and length of the S-1 code sequence. This approach vigorously exploits the fact that even minimal S-1 memory units have several dozen times the storage capacity of the maximum memory complement of a UYK-7.

The UYK-7 has three register sets of eight registers each: accumulators, index registers, and base registers. UYK-7 index register zero always contains zero. Because the UYK-7 uses a 16 bit one's complement end-around carry adder for indexing, the S-1 indexing modes cannot be used to do UYK-7 indexing. In the UYK-7 architecture, the high order half of a doubleword is contained in the higher numbered word. (e.g., if accumulators 1 and 2 form a doubleword, accumulator 2 has the high order half.) This is backwards relative to the S-1 convention, so to make doubleword manipulation easier, the UYK-7 accumulators are stored in descending order in S-1 processors. Since the top bits of the UYK-7 base registers are very seldom used, only the bottom 16 bits of them are stored in S-1 registers, with the top bits being stored in main memory. Based on all these constraints, the 32 S-1 registers are allocated as follows:

S-1 Register	Purpose
0 . . 2	UYK-7 registers S5 through S7
3	UYK-7 PC
4	S-1 RTA
5	S-1 RTA1
6	S-1 RTB
7	S-1 RTB1
8	Temporary
9 . . 16	UYK-7 registers A7 through A0
17	Temporary
18 . . 24	UYK-7 registers B1 through B7
25 . . 29	UYK-7 registers S0 through S4
30	S-1 stack pointer
31	S-1 stack limit

The post-compiler does not presently attempt to compile all UYK-7 instructions into S-1 native mode sequences. In general, it currently handles only CPU emulation, leaving other areas of the architecture (e.g., interrupt handling, input/output, and protection) for higher level emulation, to be implemented as determined by Navy interest in such. This is consonant with Navy policy regarding the NECS version of the UYK-7, which current specifications state is to emulate only the UYK-7 CPU and is to have a different I/O architecture.

2.2 Problems posed by the UYK-7 architecture

Unfortunately, the UYK-7 architecture is not a simple one. Several features of it require special attention to ensure the emulation preserves the meaning of the original UYK-7 program.

As mentioned earlier, perhaps the biggest problem faced by the post-compiler is that of self-modifying UYK-7 code. Should the UYK-7 program try to modify itself once it has been compiled into an S-1 program, it will modify the UYK-7 copy and not the S-1 copy. The emulation must propagate this change into the S-1 copy. The approach that has been implemented to handle this is the following. As stated previously, UYK-7 words are stored right justified and zero extended in S-1 memory. When a UYK-7 instruction is compiled into S-1 code, the high order bit of the S-1 memory word holding that UYK-7 instruction is turned on. The S-1 code generated tests this bit (by trapping if the word is negative) before actually executing the code for the UYK-7 instruction. If the bit is no longer set, the trap reinvokes the post-compiler to recompile the code that has been modified.

Two additional problems are posed by the UYK-7 repeat instruction (RP), which executes the instruction following it a certain number of times (specified by index register 7) or until a certain condition is met (specified by the "a" field in the instruction word). Some instructions are specified in the architecture as not being repeatable. The first problem is that it is not clearly specified anywhere what happens when a repeat instruction attempts to repeat an "unrepeatable" instruction. The *AN/UYK-7 Technical Description* states on page 31, "If an attempt is made to repeat such an instruction [one which cannot be repeated], the repeat mode may clear with the repeated instruction executed once, or the repeat mode may go to completion with unreliable results from the repeated instruction." This is too vague for the post-compiler to implement. Instead, whenever the post-compiler finds a repeat instruction followed by an instruction which should not be repeated, the post-compiler prints a warning and generates code as if the repeat instruction were not present.

The second problem with the repeat instruction is that it is possible to jump to or execute remotely the instruction immediately following a repeat instruction. This means that the code generated for a repeat instruction must consist of the code to repeat the following instruction (in case the flow of control proceeds normally through the code), followed by a jump instruction, followed by the code to execute the following instruction by itself (in case a jump is made to it or it is executed remotely). In the normal case, a jump will be made around this second sequence of code.

Yet another problem is posed by the execute remote instructions ("XR" and "XRL"). These are handled by use of the two tables mentioned earlier. Once the execute remote instruction has determined the address of the instruction it is to execute, it looks up the address in S-1 memory of the start of the S-1 code to execute the instruction using the first table and the length of the S-1 code using the second table. It then copies the S-1 code into a temporary area, places a jump back to the start of the next instruction to be executed after the copied code, and jumps to the copied

code. This works well even when the object of an execute instruction is another execute instruction, but not, unfortunately, when the object is a repeat instruction. When the object is a repeat instruction, the UYK-7 repeats the instruction following the XR or XRL, not the instruction following the RP. In this case the UYK-7 post-compiler must be reinvoked at run time to compile code for the repeat instruction.

Finally, special sequences of instructions are generated to perform one's complement end-around carry arithmetic on the two's complement architecture of the S-1.

2.3 Preparing input for the post-compiler

The post-compiler operates on files with the extension ".BO8". These files contain UYK-7 core images in octal represented as strings of ASCII digits. ASCII files are used because the current S-1 implementation of Pascal cannot read binary files. These files have one octal number per line. Any line may have a comment consisting of non-numeric ASCII characters before or after the number. The files consist of eight numbers followed by up to 8 blocks of data. The first eight numbers specify into which UYK-7 S register to load the starting address of the corresponding block. If not all eight blocks are present, the corresponding S register numbers at the beginning are ignored. The format of a block is:

Starting address (1 word)
N-1, where N is the number of data words in the block (1 word)
Data (N words)
Checksum (1 word)

These ".BO8" files can be generated in one of two ways. On the SAIL computer system at Stanford University, programs called TD.FAI[UYK,S1] and TD8.FAI[UYK,S1] read 556 and 800 bpi, 7 track UYK-7 boot tapes respectively, creating a binary file called "UYK.BOO". (".BOO" is the suffix for standard UYK-7 boot files.) These files can be converted to ".BO8" files by a program called BOOBO8.PAS[UYK,S1], which leaves eight commented but otherwise blank lines at the beginning of each file on which the S register numbers should be inserted.

The second way to create ".BO8" files is to use the macro facility of the FASM assembler to translate UYK-7 assembly language into the corresponding octal numbers represented as strings of ASCII digits. The file UYKMAC.S1[UYK,S1] provides suitable macros, documentation, and examples.

3 Improving performance

If desired, the performance of the emulation could be improved further without abandoning the basic approach or the work done so far.

3.1 Improvements to the post-compiler

There are several areas in which the post-compiler itself could be improved.

If the post-compiler were to optimize out the addition of the base registers at post-compile time assuming they stayed constant, the speed of the code generated would increase because fewer S-1 operations would be needed for each UYK-7 instruction.

If self-modification occurs often in the program being compiled and each new code sequence requires more space than the previous sequence, then much S-1 memory will be wasted because the post-compiler places the new sequence in a patch area. It would be useful to have some way of reclaiming the S-1 memory used for the sequences that have been replaced.

Since S-1 Pascal does not currently pack records, a full word is necessary for every piece of data mentioned above. This means that there is an overhead of four S-1 words per UYK-7 word, not counting the S-1 code generated. It is hoped that in the future, this can be improved through the use of packing.

Finally, additional research can be done to make the code generated for the repeat instruction fail in the same way that the repeat instruction fails on the real UYK-7 given various illegal instructions.

3.2 Additions to the Mark IIA instruction set

This is a proposed list of modifications to the S-1 Mark IIA architecture which would be especially useful for the post-compiler. In almost all cases, these changes would increase the code density; in some cases, the speed of the code generated would also increase because the microcode implementation of some of the new instructions would require fewer cycles than the macrocode that the post-compiler would have generated.

Add the following instructions.

{UYKADD, UYKSUB, UYKSUBV} . {S, D}

Class: TOP — One's complement arithmetic
Side Effects: CARRY, INT_OVFL

Perform one's complement 32 or 64 bit end-around borrow arithmetic (addition, subtraction, and TOP reverse form subtraction. This is the type of arithmetic that the UYK-7 uses. These instructions would be needed in both single and double word precisions. The single word precision version would operate on 32 bit quantities stored right justified in S-1 singlewords. The high order four bits of the inputs need not be zero but the high order four bits of the result will be zero. CARRY and INT_OVFL would be computed assuming 32 bit precision. The doubleword precision version would operate on 64 bit quantities stored right justified in S-1 doublewords. The high order eight bits of the first word in the double word pair comprising each input need not be zero but the high order eight bits of the first word of the doubleword pair comprising the result will be zero. CARRY and INT_OVFL are computed assuming 64 bit precision.

UYKPACK, UYKPACKV

Class: XOP — Pack a UYK-7 word

Form into the double word OP1 a quantity consisting of eight high order zero bits, followed by the low order 32 bits from {OP2, next(OP2)}, followed by the low order 32 bits from {next(OP2), OP2}. Note that the high order four bits of OP2 and next(OP2) need not be zero. Note that in this case the "V" does not indicate a TOP reverse form and is not quite consistent, but it is not clear what else to call this instruction. These instructions are useful to convert UYK-7 doubleword data into a form that the S-1 can handle better. UYKPACKV is provided because the two singlewords may be stored in either order.

UYKUNPACK, UYKUNPACKV

Class: XOP — Unpack UYK-7 words

Place bits 8-39 of OP2 in bits 4-31 of {OP1, next(OP1)} and bits 40-71 of OP2 in bits 4-31 of {next(OP1), OP1}. Note that bits 0-7 of OP2 need not be zero but bits 0-3 of OP1 and next(OP1) will be zero. Once again, in this case the "V" does not indicate a TOP reverse form. These instructions are useful as the inverse of the UYKPACK and UYKPACKV instructions.

{UYKSHF, UYKSHFV} . {LF, RT} . {S, D}

Class: TOP — Logically shift a UYK-7 word

Read the {32,64} low order bits from the {singleword, doubleword} S1, logically shift them {left, right} by the amount specified by singleword S2, and deposit the result in the low order bits of DEST. UYKSHFV is the reverse form. Note that bits {0-3, 0-7} need not be zero in S1 but will be zero in DEST.

{UYKSHFA, UYKSHFAV . {S, D} . {LF, RT}

Class: TOP — Arithmetically shift a UYK-7 word

Read the {32,64} low order bits of the {singleword, doubleword} S1, arithmetically shift them {left, right} by the amount specified by the singleword S2, and deposit the result in the low order bits of DEST. UYKSHFAV is the reverse form. Note that bits {0-3, 0-7} need not be zero in S1 but will be zero in DEST.

{UYKROT, UYKROTV} . {S, D} . {LF, RT}

Class: TOP — Rotate a UYK-7 word

Read the {32, 64} low order bits from the {singleword, doubleword} S1, rotate them {left, right} by the amount specified by the singleword S2, and deposit the result in the low order bits of DEST. UYKROTV is the reverse form. Note that bits {0-3, 0-7} need not be zero in S1 but will be zero in DEST.

UYKINDEX

Class: TOP — Perform UYK-7 indexing arithmetic

Store in DEST the sum of S1 and S2 using 16 bit one's complement end-around carry addition. No overflow detection is done. Note that bits 0-19 of S1 and S2 need not be zero but bits 0-19 of DEST will be zero.

UYKBASE

Class: TOP — Perform UYK-7 base register arithmetic

Store in DEST the sum of S1 and S2 using 18 bit addition without carry or overflow. Note that bits 0-17 of S1 and S2 need not be zero but bits 0-17 of DEST will be zero. This instruction is roughly equivalent to

```
ADD.H result+2,op1,op2
MOV.H.H result,#0       ; clear the top half
```

UYKMOV

Class: XOP — 32 Bit Move

Store in OP1 bits 4-31 from OP2, right justified and zero extended. This operation can be accomplished with the AND instruction but the UYKMOV instruction can be faster--since it is not necessary to fetch the extended word consisting of the mask--and allows greater choice over the destination than AND (a TOP format instruction) would. Notice that "UYKMOV A,A" clears just bits 0-3 of A. This is useful when a UYK-7 half-word or quarter-word is stored with a DIBYT, but it is necessary to clear the sign bit modification flag.

4 Status of the emulation project

The post-compiler has been written and is in the final stages of testing.

Runtime support for the post-compiler, which will provide interrupt handling, input/output, and protection, will be written later.

U.S.GPO:1980-689-102/5502

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy $;Microfiche $3.50

Page Range	Domestic Price	Page Range	Domestic Price
001-025	$ 5.00	326-250	$18.00
026-050	6.00	351-375	19.00
051-075	7.00	376-400	20.00
076-100	8.00	401-425	21.00
101-125	9.00	426-450	22.00
126-150	10.00	451-475	23.00
151-175	11.00	476-500	24.00
176-200	12.00	501-525	25.00
201-225	13.00	526-550	26.00
226-250	14.00	551-575	27.00
251-275	15.00	576-600	28.00
276-300	16.00	601-up [1]	
301-325	17.00		

[1] Add 2.00 for each additional 25 page increment from 601 pages up.

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under Contract W-7405-Eng-48.

Technical Information Department · Lawrence Livermore Laboratory
University of California · Livermore, California 94550