BiiN™

CPU ARCHITECTURE REFERENCE MANUAL

Order Code: 6AM9000-4AB00-0BA2

LIMITED DISTRIBUTION MANUAL

This manual is for customers who receive preliminary versions of this product. It may contain material subject to change.

BiiN™
2111 NE 25th Ave.
Hillsboro, OR 97124

REV.	REVISION HISTORY	DATE
-001	Preliminary Edition	7/88

BiiN™ MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

BiiN™ assumes no responsibility for any errors that may appear in this document. BiiN™ makes no commitment to update nor to keep current the information contained in this document.

BiiN™ retains the right to make changes to these specifications at any time, without notice.

The following are trademarks of BiiN™: BiiN, BiiN/OS, BiiN/UX, BiiN Series 20, BiiN Series 40, BiiN Series 60, BiiN Series 80.

Apple, Macintosh, and MacTerminal are trademarks of Apple Computer, Inc. UNIX is a trademark of AT&T Bell Laboratories. Torx is a trademark of Camcar Screw and Mfg. Ada is a certification mark of the Department of Defense, Ada Joint Program Office. DEC, VT102, and VAX are trademarks of Digital Equipment Corporation. Smartmodem is a trademark of Hayes Corporation. IBM is a trademark of International Business Machines, Inc. MULTIBUS is a registered trademark of Intel Corporation. Microsoft is a registered trademark of Microsoft Corporation. Mirror is a registered trademark of SoftKlone Distributing Corporation. WYSE is a registered trademark of Wyse Technology. WY-60 and WY-50 are trademarks of Wyse Technology.

Additional copies of this or any other BiiN™ manuals are available from:

BiiN™ Corporate Literature Dept.
2111 NE 25th Ave.
Hillsboro, OR 97124

CONTENTS

Chapter 1. Guide to This Manual

Chapter 2. Architecture Overview

Chapter 3. Data Types and Addressing Modes

Chapter 4. Instruction Set Summary

Chapter 5. Floating-Point Operation

Chapter 6. Execution Environment

Chapter 7. Protection Model

Chapter 8. Object Addressing

Chapter 9. Type Management and Access Control

Chapter 10. Faults

Chapter 11. Debugging and Tracing Support

Chapter 12. Interrupts

Chapter 13. Introduction to Processes, Processors & Synchronization

Chapter 14. Interprocess Communication and Synchronization

Chapter 15. Process Management

Chapter 16. Processor Management and Interrupts

Chapter 17. Instruction Formats and Operand Addressing

Chapter 18. Instruction Reference

Appendix A. Instruction and Data Structure Quick Reference

Appendix B. Considerations for Writing Portable Software

List of Figures

List of Tables

GUIDE TO THIS MANUAL 1

This chapter describes this manual. It explains the organization of the manual, describes the contents of each chapter, and discusses terminology used in the manual. It also shows the chapters of the manual that should be of most interest to applications programmers, compiler designers, and operating-system designers.

1.1 Manual Structure

This manual is a reference manual for the BiiN™ processor. It gives programmers and system designers detailed information about the processor's programming environment and operating-system support facilities.

1.2 Chapter Overview

The following is a brief overview of the contents of each chapter:

Chapter 1 — Guide to This Manual. Overview of this manual.

Chapter 2 — Architecture Overview. Overview of the architecture implemented by this processor.

Chapter 3 — Data Types and Addressing Modes. Description of the non-floating-point data types and of how bit and byte strings are addressed. The addressing modes provided for addressing data in memory are also described in this chapter.

Chapter 4 — Instruction Set Summary. Overview of all the non-floating-point instructions in the instruction set, arranged by functional groups.

Chapter 5 — Floating-Point Operation. Description of the processor's floating-point processing facilities. This chapter includes an overview of floating-point numbers and a description of the floating-point data types and their relationship to the IEEE floating-point standard. Descriptions of the floating-point instructions, exceptions, and faults are also included.

Chapter 6 — Execution Environment. Describes the basic execution environment, how the processor executes instructions, and how the processor manipulates data.

Chapter 7 — Protection Model. Describes the subsystem call and protection mechanisms.

Chapter 8 — Object Addressing. Describes objects, and how they are addressed.

Chapter 9 — Type Management and Access Control. Describes object typing and access control.

Chapter 10 — Faults. Describes the fault handling facilities.

Chapter 11 — Debugging and Tracing Support. Describes the tracing facilities.

Chapter 12 — Interrupts. Describes the interrupt handling facilities.

Chapter 13 — Introduction to Processes, Processors, and Synchronization. An overview of the mechanisms to control processes and processors.

Chapter 14 — Interprocess Communication and Synchronization. Describes the methods by which processes may communicate and synchronize.

Chapter 15 — Process Management. Describes the management of processes.

Chapter 16 — Processor Management and Interrupts. Describes the management of processors.

Chapter 17 — Instruction Formats and Operand Addressing. Describes the instruction format, and the mechanism for specifying the addresses of memory operands.

Chapter 18 — Instruction Reference. A detailed reference about each of the instructions for the processor.

Appendix A — Instruction and Data Structure Quick Reference. A quick reference for the instructions and data structures.

Appendix B — Considerations for Writing Portable Software. Describes the parts of the processor implementation that may change between implementations of other processors in the same family.

1.3 Notation and Terminology

The following paragraphs describe the notation and terminology used in this manual that have special meaning.

1.3.1 Reserved and Preserved

Certain fields in the processor's system data structures are described as being either *reserved* fields or *preserved* fields. A reserved field is one that may be used by other implementations of the processor architecture. To help insure that a current software design is compatible with future processors based on the BiiN™ CPU architecture, the bits in reserved fields should be set to 0 when the data structure is initially created. Thereafter, software should not access these fields.

Some fields in system data structures are shown as being required to be set to either 1 or 0. These fields should be treated as if they were reserved fields. They should be set to the specified value when the data structure is created and not accessed by software thereafter.

A preserved field is one that the processor does not use. Software may use preserved fields for any function.

1.3.2 Set and Clear

The terms *set* and *clear* are used in this manual to refer to the value of a bit field in a system data structure. If a bit is set, its value is 1; if the bit is clear, its value is 0. Likewise, setting a bit means giving it a value of 1 and clearing a bit means giving it a value of 0.

ARCHITECTURE OVERVIEW 2

This chapter provides an overview of the architecture on which the processor described in this manual is based.

2.1 General Comments

This architecture has been designed to provide reliable, high-speed data processing and computational support for the BiiN™ family of computer systems.

The architecture can best be characterized as a high-performance computing engine to which many extensions have been added to support system functions. The computing engine features high-speed instruction execution and ease of programming. Some of its more important attributes include:

- full 32-bit registers
- high-speed, pipelined instruction execution
- a convenient program execution environment with 32 general-purpose registers
- a highly optimized procedure call mechanism that features on-chip caching of local variables and parameters
- extensive facilities for handling interrupts and faults
- extensive tracing facilities to support efficient program debugging and monitoring
- register scoreboarding and write buffering so that memory operations can be overlapped with computations.

The architectural extensions added to the processor's core computing engine are aimed at improving overall system performance and at enhancing the reliability and robustness of the system. Those extensions designed to improve system performance include on-chip support for floating-point arithmetic, virtual memory management, multitasking, and multiprocessing. Those extensions designed to enhance reliability include support for fault tolerant system design through the use of redundant processors and facilities for fine-grained protection so that each process can consist of many distinct address spaces. This latter feature allows each software service needed by an application to have its own address space and yet still execute in the same process as the application. The BiiN™ Operating System, which uses this feature, is not monolith, as are other operating systems, but rather a collection of services.

The following sections describe those features of the core computing engine that are provided to streamline code execution and simplify programming. An overview of the extensions added to this computing engine is provided at the end of the chapter.

2.2 High Performance Program Execution

Much of the design of the architecture has been aimed at maximizing the processor's performance through the implementation of a high-speed computational unit and through minimizing the amount of and effect of memory accesses. The following paragraphs describe several of the mechanisms and techniques used to accomplish this design, including:

- a large register file
- caching of code and procedural data
- overlapped execution of instructions
- many single-clock instructions

2.2.1 Large Register File

A large register file contributes to performance by allowing variables to be held in registers and thus reducing the number of memory accesses required to execute a program. A generous supply of general-purpose registers is provided.

For each procedure, 32 registers are available (28 of which are available for general use). These registers are divided into two types: "global" and "local". Both these types of registers can be used for general storage of operands. The only difference is that global registers retain their contents across procedure boundaries, whereas the processor allocates a new set of local registers each time a new procedure is called.

2.2.2 On-Chip Caching of Code and Data

To further reduce memory accesses, the architecture offers two mechanisms for caching code and data on chip: an instruction cache and multiple sets of local registers. The instruction cache allows prefetching of blocks of instruction from memory, which helps insure that the instruction execution pipeline is supplied with a steady stream of instructions. It also reduces the number of memory accesses required when performing iterative operations such as loops. (The size of the instruction cache can vary. With the processor described in this manual, it is 512 bytes.)

To optimize the architecture's procedure call mechanism, the processor provides multiple sets of local registers. This allows the processor to perform most procedure calls without having to write the local registers out to the stack in memory. The global registers can be used for parameter passing, and the local registers to hold local variables of a procedures.

(The number of local-register sets provided depends on the processor implementation. The processor described in this manual provides four sets of local registers.)

2.2.3 Overlapped Instruction Execution

Another technique that the architecture employs to enhance program execution speed is overlapping memory accesses with computational operations. Separate instruction classes allow this overlapping to occur. For example, all the arithmetic, logic, comparison, branching, and bit operations are performed with just registers and literals. A set of fast, versatile load and store instructions are also provided. These instructions allow burst transfers of 1, 2, 4, 8, 12, or 16 bytes of information between memory and the registers.

Overlapping of memory accesses with computation is accomplished through two mechanisms: write buffering and register scoreboarding. Write buffering allows a store instruction to complete execution as soon as the operand of the store is transferred to an on-chip write-buffer. The transfer of the operand to memory can then occur in parallel with the execution of the instructions following the store.

Register scoreboarding permits instruction execution to continue while data is being fetched from memory. When a load instruction is executed, the processor sets one or more scoreboard bits to indicate the target registers to be loaded. After the target registers are loaded, the scoreboard bits are cleared. While the target registers are being loaded, the processor is allowed to execute other instructions that do not use these registers. The processor uses the scoreboard bits to insure that target registers are not used until the loads are complete. (The checking of scoreboard bits is carried out transparently to software.)

2.2.4 Single-Clock Instructions

It is the intent of the architecture that a processor be able to execute commonly-used instructions such as moves, adds, subtracts, logical operations, and branches in a minimum number of clock cycles. Thus, over 50 instructions can be executed in a single clock cycle.

To maintain a high-execution rate, it must be possible to decode instructions quickly. All the instructions in the architecture are 32 bits long and aligned on 32-bit boundaries. This feature allows instructions to be decoded in one clock cycle. While one instruction is being executed, the next instruction is being decoded.

2.2.5 Efficient Interrupt Model

The architecture provides an efficient mechanism for servicing interrupts from external sources. To handle interrupts, the processor maintains an interrupt table of 248 interrupt vectors (240 of which are available for general use). When an interrupt is signaled, the processor uses a pointer from the interrupt table to perform an implicit call to an interrupt handler procedure. In performing this call, the processor automatically saves the state of the processor prior to receiving the interrupt; performs the interrupt routine; and then restores the state of the processor. A separate interrupt stack is also provided to segregate interrupt handling from application programs.

The interrupt handling facilities also feature a method of evaluating interrupts by priority. The processor is then able to store interrupt vectors that are lower in priority than the task that the processor is currently working on in a pending interrupt section of the interrupt table. When the priority of the processor is lowered, the processor checks the pending interrupts and services the highest priority pending interrupt that is above the processor's priority level.

2.3 Simplified Programming Environment

Partly as a side benefit of its streamlined execution environment and partly by design, processors based on the architecture are particularly easy to program. For example, the large number of general-purpose registers allows relatively complex algorithms to be executed with a minimum number of memory accesses. The following paragraphs describe some of the other features for the architecture that simplify programming.

2.3.1 Highly Efficient Procedure Call Mechanism

The procedure call mechanism makes procedure calls and parameter passing between procedures simple and compact. Each time a call instruction is issued, the processor automatically saves the current set of local registers and allocates a new set of local registers for the called procedure. Likewise, on a return from a procedure, the current set of local registers is deallocated and the local registers for the procedure being returned to are restored. On a procedure call, the program thus never has to explicitly save and restore those local variables and parameters that are stored in local registers.

2.3.2 Versatile Instruction Set and Addressing

The selection of instructions and addressing modes also simplifies programming. The architecture offers a full set of load, store, move, arithmetic, comparison, and branch instructions, with operations on both integer and ordinal data types. It also provides a complete set of Boolean and bit-field instructions, to simplify operations on bits and bit strings.

The addressing modes are efficient and straightforward, while at the same time providing the necessary indexing and scaling modes required to address complex arrays and record structures.

The large 4-gigabyte address space provides ample room for programs and data.

2.3.3 Extensive Fault Handling Capability

To aid in program development, the architecture defines a wide selection of faults that the processor detects, including arithmetic faults, invalid operands, invalid operations, and machine faults. When a fault is detected, the processor makes an implicit call to a fault handler routine, using a mechanism similar to that described above for interrupts. The information collected for each fault allows program developers to quickly correct faulting code. It also allows automatic fault recovery from some faults.

2.3.4 Debugging and Monitoring

To support debugging systems, the architecture provides a mechanism for monitoring processor activity by means of trace events. The processor can be configured to detect as many as seven different trace events, including the instruction execution, branch events, calls, subsystem calls, returns, prereturns, and breakpoints. When the processor detects a trace event, it signals a trace fault and calls a fault handler.

2.4 System-Support Extensions

The system-support extensions in the architecture are built on top of the processor's core computing engine. These extensions are summarized in the following paragraphs.

2.4.1 On-Chip Floating Point

The architecture provides a complete implementation of the IEEE standard for binary floating-point arithmetic (IEEE 754-185). This implementation includes a full set of floating-point operations, including add, subtract, multiply, divide, trigonometric functions, and logarithmic functions. These operations are performed on single precision (32-bit), double precision (64-bit), and extended precision (80-bit) real numbers.

One of the benefits of this implementation is that the floating-point handling facilities are completely integrated into the normal instruction execution environment. Single- and double-precision floating-point values are stored in the same registers as non-floating-point values. In addition, to the 32 (global and local) registers, the four floating-point registers are provided to hold extended-precision values.

2.4.2 String and Decimal Operations

The architecture provides some instructions for moving, filling, and comparing byte strings in memory. These instructions speed up string operations and reduce the amount of code required to handle strings.

The decimal instructions perform move, add with carry, and subtract with carry operations on BCD coded decimals.

2.4.3 Virtual Memory Support

Another of the architecture's important features is support for virtual memory management. The processor's virtual memory mechanism provides each process (or task) with an immediate address space of up to 2^{32} bytes, an extended address space of up to 2^{58} bytes. An address space is paged into physical memory in 4K-byte pages. On-chip memory management facilities handle virtual-to-physical address translation. A on-chip "translation look-aside buffer" (TLB) speeds address translation by storing virtual-to-physical address translations for frequently accessed parts of memory, such as the location of the page tables and the location of often used system data structures.

2.4.4 Multitasking

The architecture offers a variety of process management facilities to support concurrent execution of multiple processes (also known as tasks in some systems). These facilities can be divided into two groups: process scheduling and interprocess communication/synchronization.

The processor provides a unique process handling feature called *self-dispatching*. Here, an operating system schedules processes by queuing them to a dispatch port. Thereafter, the processor handles the dispatching, preempting, and rescheduling of the tasks automatically, independent of the operating system. When using this mechanism, processes can be scheduled by priority, with up to 32 priority levels to choose from. Scheduling within a priority-level is handled on a round-robin basis, based on a time-slice associated with each process.

The processor's interprocess synchronization/communication facilities include support for semaphores and communication ports. Semaphores allow processes that are sharing a resource (e.g. a shared data structure) to synchronize their accesses so that only process is accessing the resource at a time. Semaphores can also be used to signal a process that some event has ocurred. Ports allow processes to pass information. One process can send messages to a port while another process can receive messages from the port. The port handles the enqueuing of messages when when there isn't a process waiting to receive a message, and the enqueuing of processes when there aren't messages to be received.

The Dispatching facility and the interprocess synchronization/communication facilities are fully integrated. For example, suppose a process executes a Receive instruction on an empty port. The execution of the process is suspended, the process is enqueued on the port, the processor goes to its dispatch port to dequeue the highest priority process, loads the state of this process on-chip, and then executes it. This entire sequence of operations is handled without any operating system intervention.

The BiiN™ Operating System uses these same facilities for its own synchronization. Unlike operating systems on other computers, there is no need for the BiiN™ Operating System to mask-out interrupts for synchronization purposes. In effect, the operating system can stay out of the way of processes and interrupt routines that need very fast real-time responsiveness.

2.4.5 Multiprocessing

The architecture provides several mechanisms designed to simplify the design of multiple processor systems, allowing several processors to run in parallel, using shared memory resources. One of these mechanisms is described above.

All the processors in a system can share the same dispatch port so that the processing load is automatically load-balanced across all processors. The communication/synchronization mechanisms were designed to work in a multiple-processor environment. Because these mechanisms are implemented directly in the silicon, multiprocessing is more efficient on BiiN™ systems than on other computer systems.

The processor also provides an "interagent communication" (IAC) mechanism that allows processors to exchange messages among themselves on the bus. This mechanism operates similarly to the interrupt mechanism, except that IAC messages are passed through dedicated memory-mapped registers on the system bus. The IAC mechanism can be used to preempt processes running on another processor, to manage interrupt handling, or to control (start, stop, resume) another processor.

A set of atomic instructions are also provided to perform atomic modifications to a memory location. For example, the **atmod** instruction can be used to set one or more bits in a memory word. The value of the memory location before the modification is a result operand.

2.4.6 Fault Tolerance

The architecture supports fault-tolerant system design through the use of a BiiN™ system architecture component called the Bus Extension Unit (BXU). In a BiiN™ system, a *processor module* consists a processor, a BXU, and an external cache (consisting of SRAM chips). The directory for this cache is on the BXU, which also takes care of cache coherency in a multiprocessor configuration. The BXU connects to the system bus so that processor data flows through the BXU to the system bus and then to main memory.

Two processor modules can be configured to work independently or as a "fault-checking module". In the latter case, one module is refered to as the "master" and the other as the "checker". The master and checker operate in lock-step executing the identical instruction sequence. The principal difference between master and checker occurs in putting data onto the system bus. The BXU in the master module puts data onto the system bus, but the checker does not; instead, the checker compares the data output by the master with the data it would have put out (if it was the master). If the data does not match, an error is signalled in time for the memory unit to suppress the memory write. The role of master/checker are switched on every clock cycle so that a checker failure does not go undetected.

This fault detection mechanism supports several fault detection and recovery techniques, including self-healing, and continuous-operation systems. For a more in depth overview, see the *BiiN™ Systems Overview.*

2.4.7 Support for Reliable and Secure System Software

Most computer architectures employ a supervisor/user protection scheme. In this scheme, a process has two address spaces, one for the user/application, and one for the operating system. Each process can have a different user address space. Some other architectures have generalized this scheme to provide more than 2 address spaces per process; for example, the Intel 286 and 386 architectures support 4 address spaces per process, labelled 0 through 3. Address space 0 typically contains the operating system kernel and address space 3 the user application. The other 2 address spaces are can be used by higher-level operating system services or other system services (e.g. a DBMS). These address spaces are hiearchical, with repect to accessibility. For example, a procedure executing in address space 1 has complete access to address spaces 1, 2, and 3.

The BiiN™ processor provides a more fine-grained protection scheme. The number of address spaces per process is effectively unlimited. Furthermore, each address space is independent of the others so that execution in one address space does not imply implicit access to another. The processor provides efficient call/return instructions for switching address spaces. A program can also pass selective access to part of its address space to a procedure in a different address space.

This protection scheme is used in BiiN™ systems to provide reliable and secure system software. For example, the BiiN™ Operating System is a collection of protected services, compared with the monolithic nature of other commercial operating systems. This scheme also allows new services to be easily added without compromising the reliability and security of established services, because the new service would execute in its own address space and would not be able to corrupt any other services.

2.5 Addressing and Protection

The "virtual address space" of a BiiN™ system is made up of objects. An "object" is a typed, protected segment of memory. The size of an object can be a small as 64 bytes and as large as 2^{32} bytes (over 4 Gigabytes). Objects that are bigger than 4,096 bytes are paged so that only the actively referenced parts of an object need be in physical memory. The virtual address space of a BiiN™ system can contain up to 2^{26} (more than 64 million) objects. Thus, the processor can address up to 2^{58} bytes of virtual memory.

2.5.1 How are Objects Referenced?

An "access descriptor" (AD) is a protected pointer to an object. The only way to reference an object is via an AD.

In most computer systems, a pointer is simply an arbitrary bit pattern used as an address. Such pointers can be corrupted without detection by the hardware or OS. ADs are specially tagged memory words that can only be created or modified in carefully controlled ways. A memory word in a BiiN™ computer system is actually 33 bits, the 33rd bit being the tag bit that distinguishes ADs from data. Changing an AD in an unauthorized way invalidates the AD (by turning off the tag bit).

An AD contains both addressing information, used to find the object in memory, and also *access rights*, that indicate what operations are possible with the AD (Figure 2-1).

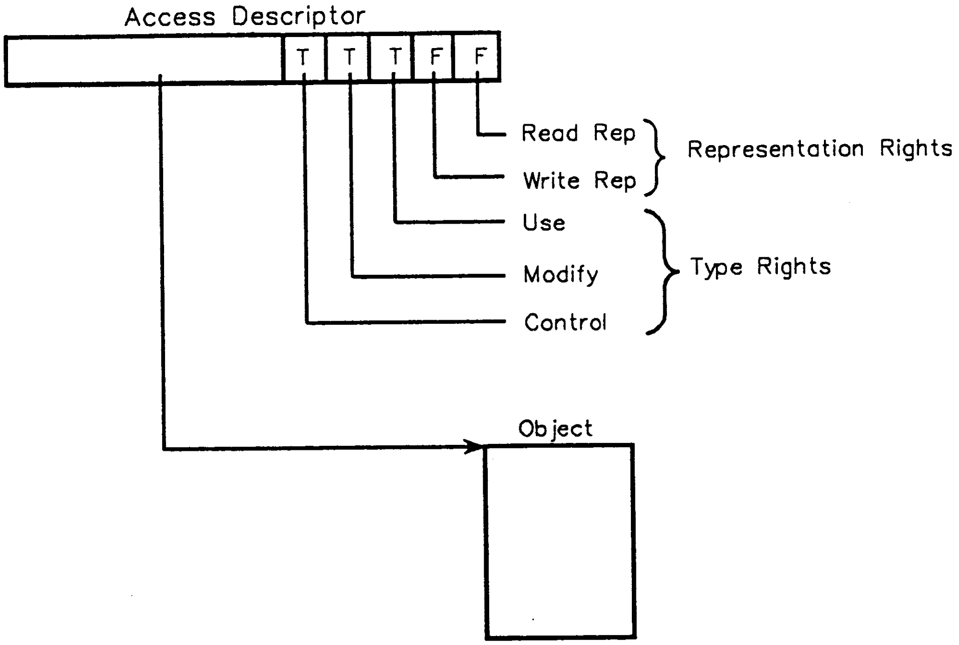


Figure 2-1. AD and Object

There are five access rights stored in each AD:

Read rep rights Required to read an object's representation. This right is checked and enforced by the processor on every read access.

Write rep rights Required to write an object's representation. This right is checked and enforced by the processor on every write access.

Three *type rights* Required for type-specific operations. These rights can be defined differently and renamed for each type of object. These rights are checked and enforced by the processor on instructions that manipulate a hardware-recognized type (e.g. port, semaphore, process). In all other cases, these rights are checked and enforced by software.

Different users or programs may have ADs with different rights to the same object. Mary may have an AD with both read and write rights to an object and John may have an AD with only read rights.

To reference a particular field within an object, a program can use a two-part *virtual address*: a 32-bit offset to a byte within the object plus an AD to the object.

2.5.2 Address Spaces Within a Program

A BiiN™ program can be partitioned into multiple protected modules (referred to as "subsystems"), each with its own "linear address space" or "domain". See Figure 2-2. A linear address space contains up to 2^{32} bytes mapped onto four objects by the processor: static data, instructions, stack, and a special object used only by the OS. Each domain provides access to a particular collection of objects that can be reached from the objects mapped by its linear address space. Only those program modules with a "need to know" about a particular object have access to it, and then only have the access rights that they need. For example, each software service can be placed in its own domain.

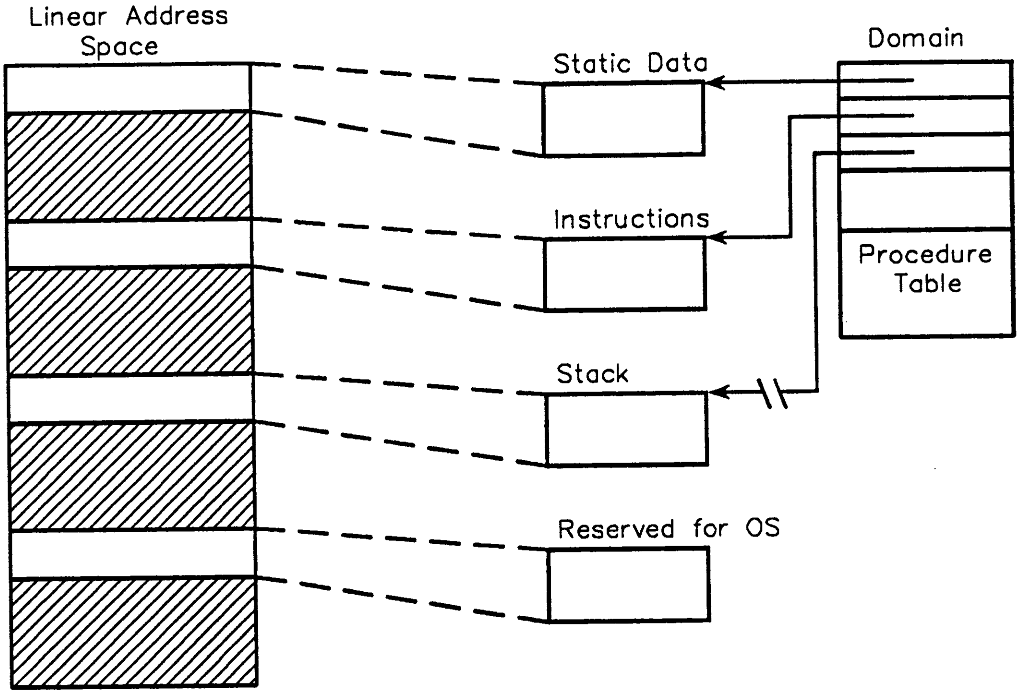


Figure 2-2. Linear Address Space and Domain

The memory access instructions (e.g. Load, Store) of the processor support both virtual addressing and linear addressing. A virtual address is 2 words and consists of an AD and a 32-bit offset. A linear address is 1 word (32 bits). The high-order 2 bits select one of the four objects that make-up the current linear address space. The low-order 30 bits specify a byte displacement into the selected object.

When one domain calls a routine in another domain, switching address spaces is done by the processor, as part of a inter-subystem call.

BiiN™ programmers can choose either a one-domain (linear) or multiple-domain (structured) organization for their programs:

- A program that does not use object-based protection can be compiled entirely into one domain. Because there is a single linear address space for the entire program, linear addresses can be used for pointers. This is the typical approach used in porting programs from other computer systems over to BiiN™ systems.
- A program that uses object-based protection can be compiled into multiple domains. Because linear addresses are only valid within a particular domain, ADs or virtual addresses are normally used for pointers.

The organization of modules into domains can be varied to trade greater protection for greater execution speed. For example, inter-related software services can be grouped into the same domain.

2.5.3 Three-Fold Protection

Each object in a BiiN™ system is protected in three ways, as shown in Figure 2-3:

- Limited access: Only those modules with a "need-to-know" can reference the object.
- Type checking: If an object's type is not the proper type required by an operation, then the operation fails.
- Right checking: If the AD used does not have rights that allow the operation, then the operation fails.

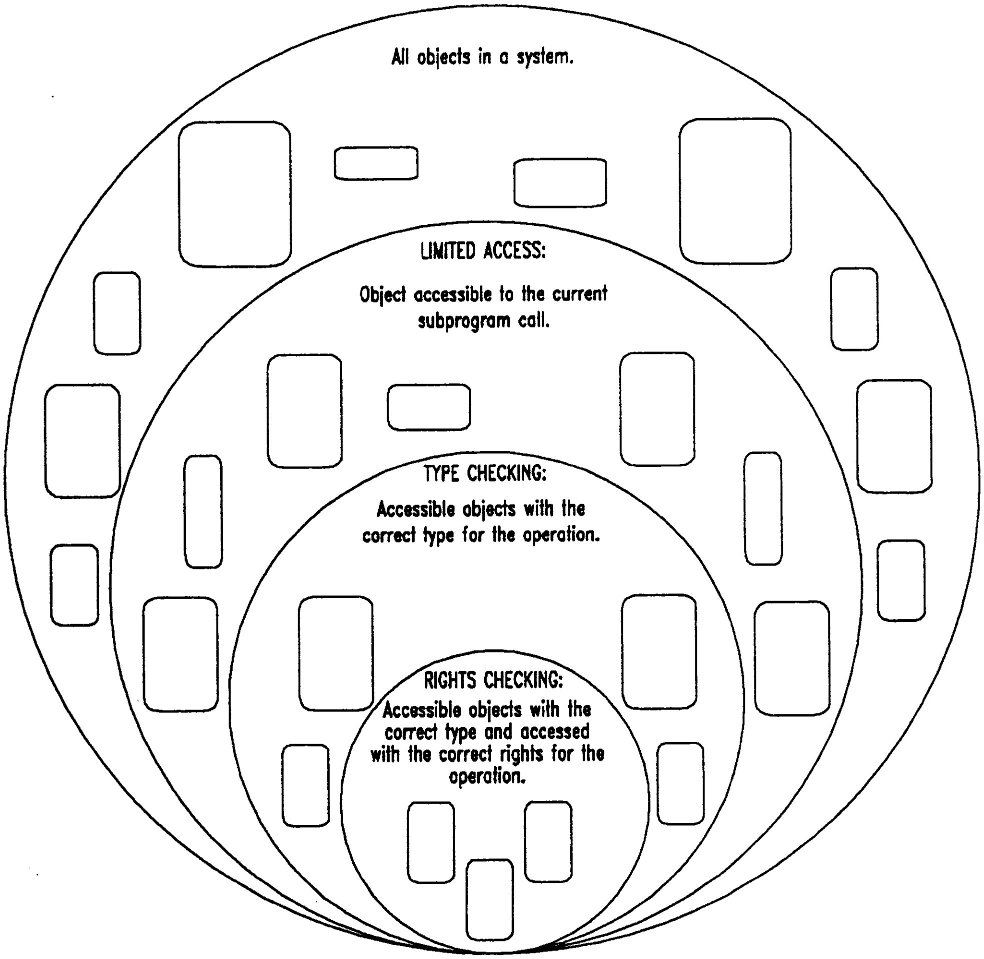

All objects in a system.
LIMITED ACCESS:
Object accessible to the current subprogram call.
TYPE CHECKING:
Accessible objects with the correct type for the operation.
RIGHTS CHECKING:
Accessible objects with the correct type and accessed with the correct rights for the operation.

Figure 2-3. Three-Fold Object Protection

DATA TYPES AND ADDRESSING MODES 3

This chapter describes the available data types and addressing modes.

3.1 Data Types

The following data types are recognized:

- Integer (8, 16, 32, and 64 bits)
- Ordinal (8, 16, 32, and 64 bits)
- Real (32, 64, and 80 bits)
- Decimal (ASCII digits)
- Bit Field
- Byte String
- Triple-Word (96 bits)
- Quad-Word (128 bits)
- Access Descriptor

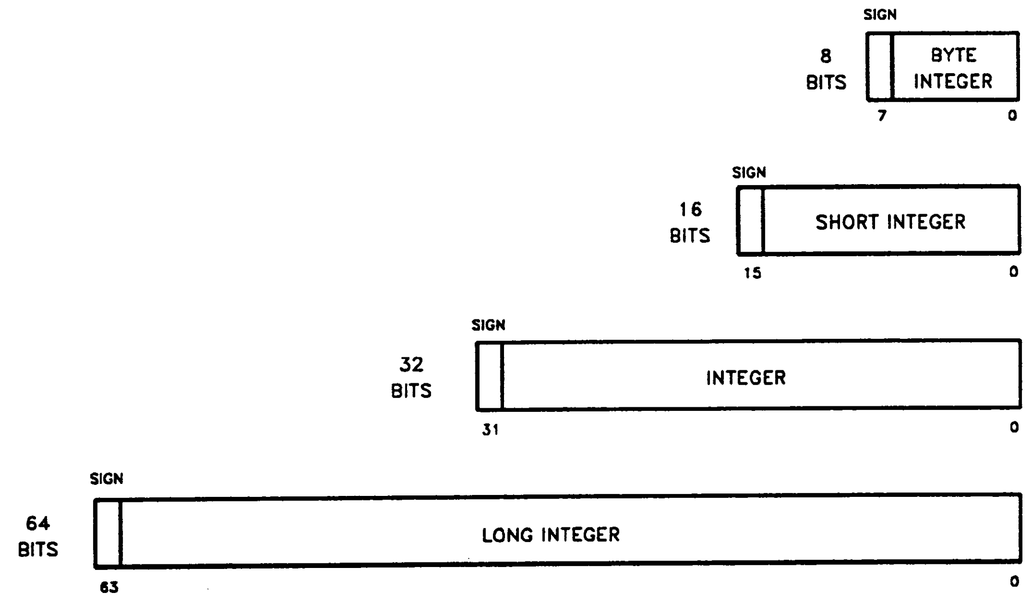
The integer, ordinal, real, and decimal data types can be thought of as numeric data types because some operations on these data types produce numeric results (for example, add and subtract).

The bit field, byte string, triple-word and quad-word data types represent groupings of bits, bytes, or words that can be operated on as a whole, regardless of the nature of the data contained in the group. These data types facilitate the moving of blocks of bits or bytes.

The access descriptor (AD) data type is a special data type that is used in conjunction with objects. The AD data type is described in Chapter 8.

3.1.1 Integers

Integers are signed whole numbers, which are stored and operated on in two's-complement format. Four sizes of integers are available: 8 bit (byte integers), 16 bit (short integers), 32 bit (integers), and 64 bit (long integers). Figure 3-1 shows the formats for the four integer sizes and the ranges of values allowed for each size.

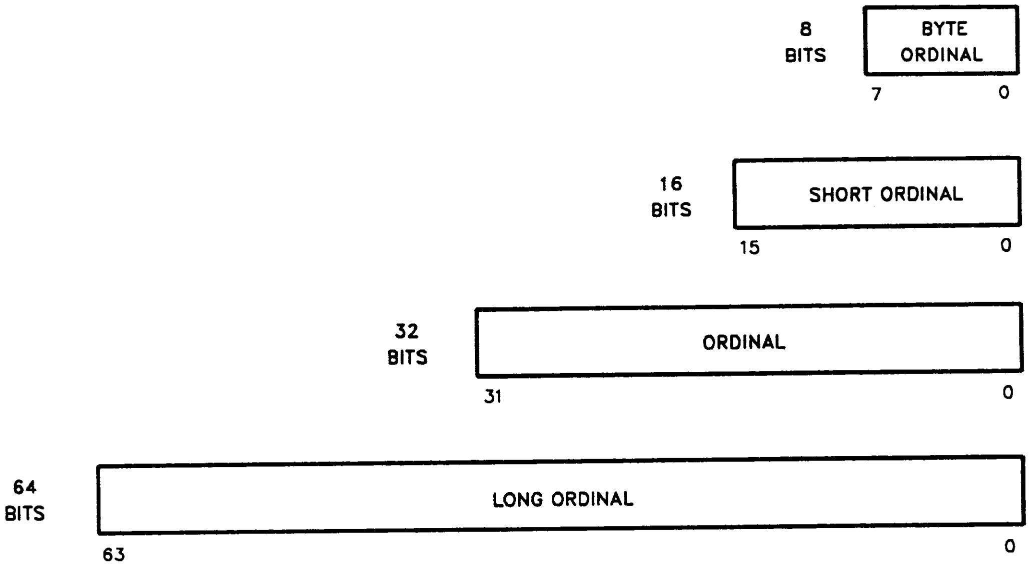


DATA TYPE	RANGE	DECIMAL EQUIVALENT
BYTE INTEGER	-2^7 to $2^7 -1$	−128 to 127
SHORT INTEGER	-2^{15} to $2^{15} -1$	−32,768 to 32,767
INTEGER	-2^{31} to $2^{31} -1$	$\sim -2.14 \times 10^9$ to $\sim 2.14 \times 10^9$
LONG INTEGER	-2^{63} to $2^{63} -1$	$\sim -9.22 \times 10^{18}$ to $\sim 9.22 \times 10^{18}$

Figure 3-1. Integer Format and Range

3.1.2 Ordinals

Four sizes of ordinals are available: 8 bit (byte ordinals), 16 bit (short ordinals), 32 bit (ordinals), and 64 bit (long ordinals). Figure 3-2 shows the formats for the four ordinal sizes and the ranges of values allowed for each size.



DATA TYPE	RANGE	DECIMAL EQUIVALENT
BYTE ORDINAL	0 to $2^8 - 1$	0 to 255
SHORT ORDINAL	0 to $2^{16} - 1$	0 to 65,535
ORDINAL	0 to $2^{32} - 1$	0 to ~4.29 x 10^9
LONG ORDINAL	0 to $2^{64} - 1$	0 to ~1.84 x 10^{19}

Figure 3-2. Ordinal Format and Range

Ordinals can be used for both numeric and non-numeric operations. For numeric operations, ordinals are treated as unsigned whole numbers. Some arithmetic instructions operate on ordinals. For non-numeric operations, ordinals contain bit fields, byte strings, and Boolean values.

When ordinals are used to represent Boolean values, a 1 represents a TRUE and a 0 represents a FALSE.

3.1.3 Reals

Reals (also known as floating-point numbers) are one of three sizes: 32 bit (reals), 64 bit (long reals), and 80 bit (extended reals). The real-number format conforms to ANSI/IEEE Std. 754-1985, the IEEE Standard For Binary Floating-Point Arithmetic. Real numbers are discussed in Chapter 5.

3.1.4 Decimals

Three instructions perform operations on decimal values when the values are presented in ASCII format. Figure 3-3 shows the ASCII format for decimal digits. Each decimal digit is contained in the least-significant byte of an ordinal (32 bits). The decimal digit must be of the form $0011dddd_2$, where $dddd_2$ is a binary-coded decimal value from 0 to 9. For decimal operations, bits 8 through 31 of the ordinal containing the decimal digit are ignored.

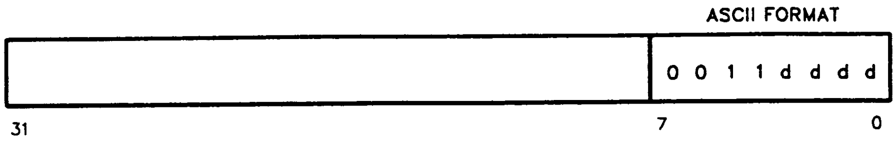


Figure 3-3. Decimal Format

3.1.5 Bits and Bit Fields

Several instructions perform operations on individual bits or fields of bits within an ordinal (32 bit) operand. Figure 3-4 shows these data types.

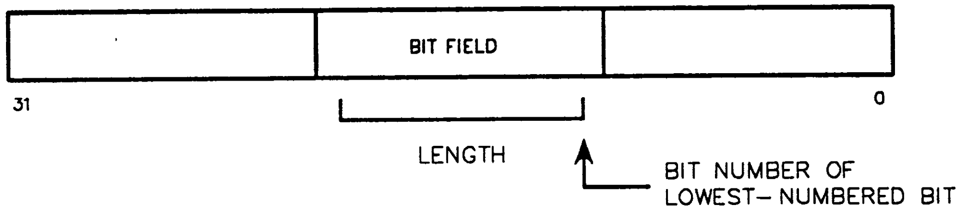


Figure 3-4. Bits and Bit Fields

An individual bit is specified for a bit operation by giving its bit number in the ordinal in which it resides. The least-significant bit of a 32-bit ordinal is bit 0; the most-significant bit is bit 31.

A bit field is a contiguous sequence of bits of from 0 to 32 bits in length within a 32-bit ordinal. A bit field is defined by giving its length in bits and the bit number of its lowest-numbered bit.

3.1.6 Byte String

A byte string is a contiguous sequence of byte ordinals. The length of a byte string is the number of bytes in the string; a length of zero specifies an empty string. The maximum length of a byte string is 2^{32} - 1 bytes.

Byte-string operations are performed on byte strings in memory. The address of a byte string is the address of the first byte in the string. Consecutive bytes of the string are stored in increasing byte addresses.

3.1.7 Triple and Quad Words

Triple and quad words refer to consecutive bytes in memory or in registers: a triple word is 12 bytes and a quad word is 16 bytes. These data types facilitate the moving of blocks of bytes. The triple-word data type is useful for moving extended-real numbers (80 bits).

The quad-word instructions (ldq, stq, and movq) offer the most efficient way to move large blocks of data.

3.2 Byte, Word, and Bit Addressing

Some instructions move blocks of data from memory to registers (load) and from registers to memory (store). The allowable sizes for blocks are bytes, half-words (2 bytes), words (4 bytes), double words, triple words, and quad words. For example, the stl (store long) instruction stores an 8-byte (double word) block of data in memory.

When a block of data is stored in memory, the least-significant byte of the block is stored at a base memory address and the following bytes are stored at successively higher addresses.

When loading a byte, half-word, or word from memory to a register, the least-significant bit of the block is always loaded in bit 0 of the register. When loading double words, triple words, and quad words, the least-significant word is stored in the base register. The following words are then stored at successively higher-numbered registers. Double words, triple words, and quad words must also be aligned in registers to natural boundaries as described in Section 6.2.4.

Bits can only be addressed in data that resides in a register. Bit 0 in a register is the least-significant bit and bit 31 is the most-significant bit.

This numbering of bits within bytes, bytes within words, and words within multiwords is sometimes referred to as "little-endian".

3.3 Addressing Modes

This section provides a summary of the addressing modes available for operands of instructions. Detailed information (including memory representation) may be found in Chapter 17.

3.3.1 Literals and Registers

The majority of the instructions are register-to-register operations (denoted as a "REG"-format instruction). These instructions usually take three operands (two sources and a destination), although some instructions use fewer operands. Registers are described in Chapter 6.

The REG-format instructions may also use small literal values in place of a register designation. Literal values are restricted to integer or unsigned values that can be represented in five bits (-16 through 15 for signed integers, 0 through 31 for unsigned), or the floating-point constants +0.0 and +1.0.

3.3.2 Memory Access

Some of the remaining instructions transfer values between registers and memory (denoted as "MEM"-format instructions). These instructions may use any of the following addressing modes:

- **register indirect**: a register contains the target address.
- **register indirect with displacement**: a constant is added to a register to compute the target address.
- **register indirect with index**: two registers are added (after an optional scaling of the second "index" register) to compute the target address.
- **register indirect with index and displacement**: similar to register indirect with index, with an additional constant added to the result to compute the target address.

- **absolute**: the instruction contains the target address.
- **absolute with index**: similar to register indirect with index, but a literal constant provides the base address.
- **IP with displacement**: the target address is computed as an offset from the current instruction pointer (IP).

The first four addressing modes (those that are "register indirect" plus something) may be used with either a linear address or a virtual address. (The remaining addressing modes may be used with only linear addresses.) A linear address is a conventional address, viewing the address space immediately accessible to the process as a contiguous sequence of bytes. A virtual address is a pair of values: an access descriptor denoting an object, and an offset from the beginning of that object. See Chapter 8 for details.

INSTRUCTION SET SUMMARY 4

This chapter provides an overview of the instruction set. Chapter 18 gives detailed descriptions of each instruction.

4.1 Instruction Groups

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

- Data Movement
- Address Computation
- Arithmetic (Ordinal, Integer and Floating Point)
- Decimal
- Logical
- Bit and Bit Field
- Comparison
- String
- Conversion
- Branch
- Call/Return
- Execution Environment Management
- Debug
- Object Management
- Atomic
- Process Management

The following sections give a brief overview of the instructions in each of these groups. The floating-point instructions are described in Chapter 5.

4.2 Data Movement

The data-movement instructions include those instructions that move data from memory to the general registers; that move data from the general registers to memory; and that move data among the general registers.

4.2.1 Load

The four types of load instructions are load, load virtual, load mixed, and load virtual mixed. The load and load virtual instructions load general data (words that are not ADs) from memory to registers; the load mixed and load virtual mixed instructions move words that contain either ADs or general data.

4.2.1.1 Load (Linear)

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

The load instructions copy the values from the linear address space to the destination general registers. For the **ld**, **ldob**, **ldos**, **ldib**, and **ldis** instructions, a linear address and a destination register are specified and the value at the given address in memory is copied into the register. Zero and sign extending is performed automatically for byte and short (half-word) operands; meaning that ordinals are zero-padded to the length of the destination operand, and integers are sign-extended to the length of the destination operand.

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

NOTE

When using the load, store, and move instructions that move 8, 12, or 16 bytes at a time, the rules for register alignment must be followed. Refer to Section 6.2.4 for a discussion of these rules.

For the load instructions, the tag bits in the registers are always zeroed.

4.2.1.2 Load Virtual

ldvib	load virtual byte integer
ldvob	load virtual byte ordinal
ldvis	load virtual short integer
ldvos	load virtual short ordinal
ldv	load virtual
ldvl	load virtual long
ldvt	load virtual triple
ldvq	load virtual quad

The load virtual instructions copy values from any accessible object, which may be outside the current linear address space, into the registers.

The source operand specifies a virtual address (AD for an object and an offset into the object) and the destination operand specifies the register (or first register of a group of registers) to receive the values. The load virtual instructions are essentially the same as the load instructions, except that they allow values to be loaded from any object specified with a valid AD.

For the load virtual instructions, the tag bits in the registers are always zero.

Object addressing is described in Chapter 8. The virtual addressing modes are described in Chapter 17.

4.2.1.3 Load Mixed

ldm	load mixed
ldml	load mixed long
ldmq	load mixed quad

The load mixed instructions perform the same functions as the load instructions, except that the tag bits are also copied.

4.2.1.4 Load Virtual Mixed

ldvm	load virtual mixed
ldvml	load virtual mixed long
ldvmq	load virtual mixed quad

The load virtual mixed instructions perform the same functions as the load virtual instructions, except that the tag bits are also copied.

4.2.2 Store

For each load instruction there is a corresponding store instruction, which copies a value from the source registers to memory.

4.2.2.1 Store (linear)

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

The store instructions copy the source registers (with the tag bits set to zero) into the linear address space. For the **st**, **stob**, **stos**, **stib**, and **stis** instructions, a register and linear address are specified and the value in the register is copied into memory. For the byte and short instructions, the value in the register is truncated for the shorter memory location. For the **stib** and **stis** instructions, the truncation can lead to integer overflow if the register value is too large to be represented in the shorter memory location.

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

4.2.2.2 Store Virtual

stvib	store virtual byte integer
stvob	store virtual byte ordinal
stvis	store virtual short integer
stvos	store virtual short ordinal
stv	store virtual
stvl	store virtual long

stvt	store virtual triple
stvq	store virtual quad

The store virtual instructions copy the source registers (with the tag bits set to zero) to any object with a valid AD.

4.2.2.3 Store Mixed

stm	store mixed
stml	store mixed long
stmq	store mixed quad

The store mixed instructions copy ADs, general data, or a mixture of the two from registers to the current linear address space. The destination tag bits are copied from the source tag bits; meaning that ADs are copied as ADs, and data values are copied as data values.

4.2.2.4 Store Virtual Mixed

stm	store mixed
stml	store mixed long
stmq	store mixed quad
stvm	store virtual mixed
stvml	store virtual mixed long
stvmq	store virtual mixed quad

The store virtual mixed instructions copy ADs, general data, or a mixture of the two from registers to any object.

4.2.3 Move

The move instructions copy values from a register or group of registers (or small literals) to another register or group of registers.

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

The move (data) instructions zero the tag bit in the destination register or registers.

movm	move mixed word
movml	move mixed long word
movmq	move mixed quad word

The move mixed instructions copy the tag bit during the move, meaning that ADs remain ADs, and data values remain data values.

4.2.4 Address Computation

lda	load address
cvtadr	convert address

The **lda** instruction computes an effective address using one of the addressing modes. A frequent use of this instruction is to load a constant into a register, or to add a large constant to a register.

The **cvtadr** instructions translates a linear address into a virtual address. The resulting address consists of the AD for the region that contains the linear address and the offset into that region.

4.3 Arithmetic

Table 4-1 lists the arithmetic instructions and the data types on which those instructions operate. An "X" in this table indicates that an instruction is provided for the specified operation and data type. An "*" indicates that the specified operation can be performed on the specified data type, but that a unique instruction for this operation is not provided. For example, a specific instruction is not provided that allows two extended-real values to be added together. However, this operation can be carried out with either the add real (**addr**) or the add long real (**addrl**) instruction. A "N/A" in this table indicates that the operation is not appropriate for the data type.

Table 4-1. Arithmetic Operations

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

A summary of the arithmetic instructions for real (floating-point) data types is provided in Chapter 5. The following sections describe the arithmetic instructions for ordinal and integer data types.

Arithmetic instructions are generally "three-operand" instructions, specifying two source operands and a destination operand. Exceptions are noted in Chapter 17.

4.3.1 Add, Subtract, Multiply, and Divide

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

These instructions operate on 32-bit integer or ordinal operands in registers (or small literals) and store the results in a register. The integer versions generate an integer overflow if the result is outside the range of the destination.

4.3.2 Extended Arithmetic

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

The **addc** and **subc** instructions add or subtract two operands and a carry bit (in the condition code). If the result generates a carry, the carry bit in the condition code is set. Also, a second condition code bit is set if the operation would have resulted in an integer overflow condition.

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

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

4.3.3 Remainder and Modulo

remi	remainder integer
remo	remainder ordinal
modi	modulo integer

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

4.3.4 Shift and Rotate

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

In **shlo** and **shli**, zeroes are shifted in from the least significant bit. If the bits shifted out in **shli** are not the same as the sign bit, an integer overflow exception is generated. In **shro**, zeroes are shifted in from the most significant bit. In **shri** and **shrdi**, the value of the sign bit is shifted in from the most significant bit. The **shri** instruction discards the bits shifted out which has the effect of rounding the result toward negative. Hence, the **shri** may generate different result than **divi** by 2 when the operand is negative. The **shrdi** corrects the result, by adding 1 to the result, if the bits shifted out are non-zero and the operand is negative.

Shli and **shrdi** instructions are equivalent to **muli** and **divi** by two.

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

4.4 Decimal

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

These instructions operate on 32-bit decimal operands that contain an 8-bit, ASCII-coded decimal in the least-significant byte of the word (as shown in Figure 3-3).

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

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

4.5 Logical

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

These instructions provide logical bit-by-bit operations on the values contained in the registers (or small literals).

4.6 Bit and Bit Field

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

4.6.1 Bit Operations

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

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

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

The **alterbit** instruction alters the state of a specified bit in an ordinal according to the condition code. If the condition code is $x1x_2$, the bit is set; if the condition code is $x0x_2$, the bit is cleared.

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

4.6.2 Bit Field Operations

extract	Extract bit field
modify	Modify bit field

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

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

4.7 Comparison

Several types of instructions may be used to compare two operands. The following sections describe the compare instructions for ordinal, integer, and AD data types. The compare instructions for real data types are discussed in Chapter 5.

4.7.1 Compare and Conditional Compare

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

The compare instructions compare two operands, then set the condition code according to the result. The condition code is set to indicate whether one operand is less than, equal to, or greater than the other operand.

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

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

The conditional compare instructions are provided specifically to optimize two-sided range comparisons to check if A is between B and C (that is, B ≤ A ≤ C). Here, a compare instruction (**cmpi** or **cmpo**) is used to check one side of the range (for example, A ≥ B) and a conditional compare instruction (**concmpi** or **concmpo**) is used to check the other side (for example, A ≤ C) according to the result of the first comparison.

4.7.2 Compare and Increment or Decrement

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

These instructions compare two operands, set the condition code according to the results, then increment or decrement one of the operands. These instructions are intended for loop end comparisons.

4.7.3 Compare Mixed

cmpm	compare mixed
chktag	check tag

The compare instructions described above do not check the tag bits of the operands: they are assumed to be 0. When the state of the tag bit is important in a comparison, the compare mixed instruction (**cmpm**) is used. This compares two words for either access equality (if the words are both ADs) or data equality (if the words are both general data words).

For example, if the two words are ADs (their tag bits are set to 1), the processor compares the object index field for each word. If the ADs point to the same object, the condition code bits are set to 010_2. Likewise, if the two words are data words (their tag bits are set to 0), the processor performs a bit-by-bit comparison of the two words, and if the words are equivalent, the condition code bits are set to 010_2.

If the tag bits of the two words are different, or if the object indices or word values are different, the condition code is set to 000_2.

The **chktag** instruction checks the tag bit of an operand and set the condition code bits to 010_2 if the tag bit is 1 and 000_2 if the tag bit is 0.

4.8 String

movstr	move string
movqstr	move quick string
fill	fill string
cmpstr	compare string
scanbyte	scan byte equal

The **movstr** and **movqstr** instructions move a byte string from one location in memory to another. These instructions operate identically except that the **movstr** instruction guarantees that if the strings overlap, no byte in the source string is overwritten until it is copied to the destination string.

The **fill** instruction copies an ordinal operand repeatedly into a byte string in memory.

The **cmpstr** instruction compares two byte strings of equal length, and then sets the condition code to show whether or not the strings are identical.

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

4.9 Conversion

cvtri	convert real to integer
cvtril	convert real to integer long
cvtzri	convert truncated real to integer
cvtzril	convert truncated real to integer long
cvtir	convert integer to real
cvtilr	convert integer long to real

These instructions convert data between integers and floating-point numbers (reals). They are discussed in detail in Chapter 5.

4.10 Branch

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

- unconditional branch
- conditional branch
- compare and branch

Most of the branch instructions specify the target IP with a signed displacement to be added to the current IP. Extended branch instructions specify the memory address of the target IP using one of the addressing modes.

4.10.1 Unconditional Branch

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

The **b** and **bx** instructions cause program execution to jump to the specified target IP.

The **bal** and **balx** instructions store the address of the next instruction in a register, then jump to the target IP. For the **bal** instruction, the RIP is automatically stored in register G14. For the **balx** instruction the RIP is stored in the destination register of the instruction. As described in Chapter 7, the branch and link instructions provide a alternate method of performing procedure calls that do not use the processor's call/return mechanism. The saved instruction address is used as a return IP.

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

4.10.2 Conditional Branch

With the conditional branch (branch if) instructions, the processor checks the condition code. If the condition code "matches" the mask value specified in the instruction, the processor jumps to the target IP. These instructions use the displacement plus IP method of specifying the target IP:

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

Refer to Section 6.4 for an explanation of the condition-code bits.

The **bo** and **bno** instructions refer to comparisons of real numbers. Ordered and unordered real numbers are described in Chapter 5.

4.10.3 Compare and Branch

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

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

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

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

4.10.4 Conditional Faults

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

The conditional fault instructions generate a fault according to the state of the condition code.

For further information about faults and fault-related instructions, see Chapter 10.

4.10.5 Conditional Tests

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

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

4.11 Call and Return

The call/return mechanism makes calls to procedures; these procedures may be located in the current linear address space (a "local" call), or in another linear address space (a "subsystem" call). The local call/return mechanism is described in detail in Chapter 6, while the subsystem call/return mechanism is described in detail in Chapter 7.

call	call
callx	call extended
calld	call domain
calls	call system
ret	return

The **call** and **callx** instructions call local procedures (procedures in the current linear address space). The **call** instruction specifies the target procedure by adding a signed displacement to the IP. The **callx** instruction uses an address mode to specify the target procedure. For these instructions, a new set of local registers and a new stack frame are allocated for the called procedure.

The **calld** instruction calls a procedure in another subsystem. This instruction switches to another linear address space. One of the operands for this instruction specifies a special data structure called a domain object. The domain object specifies the ADs for regions 0, 1, and 2 of the target linear address space. (Region 2 is selected indirectly by means of a subsystem ID.) The domain object also provides a procedure table, which contains pointers to procedures in the selected subsystem. Another operand in the **calld** instruction selects a procedure entry from the procedure table.

The **calls** instruction operates similarly to the **calld** instruction, except that the system domain is used instead. The system domain is a special domain that is shared by all processes.

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

4.12 Execution Environment Management

modac	modify arithmetic controls
flushreg	flush local registers
ldcsp	load control-stack pointer

The **modac** instruction modifies the arithmetic controls register under the control of a mask. See Section 6.4 for details.

The **flushreg** instruction stores the contents of all the local register sets, except the current set, in the register save area of their associated stack frames, and forces these register sets to be restored from memory upon return. This is necessary in order to access or modify the value of a local register in a previous frame. If this instruction is not executed, accessing a previous stack frame may not necessarily access the corresponding local register value, since some stack frames are cached. See Appendix B for details.

The subsystem procedure mechanism maintains extra linkage information in a special stack called the "control stack". (The control stack is contained in a per-process environment table and is described in Section 7.3.3.) An on-chip control stack pointer is maintained for the control stack of the current process. The **ldcsp** instruction returns the current control stack pointer independent of whether the value is on-chip or not.

4.13 Debug

Debugging and monitoring of program activity is supported through the use of trace events. The following instructions support these debugging and monitoring tools:

modtc	modify trace controls
mark	mark
fmark	force mark

The trace functions are controlled through the process trace controls for the current process. Some of the trace-control bits allow various types of tracing to be enabled or disabled. Other bits act as flags to indicate when an enabled trace event has been detected. Trace controls are described in Chapter 11.

The **modtc** instruction modifies the bits in the process trace controls.

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

4.14 Object Management

cread	create AD
restrict	restrict rights
amplify	amplify rights
inspacc	inspect access
ldtdo	load type definition object

The **cread** instruction allows a privileged operating system module to create an AD. Here, the procedure presents a general data word to the processor that contains an object index. The instruction then sets the tag bit to 1 and the rights bits set to read only.

The **restrict** and **amplify** instructions allow a software module to restrict or amplify, respectively, the access rights of an AD that it possesses.

The **inspacc** returns the respective page rights of a specified page of an object. This instruction is used in system software to check the accessibility of a location.

When a software module creates an object and AD, it has the option of including typing information for the object in the form of a type definition object (TDO) and its associated AD. The **ldtdo** instruction loads the AD of the TDO associated with a particular object into a destination register.

4.15 Atomic Instructions

atadd	atomic add
atmod	atomic modify
atrep	atomic replace mixed

The atomic instructions perform read-modify-write operations on operands in memory. These instructions provide primitive operations for synchronization among multiple processors in a shared-memory system.

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

The **atadd** and **atmod** instructions assume that the target word in memory is a general data word and always clears the tag bit. The **atrep** instruction replaces a word in memory (including its tag bit) with a source operand. This instruction atomically inserts an AD directly into a memory location.

4.16 Process Management

Several instructions are available for process management. These instructions do not dictate a particular process management scheme. Instead they provide support for a wide variety of process management mechanisms. These instructions can be divided into two groups:

- process control
- interprocess communication

Process management instructions must have access to the correct valid address descriptors. Address descriptors are available only in supervisor mode on systems with "tag mode" disabled, or to any process under defined conditions if "tag mode" is enabled. See Section 16.3 for details.

Process management is described in detail in Chapters 15 and 16.

4.16.1 Process Control

The following instructions provide process control services:

saveprcs	save process
resumprcs	resume process
schedprcs	schedule process
sendserv	send service
ldtime	load process time
modpc	modify process controls
ldglobals	load from process globals

Three data structures are used for process control: a process object, a process global object, and a dispatching port. The process object maintains information about the process, such as the status of the execution environment when the process was last suspended, and system resources allocated to the process. The process global object provides storage global information associated with the process. The dispatching port is used for queuing processes that are waiting for execution.

The **resumprcs** instruction switches to the specified process. The **saveprcs** instruction causes the current state of the currently running process to be saved in the proces object.

These two instructions perform roughly the same functions as the RESUME and SAVE functions of most UNIX™ kernels. A dispatching port is not needed with these instructions.

The **schedprcs** instruction causes a process to be enqueued at a dispatching port.

The **sendserv** instruction suspends the current process and sends a message to the specified communication port.

The **ldtime** instruction accesses the execution time of a process.

The **modpc** instruction reads and optionally modifies the contents of the process controls for the currently running process.

The **ldglobals** instruction reads a word from the process globals object of the current process.

4.16.2 Interprocess Communication

Two techniques are available for communication among processes: semaphores and communication ports.

4.16.2.1 Semaphores

wait	wait
condwait	conditional wait
signal	signal

Counting semaphores are supported for synchronization among processes. A semaphore contains a queue for waiting processes.

The **wait** instruction attempts to decrement the sempahore count. If the semaphore count is non-zero, the count is decremented, and the process continues execution. If the count is zero, the current process suspends and is queued to the semaphore. The process is then said to be blocked.

The **condwait** instruction performs the same function as the **wait** instruction, except that the process never blocks. Instead, the condition code is set to indicate whether or not operation is successful or not.

The **signal** instruction releases a semaphore by incrementing the semaphore count if there is no waiting process. Otherwise, the highest priority process is unblocked and rescheduled.

4.16.2.2 Ports

A port is similar to a semaphore except that a port also provides a message-passing mechanism. A port can be used both for synchronizing processes and as a means of passing messages among processes. Messages are objects and contain their own queuing space.

receive	receive
condrec	conditional receive
send	send
sendserv	send service

With the **receive** instruction, the specified port is checked for a message. If a message is queued at the port, the message is loaded into a specified register and the current process continues execution. If the message queue is empty, the the current process is suspended, and queued at the communication port, thus blocking the process.

The **condrec** instruction is similar to the **receive** instruction except that the process is not blocked if the message queue is empty. Instead the the condition code is set to indicate whether or not a message has been received.

The **send** instruction sends a message to a specified communication port. If there are no processes at the port waiting for messages, the message is queued at the port and the current process continues. If there are queued processes at the port, the first process in the queue is unblocked, given the message, and rescheduled at the dispatching port. The current process is then resumed.

FLOATING-POINT OPERATION 5

This chapter describes the floating-point processing capabilities. The subjects discussed include the real number data types, the execution environment for floating-point operations, the floating-point instructions, and fault and exception handling.

5.1 Introducing the Floating-Point Architecture

The floating-point architecture is designed to allow a convenient implementation of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985). This hardware architecture, along with a small amount of software support, conforms to the IEEE standard and provides support for the following data structures and operations:

- Real (32-bit), long real (64-bit), and extended real (80-bit) floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversion between integer and floating-point formats
- Conversion between different floating-point formats
- Handling of floating-point exceptions, including non-numbers (NaNs)

The software to support the floating-point architecture is needed primarily to handle conversions between real numbers and decimal strings.

In addition, the floating-point architecture supports several functions that go beyond the IEEE standard. These functions fall into two categories:

- functions recommended in the appendix to the IEEE standard, such as copy sign and classify, and
- commonly used transcendental functions, including trigonometric, logarithmic, and exponential functions.

5.2 Real Numbers and Floating-Point Format

This section provides an introduction to real numbers and how they are represented in floating-point format. Readers who are already familiar with numeric processing techniques and the IEEE standard may wish to skip this section.

5.2.1 Real Number System

As shown at the top of Figure 5-1, the real-number system comprises the continuum of real numbers from minus infinity (-∞) to plus infinity (+∞).

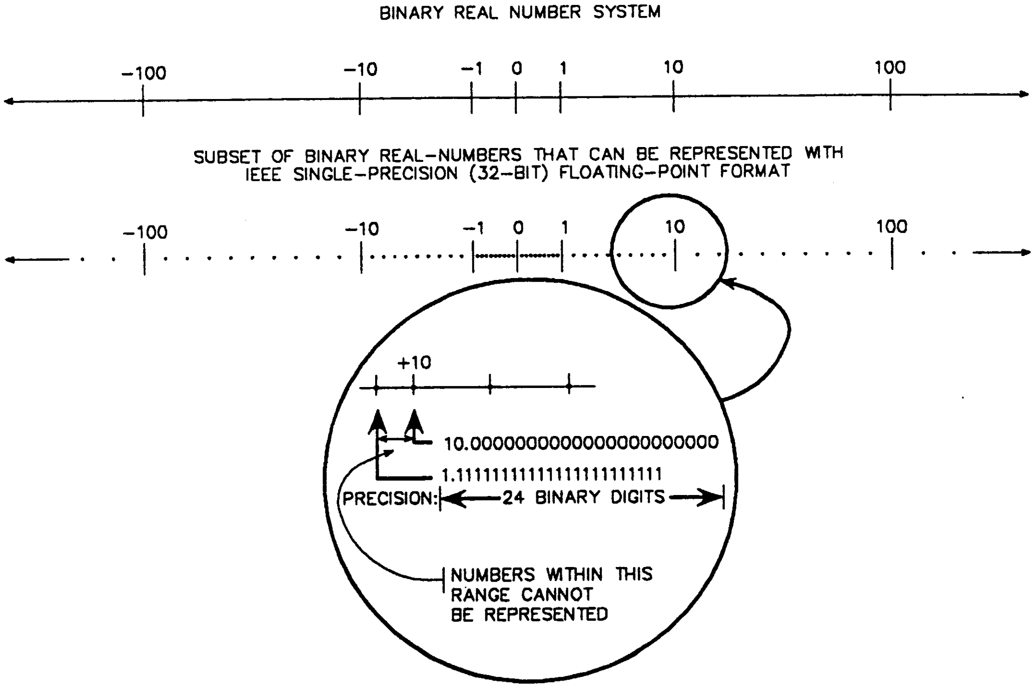

BINARY REAL NUMBER SYSTEM
-100
-10
-1
0
1
10
100
SUBSET OF BINARY REAL-NUMBERS THAT CAN BE REPRESENTED WITH IEEE SINGLE-PRECISION (32-BIT) FLOATING-POINT FORMAT
-100
-10
-1
0
1
10
100
+10
10.0000000000000000000000
1.11111111111111111111111
PRECISION: 24 BINARY DIGITS
NUMBERS WITHIN THIS RANGE CANNOT BE REPRESENTED

Figure 5-1. Binary Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 5-1, the subset of real numbers represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format used to represent real numbers.

5.2.2 Floating-Point Format

To increase the speed and efficiency of real number computations, real numbers are typically represented in a binary floating-point format. In this format, a real number has three parts: a sign, a significand, and an exponent. Figure 5-2 shows the binary floating-point format. This format conforms to the IEEE standard.

$$\text{real value} = (-1)^{\text{sign}} * \text{significand} * 2^{\text{exponent}}$$

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a one-bit binary integer (also referred to as the j-bit) and a binary fraction. The j-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power to which the significand is raised.

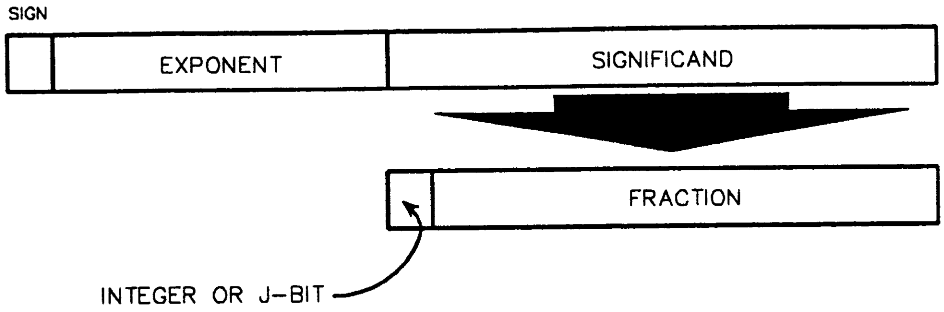


Figure 5-2. Binary Floating-Point Format

Table 5-1 shows how the real number 201.187 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the storage format. In this format, the binary real number is normalized and the exponent is biased.

Table 5-1. Real Number Notation

NOTATION	VALUE		
ORDINARY DECIMAL	201.187		
SCIENTIFIC DECIMAL	$2.01187E_{10}2$		
SCIENTIFIC BINARY	$1.1001001001011111E_{2}111$		
SCIENTIFIC BINARY (BIASED EXPONENT)	$1.1001001001011111E_{2}10000110$		
	SIGN	BIASED EXPONENT	SIGNIFICAND
32-BIT FLOATING-POINT FORMAT (NORMALIZED)	0	10000110	1001001001011111 ^ |_____1. (IMPLIED)

5.2.2.1 Normalized Numbers

In most cases, real numbers are represented in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and a fraction as follows:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 (inclusive) and 2 (exclusive) and an exponent that gives the number's binary point.

5.2.2.2 Biased Exponent

Exponents are represented in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. This allows two real numbers (of the same format and sign) to be compared as if they are unsigned binary integers. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

5.2.3 Real Number and Non-Number Encodings

The real numbers that are encoded in the floating-point format described above are generally divided into three classes: ± 0, ± nonzero-finite numbers, and ± ∞. Encodings for non-numbers (NaNs) are also defined. The term NaN stands for "Not a Number."

Figure 5-3 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term "s" indicates the sign bit, "e" the biased exponent, and "f" the fraction. (The exponent values are given in decimal.)

5.2.3.1 Signed Zeros

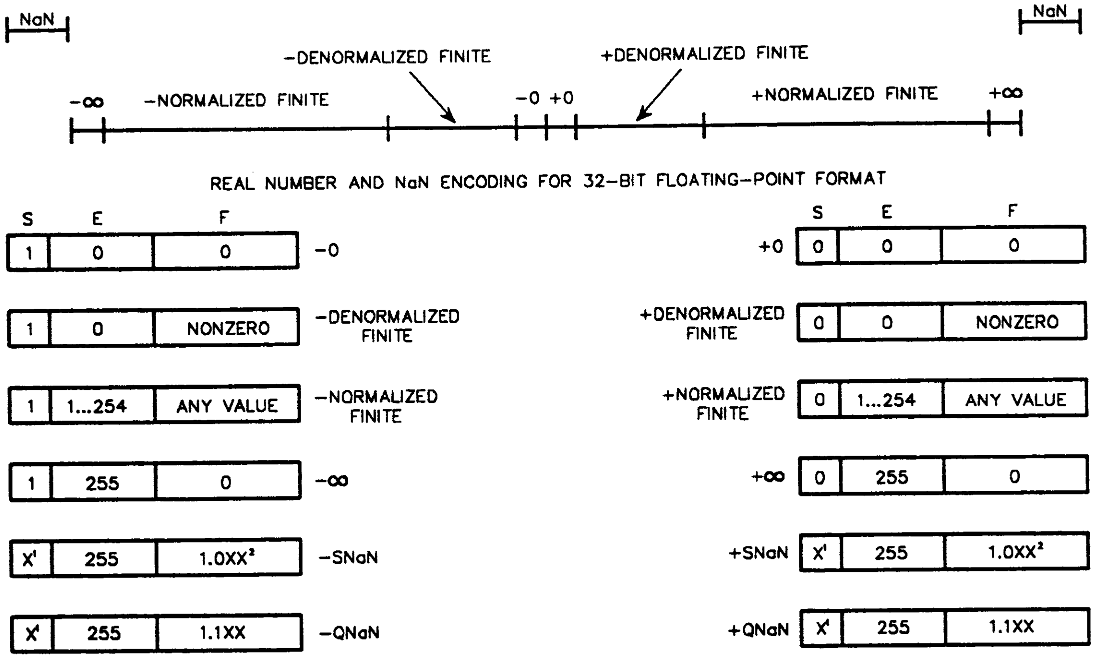
Zero can be represented as a +0 or a -0 depending on the sign bit. Both encodings are equal in value, but may produce different results depending on the operation. The sign of a zero result depends on the operation being performed and the rounding mode being used. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

5.2.3.2 Signed, Nonzero, Finite Values

The class of signed, nonzero, finite values is divided into two groups: normalized and denormalized. The normalized finite numbers comprise all the nonzero finite values that can be encoded in a normalized real number format. In the 32-bit form shown in Figure 5-3, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

5.2.3.3 Denormalized Numbers

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros. Denormalized numbers are used to represent values between 0 and the smallest normalized number.



Notes:
1. Sign bit ignored.
2. Fraction must be nonzero.

Figure 5-3. Real Numbers and NaNs

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called denormalized numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, operations are normally performed on normalized numbers and produce normalized numbers as results. Denormalized numbers represent an underflow condition.

A denormalized number is computed through a technique called gradual underflow. Table 5-2 gives an example of gradual underflow in the denormalization process. Here the 32-bit format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 5-2. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	-129	1.01011100...00
Denormalize	0	-128	0.101011100...00
Denormalize	0	-127	0.0101011100...00
Denormalize	0	-126	0.00101011100...00
Denormalize Result	0	-126	0.00101011100...00

*Expressed as unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

5.2.4 Signed Infinities

Arithmetic on infinities is defined as the limiting case of real arithmetic with operands of arbitrarily large magnitude. Negative infinity is less than every finite number. Positive infinity is greater than every finite number. Arithmetic on infinity is always exact and generates no exceptions except the cases noted in Section 5.10.3. Infinity is always represented by a zero fraction and the maximum biased exponent allowed in the specified format (e.g., 255_{10} for the 32-bit format).

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

5.2.5 NaNs

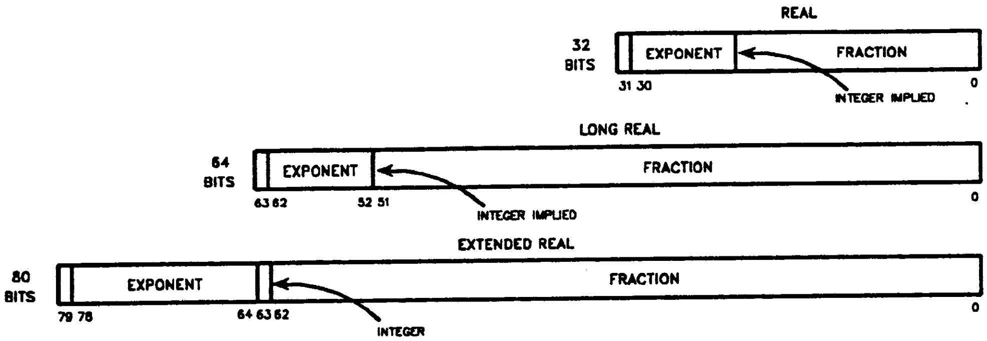
Since NaNs are non-numbers, they are not part of the real number line. In Figure 5-3, the encoding space for NaNs in the floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two specific NaN values: a quiet NaN (QNaN) and a signaling NaN (SNaN). A QNaN is a NaN with the most significant fraction bit set; a SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed in Section 5.10.

Section 5.9 provides detailed information on how the NaNs are handled.

5.3 Real Data Types

Three real-number data formats are supported: real, long real, and extended real. These formats correspond directly to the single-precision, double-precision, and double-extended precision formats in the IEEE standard. Figure 5-4 shows these data formats and gives the resolution that each provides.



DATA TYPE	RANGE
REAL	2^{-126} to 2^{127} ($\sim 10^{-45}$ to $\sim 10^{38}$)
LONG REAL	2^{-1022} to 2^{1023} ($\sim 10^{-324}$ to $\sim 10^{308}$)
EXTENDED REAL	2^{-16382} to 2^{16383} ($\sim 10^{-4950}$ to $\sim 10^{+4932}$)

Figure 5-4. Real Number Formats

For the real and long-real formats, only the fraction is given for the significand. The integer is assumed to be 1 for all numbers except 0 and denormalized finite numbers.

For the extended-real format, the integer is contained in bit 63, and the most-significant fraction bit is bit 62. Here, the integer is explicitly set to 1 for normalized numbers, infinities, and NaNs, and to 0 for zero and denormalized numbers.

Table 5-3 shows the encodings for all the classes of real numbers (that is, zero, denormalized finite, normalized finite, and ∞) and NaNs, for each of the three real data-types.

Table 5-3. Real Numbers and NaN Encodings

CLASS		SIGN	BIASED EXPONENT	INTEGER[1]	FRACTION
POSITIVE	+∞	0	11...11	1	
	+ NORMALS	0 . . 0	11...10 . . 00...01	1 . . 1	11...11 00...00
	+ DENORMALS	0 . . 0	00...00 . . 00...00	0 . . 0	11...11 . . 00..01
	+ ZERO	0	00...00	0	00..00
NEGATIVE	- ZERO	1	00...00	0	00..00
	- DENORMALS	1 . . 1	00...00 . . 00...00	0 . . 0	00...01 . . 11...11
	- NORMALS	1 . . . 1	00...01 . . . 11...10	1 . . . 1	00...00 11...11
	-∞	1	11...11	1	00...00
NaN	SNaN	X	11...11	1	0X...XX[2]
	QNaN	X	11...11	1	1X...XX

REAL:	←8 BITS→	←23 BITS→
LONG REAL:	←11 BITS→	←52 BITS→
EXTENDED REAL:	←15 BITS→	←63 BITS→

Notes:

1. Integer is implied for real and long real formats and is not stored.

2. Fraction for SNaN must be non-zero.

Where the value is listed as signed, s=0 represents positive, s=1 represents negative.

A real is a 32-bit binary floating-point number. Bit 31 is the sign (s), bits 23-30 are a biased exponent (e), and bits 0-22 are the fraction (f). The value of an occurrence of a real is as follows:

- If e=255 and f is nonzero, value is NaN, regardless of the sign.
- If e=255 and f = 0, value is signed infinity.
- If 0<e<255, value is signed (1.f*2**(e-127)).

- If e=0 and f is nonzero, value is signed (0.f*2**(-126)). This is called a denormalized number.
- If e=0 and f=0, value is signed zero.

A long_real is a 64-bit binary floating-point number. Bit 63 is the sign (s), bits 52-62 are the biased exponent (e), and bits 0-51 are the fraction (f). The value of an occurrence of a long_real data type is as follows:

- If e=2047 and f is nonzero, value is NaN, regardless of the sign.
- If e=2047 and f = 0, value is signed infinity.
- If 0<e<2047, value is signed (1.f*2**(e-1023)).
- If e=0 and f is nonzero, value is signed (0.f*2**(-1022)). This is called a denormalized number.
- If e=0 and f=0, value is signed zero.

An extended_real is an 80-bit binary floating-point number satisfying the requirement for the implementation-dependent "double-extended" type specified in the IEEE standard. Bit 79 is the sign (s), bits 64-78 are the biased exponent (e), bit 63 is the integer part (j), and bits 0-62 are the fraction (f). The value of an occurrence of an extended_real data type is as follows:

- If e=32767 and j=1 and f is nonzero, value is NaN, regardless of the sign.
- If e=32767 and j=1 and f=0, value is signed infinity.
- If 0<e<32767 and j=1, value is signed (j.f*2**(e-16383)).
- If e=0 and j or f is nonzero, value is signed (j.f * 2**(-16382)). This is called a denormalized number.
- If e=0 and j=0 and f=0, value is signed zero.
- If e is nonzero and j=0, value is a reserved encoding.

5.4 Execution Environment for Floating-Point Operations

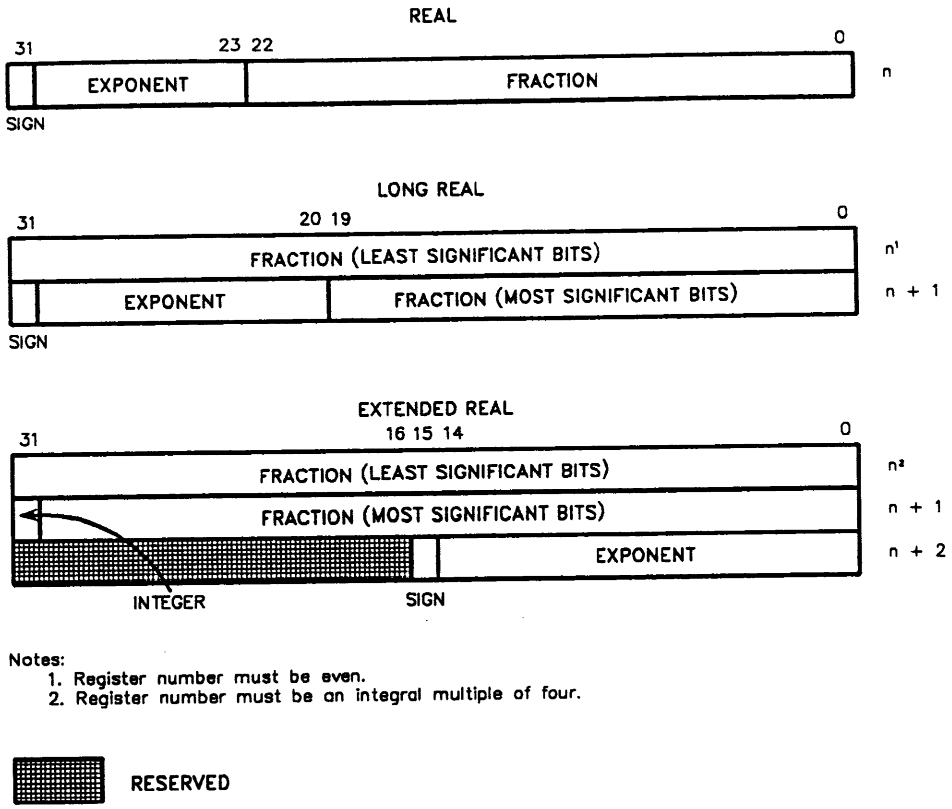
The floating-point processing capabilities are completely integrated into the execution environment. Operations on floating-point numbers are carried out using the same registers that are used for ordinals and integers. In addition, four floating-point registers have been provided for extended-precision floating-point arithmetic.

The following sections describe the handling of floating-point operations.

5.4.1 Registers

All of the global, local, and floating-point registers can be used for floating-point operations. When using global or local registers, real values (32 bits) are contained in one register, long-real values (64 bits) are contained in two successive registers, and extended-real values (80 bits) are contained in three successive registers.

Figure 5-5 shows how the three forms of the real data type are encoded when stored in global and local registers. Note that long-real values must be aligned on even-numbered register boundaries (g0, g2, ...). Extended-real values must be aligned on register boundaries that are an integral multiple of four (g0, g4, ...).



Notes:
1. Register number must be even.
2. Register number must be an integral multiple of four.

Figure 5-5. Storage of Real Values in Global and Local Registers

Real values in the floating-point registers are always in the extended-real format. When a real or long-real value is moved from global or local registers to a floating-point register, the value is automatically reformatted for the extended-real format.

5.4.2 Loading and Storing Floating-Point Values

Floating-point values are loaded from memory into global or local registers using the load (ld), load long (ldl), and load triple (ldt) instructions. Likewise, floating-point values in global or local registers are stored in memory using the store (st), store long (stl), and store triple (stt) instructions.

Loading an extended-real floating-point value into a floating-point register requires two steps (two instructions). First, the extended-real value must be loaded from memory into global or local registers. Then, the value must be moved to the floating-point register using a move extended-real (**movre**) instruction.

A similar two-step procedure is required to store an extended-real value from a floating-point register into memory. The value must first be moved into global or local registers (using a **movre** instruction), then stored in memory.

This two-step method for moving values from memory into floating-point registers and vice versa may seem a little cumbersome; however, in practice it generally is not. Floating-point registers are most often used to store and accumulate intermediate results of computations. The contents of these registers are not normally stored in memory.

EXECUTION ENVIRONMENT 6

This chapter describes the basic execution environment, how the processor executes instructions, and how the processor manipulates data. This chapter discusses the linear address space, the register model, the instruction pointer, and the arithmetic controls.

6.1 Overview of the Execution Environment

The environment when executing an instruction is shown in Figure 6-1. An execution environment consists of a 2^{32}-byte linear address space, a set of global and floating-point registers, an instruction pointer, and an arithmetic controls register.

An execution environment corresponds to a process-wide address space in other architectures. Multiple address spaces may be available to a process, but only one address space can be active at a time. Transfers from one address space to another address space are described in Chapter 7.

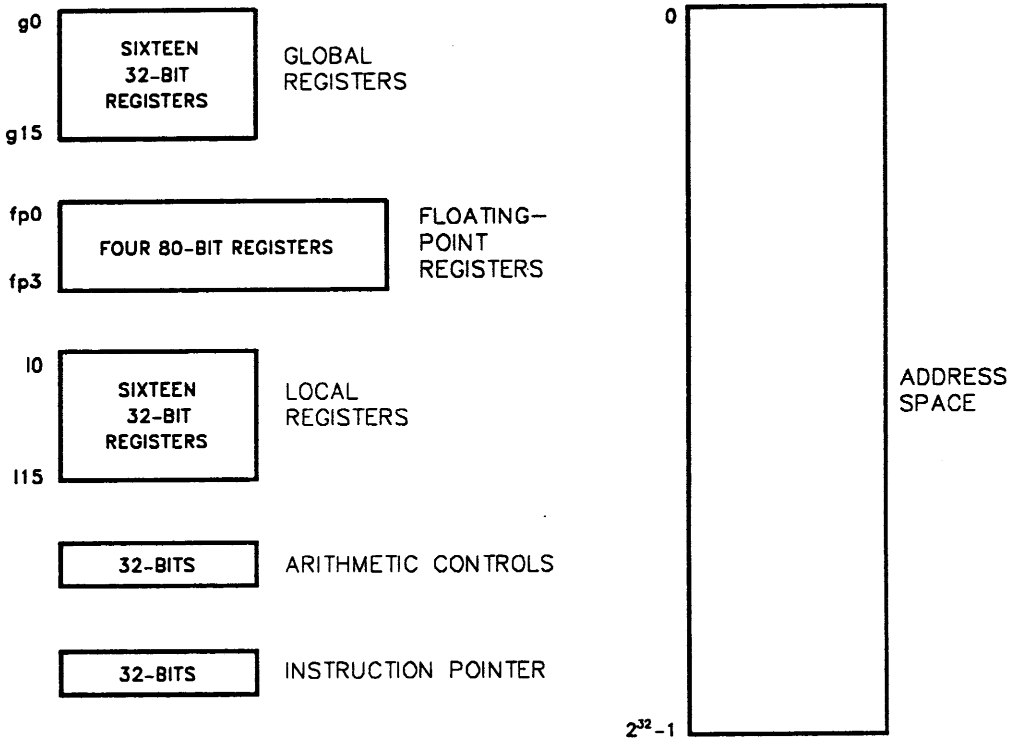


Figure 6-1. Execution Environment

6.2 Register Model

The register model consists of 16 global registers and 4 floating-point registers that are accessible across procedure boundaries, and 16 local (or frame) registers that created newly for each procedure.

At any instant, an instruction can address 36 of these registers as follows.

Register Type	Register Name
Global Register	G0 .. G15
Floating Point Register (floating-point operand)	FP0 .. FP3
Local Register	L0 .. L15

The global registers and local registers are collectively referred to as the "general registers". Some addressing modes refer to general registers, to distinguish them from the floating-point registers.

6.2.1 Global Registers

Each process has 16 associated global registers; they are saved in the process object (Chapter 15) when the process is not executing.

Of the 16 32-bit registers, g15 contains the current frame pointer (FP) and g0 through g14 are general-purpose registers. The FP contains the linear address of the current (topmost) stack frame. Since stack frames are aligned to 64-byte boundaries, the low-order 6 bits of FP are ignored and always interpreted to be zero. (The alignment of the FP and the number of ignored FP bits may change in future releases.) The FP is adjusted automatically upon each call and return, and should not otherwise be modified.

6.2.2 Floating Point Registers

Each process has four associated floating-point registers; they are saved in the process object (as described in Chapter 15) when the process is not executing.

Floating point numbers are stored in "extended real" format in the floating-point registers. Floating point registers are accessible as operands in floating-point instructions, but such instructions may also use the general registers.

6.2.3 Local (or Frame) Registers

Registers l0 through l15 (the local registers) are allocated on procedure calls and deallocated on returns.

Multiple banks of local registers are provided (four in this release, but future releases may provide a different number). When necessary, these registers are saved to and restored from the first 16 words of the stack frame, where register l0 is mapped into linear address FP+0 to FP+3, register l*i* is mapped into linear address FP+4*i* to FP+4*i*+3, and so on.

For most programs, the existence of the multiple register sets and the saving/restoring of them in the stack frames should be transparent. However, in some cases it may not be transparent. For example:

- When a stack frame is allocated as a result of a procedure call, the local registers are not necessarily cleared nor initialized from the memory values. Thus, the initial contents of a local register (other than those altered by the call operation) are unpredictable.

- The local registers are not associatively mapped into the frames; loading (or storing) a value from (or into) the first 16 words of a frame is not guaranteed to access (or modify) the associated register.
- A deallocated stack frame does not necessarily contain the local registers from the called procedure. Local registers are not necessarily flushed before a **ret** call.
- To access or modify the local registers of a previous frame, first precede the access or modification with a **flushreg** instruction. The **flushreg** instruction writes all register sets to their associated stack frames in memory. However, the current frame cannot be accessed in this way. Use register references instead.
- To modify the previous FP (in register l0), follow the modification with a **flushreg** instruction, or else the behavior of the **ret** instruction is not predictable.
- The current FP (in register g15) cannot be modified by writing into the register. Instead, a routine must be called that modifies its previous FP (in register l0), and then returns.

6.2.4 Register Alignment for Multiple Word Operands

An operand in an instruction ranges from 1 to 4 words. When multiple registers are needed for an operand, the lowest-numbered register is specified in the instruction, and additional consecutively higher registers are used as needed. The lowest-numbered register must be an even-numbered register (l0, l2, l4, and so on, or g0, g2, g4, and so on) if two registers are needed. Similarly, the lowest-numbered register must be a multiple of four register (l0, l4, l8, and so on, or g0, g4, g8, and so on) if three or four registers are needed. Failure to properly align either a source or destination will produce unpredictable results (including possibly a fault).

6.3 Instruction Pointer

The IP is the linear address (using the current linear address map) of the current instruction. Since instructions must begin on word (4-byte) boundaries, the two low-order bits of IP are presumed to be zero, and ignored.

6.4 Arithmetic Controls

The Arithmetic Controls (AC) controls the arithmetic and faulting properties of the numeric instructions, and retains the current condition codes. No faults are generated when the bits of the arithmetic controls are explicitly modified. When a process is suspended, the AC is saved in the process object.

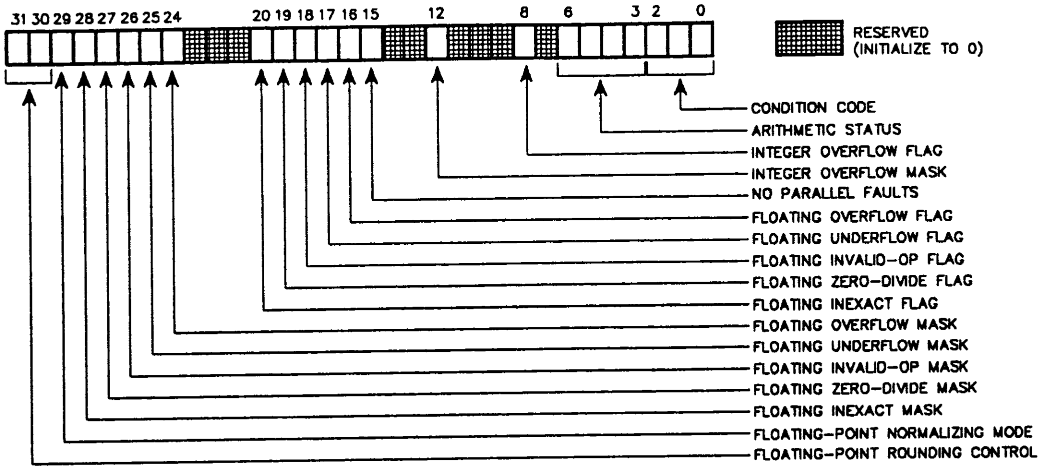


Figure 6-2. Arithmetic Controls

The AC contains the following information. All unused bits are reserved and should be set to zero.

- **Condition Code** (bits 0-2). A set of flags set by comparison (and other) instructions and examined by conditional-branch (and other) instructions.
- **Arithmetic Status** (bits 3-6). This field is altered as an indicator by certain floating-point instructions.
- **Integer Overflow Flag** (bit 8). This flag is set whenever an integer overflow occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Integer Overflow Mask** (bit 12). If set, an integer overflow does not generate an *Arithmetic* fault. If S is the destination size, the S least-significant bits of the result are stored in the destination unless otherwise noted.
- **No parallel faults** (bit 15). If set, faults are required to be synchronized. If clear, certain faults can be parallel. See Section 10.10.4.
- **Floating-point Overflow Flag** (bit 16). This flag is set whenever a floating-point overflow occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Floating-point Underflow Flag** (bit 17). This flag is set whenever a floating-point underflow occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Floating-point Invalid-op Flag** (bit 18). This flag is set whenever a floating-point invalid operation occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Floating-point Zero-divide Flag** (bit 19). This flag is set whenever a floating-point division by zero occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Floating-point Inexact Flag** (bit 20). This flag is set whenever a floating-point inexact result occurs and the mask is set. The flag is cleared only by explicit instructions.
- **Floating-point Overflow Mask** (bit 24). If set, a floating-point overflow does not generate a *floating-point* fault.
- **Floating-point Underflow Mask** (bit 25). If set, a floating-point underflow does not generate a *floating-point* fault.

- **Floating-point Invalid-op Mask** (bit 26). If set, a floating-point invalid operation does not generate a *floating-point* fault.
- **Floating-point Zero-divide Mask** (bit 27). If set, a floating-point division by zero does not generate a *floating-point* fault.
- **Floating-point Inexact Mask** (bit 28). If set, a floating-point inexact result does not generate a *floating-point* fault.
- **Floating-point Normalizing Mode** (bit 29). If set, denormalized numbers in reals, long reals or extended reals are first normalized before arithmetic is performed. If clear, denormalized numbers generate a *floating-point* fault.
- **Floating-point Rounding Control** (bits 31-30). This field indicates the rounding mode for floating-point computations:

00	round to nearest
01	round down (toward negative infinity)
10	round up (toward positive infinity)
11	truncate (round toward zero)

Chapter 5 contains further information about floating-point rounding.

6.5 Stack Frame

The stack frame is a contiguous portion of the current linear address space, containing data in a stack-like fashion. The stack grows from low addresses to high addresses. Each activated procedure has one stack frame. The stack frame contains local variables, parameters, and linkage information. A call operation acquires a new stack frame; a return operation releases it. When a new frame is acquired, it is aligned on a 64-byte boundary.

The page or the simple object into which the first 64 bytes of a frame are mapped must be of local lifetime. The lifetime of the page or the simple object is checked during a call. This restriction is also necessary to ensure efficient manipulation of ADs in the local registers. (See Chapter 8).

In addition to the requirement that a frame is mapped onto a local page or local simple object, the mixed bit of the page or the object descriptor is set even though no tag bit may be written to the frame. (Descriptors are discussed in Chapter 8.) This restriction is necessary to ensure efficient manipulation of ADs in the local registers.

The structure of a stack frame is shown in Figure 6-3.

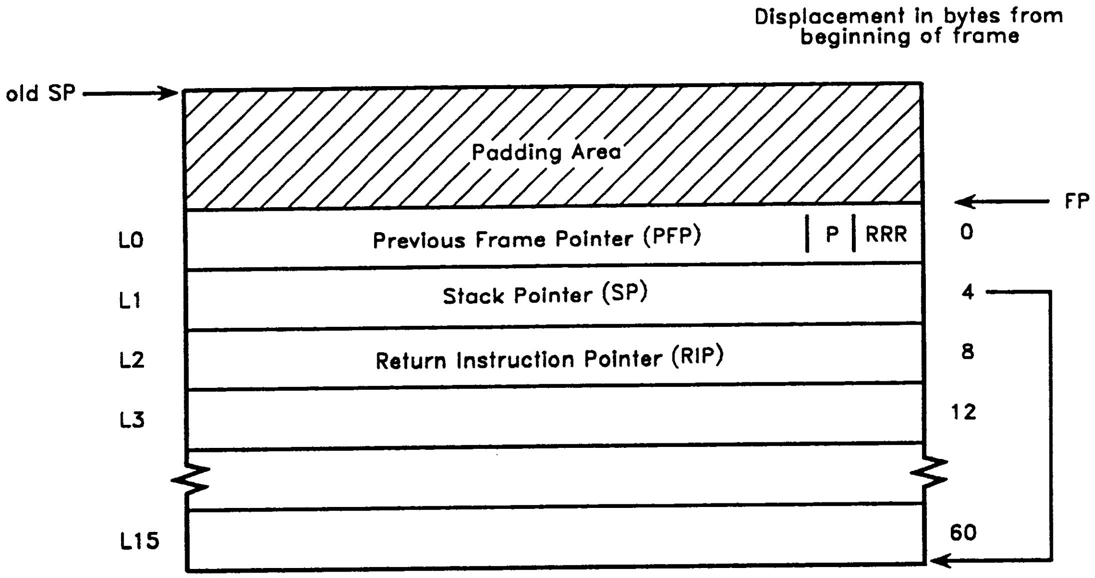

Displacement in bytes from beginning of frame
old SP
Padding Area
FP
L0
Previous Frame Pointer (PFP)
P
RRR
0
L1
Stack Pointer (SP)
4
L2
Return Instruction Pointer (RIP)
8
L3
12
L15
60

Figure 6-3. Stack Frame Structure

The fields in the stack frame are defined as follows:

- **Padding Area.** This area is used to align the FP to the next 64-byte boundary. The size of this area varies from 0 to 63 bytes. When a call operation is performed, a padding area is added to round the caller's SP to the next boundary to form the FP for this frame. If the caller's SP is already aligned, the padding area is absent.
- **Frame Status (L0).** The frame status records the information associated with the frame, after a call, to be used on a return from the frame. The fields of a frame status are defined as follows:
  - **Return Status,** RRR (bits 0-2). This 3-bit field records the call mechanism used in the creation of this frame and is used to select the the return mechanism to be used on return. The encodings of this field are as follows:

000	Local
001	Fault
010	Supervisor, trace was disabled before call
011	Supervisor, trace was enabled before call
100	Subsystem (intrasubsystem)
101	Subsystem (intersubsystem)
110	Idle interrupt
111	Interrupt

  - **Prereturn Trace,** P (bit 3). On a return from a frame when the prereturn trace bit is 1, a prereturn trace event (if enabled) occurs before any actions association with the return operation is performed. This bit is initialized to 1 on a call if a call-trace event occurred; otherwise it is initialized to 0.
  - **Previous Frame Pointer,** PFP (bits 4-31). The most-significant 28 bits of the linear address of the first byte of the previous frame. Since frames are aligned to boundaries of 64 bytes or more, the lower two bits of this field (bits 4 and 5) are always zero.

On all returns, the PRRR bits are interpreted as follows:

1xxx Generate a prereturn trace
0000 Perform a local return
0001 Perform a fault return (see Chapter 10).
001T In supervisor mode, perform a supervisor return (see Chapter 7). The T bit is assigned to the trace enable bit in the process controls, and the execution mode bit is set to user. Otherwise, perform a local return.
010x Perform a subsystem return (see Chapter 7).
011x Perform an interrupt return (see Chapter 16).

- **Stack Pointer,** SP (L1). A linear address to the first free byte of the stack, that is, the address of the last byte in the stack plus one. SP is initialized by the call operation to point to FP plus 64.
- **Return Instruction Pointer,** RIP (L2). When a call operation is performed to a new frame (via an instruction, interrupt, or fault), the return IP is saved here. When the process is suspended, the instruction pointer of the next instruction is stored here. The RIP contains a 32-bit linear address to which control is returned after a return to this frame. Since the RIP can be modified by implicit calls and by certain instructions implicitly, programs should never use this register for other purposes.

6.6 Linear Address Space

The linear address space is partitioned into four regions as shown in Figure 6-4. Each region is defined by an object. The first three regions of an address space are specific to the current process (defined by the process object). The composition of the process-specific regions can be changed by a subsystem call/return (see Chapter 7). The fourth region of an execution environment is specific to the processor (defined by the processor object), and thus is shared by all processes. Instructions, stack frames, or data may be located anywhere in the linear address space. However, the operating system may require a particular partitioning. A recommended partitioning (as used in BiiN™) is described in Section 7.3.1.

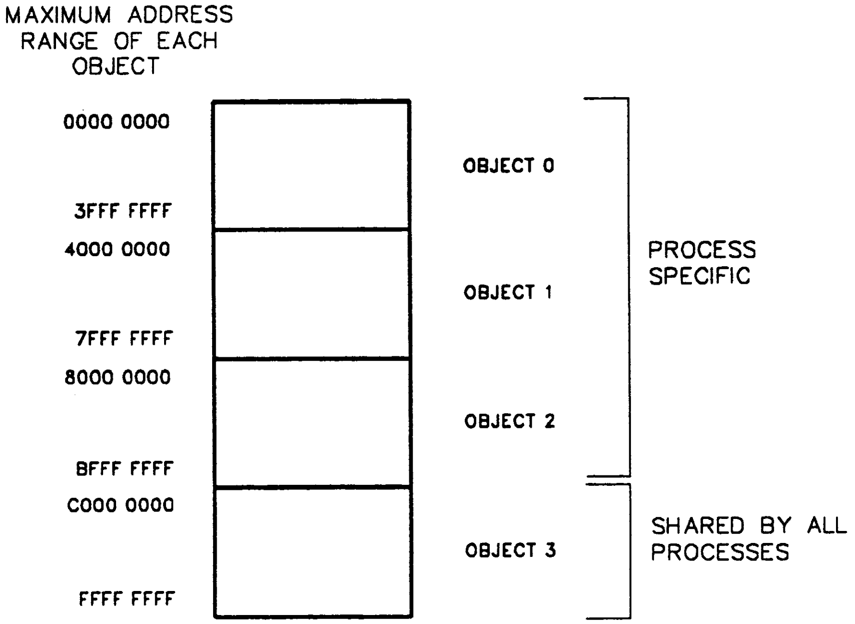


Figure 6-4. Linear Address Space

6.6.1 Region Objects

The ADs to the four regions are always interpreted to have read-write rep rights. It is not possible to protect each region differently using AD rep rights. Page-level protection can be used to achieve finer grain protection.

When an operand spans across region boundaries, the behavior is unpredictable.

Each region can be changed independently. If the region object is less than 1G bytes, a gap occurs at the end of the region. A simple object may be used to define a region if the object is 4K bytes and page aligned.

When a process is executing, all four regions must be unique. The AD for each region must have a different object index. Thus, linear address aliases are not allowed nor supported.

If the fault handler for virtual-memory faults (Chapter 10) is not a subsystem fault handler, the OTEs of the current regions must be valid for any process in the executing, ready, or blocked state (see Chapter 15). If the fault handler for virtual-memory faults is a subsystem fault handler, the OTEs of the regions in that subsystem must be valid.

6.6.2 Instruction Protection

Only read rights are necessary to fetch and execute instructions. Instruction pages should be write-protected to prevent accidental damage to the instructions.

6.6.3 Instruction Caching

The instruction stream may be non-transparently cached. Instruction caching is independent of the cacheable bit (see Chapter 8) in the page where instructions are located. Self-modifying programs may not necessarily work unless the instruction cache is purged. An IAC message (as described in Chapter 16) may be used to purge the contents of an instruction cache.

6.7 Local Procedure Mechanism

A procedure may begin at any arbitrary word address in a linear address space. Since instructions are fetched in blocks, it may be more efficient if the first instruction is aligned to a quad-word boundary. Procedure calls use a stack in the linear address space, as shown in Figure 6-5.

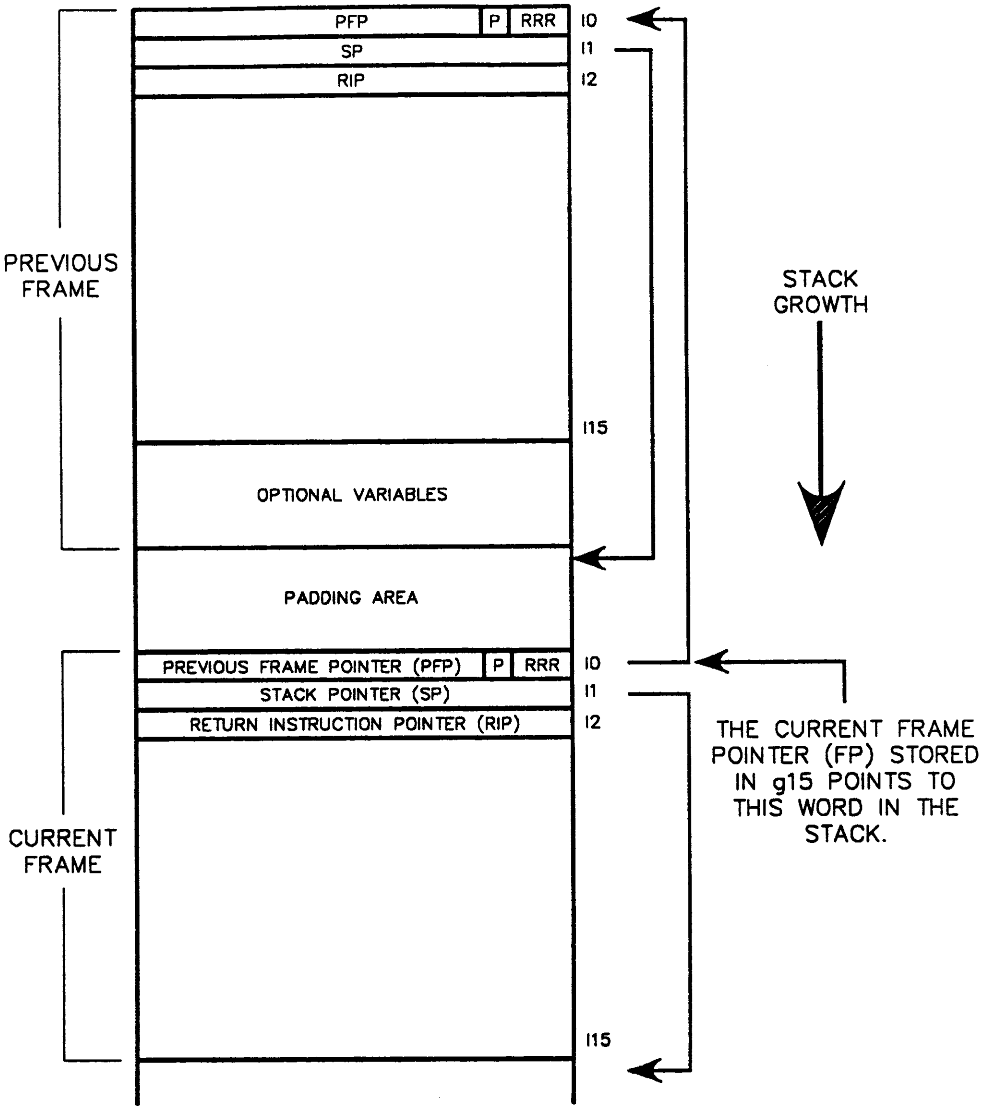

PFP
P
RRR
SP
RIP
l0
l1
l2
l15
PREVIOUS FRAME
OPTIONAL VARIABLES
PADDING AREA
STACK GROWTH
PREVIOUS FRAME POINTER (PFP)
P
RRR
STACK POINTER (SP)
RETURN INSTRUCTION POINTER (RIP)
l0
l1
l2
l15
CURRENT FRAME
THE CURRENT FRAME POINTER (FP) STORED IN g15 POINTS TO THIS WORD IN THE STACK.

Figure 6-5. Call Stack in Execution Environment

Two parameter passing mechanisms are suggested:

1. **Global Registers.** Parameters are copied to the global registers by the caller and copied out (if necessary) of the global registers by the callee after the call. Return or result parameters are copied to the global registers by the callee and copied out of the global registers by the caller after a return. This is optimized for procedures with a small number of parameters.
2. **Argument List.** An argument pointer to an argument list on the stack is used. This is an escape mechanism when there are more parameters than can be passed using global registers.

6.8 Instructions

6.8.1 Local Call and Return

call
callx
ret

The **call** and **callx** instructions invoke the procedure at the specified address. **call** specifies the procedure as the current IP plus a 24-bit signed displacement. **callx** specifies the procedure using a general address.

A new stack frame is allocated during the call operation and the control flow is transferred to the specified procedure.

The **ret** instruction transfers control back to the calling procedure and releases the called procedure's stack frame. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's frame.

6.8.2 Miscellaneous Instructions

modac
flushreg
cvtadr

The **modac** instruction reads or modifies the current arithmetic controls. The **flushreg** instruction writes all local register sets except for the current one into their associated frames in memory. The **cvtadr** instruction converts a current linear address into its corresponding virtual address.

For example, the following instruction

```
divr l3, l4, fp2
addr l5, fp2, l6
```

causes the real value in local register l4 to be divided by the value in l3, with the extended-real result stored in floating-point register fp2. Here, a move operation from the local registers to the floating-point registers is not required, since it is implicit in the divide operation. Similarly, a move operation from the fp2 to local register is not required because it is implicit in the add operation.

5.4.3 Moving Floating-Point Values

Either the move instructions (**mov, movl,** or **movt**) or the move-real instructions (**movr, movrl,** or **movre**) can be used to move real values among global and local registers. The move real instructions are generally used to convert a real value from one format to another or for moving real values between the global or local registers and floating-point registers. The move instructions are used to move real values while keeping them in the same format.

When using the **movr** and **movrl** instructions to move floating-point numbers between the global or local registers and the floating-point registers, the values are converted automatically from real and long-real format, respectively, into the extended-real format and vice versa.

For example, the following instruction

```
movr g3, fp1
```

causes a 32-bit, real value in global register g3 to be converted to 80-bit, extended-real format and placed in floating-point register fp1.

Going the opposite direction, the instruction

```
movrl fp0, l4
```

causes an extended-real value in floating-point register fp0 to be converted to 64-bit, long-real format and placed in local registers l4 and l5.

The **movre** instruction moves 80-bit, extended-real values between registers, without format conversion. When this instruction is used to move a value from three global or local registers to a floating-point register, the 80-bit value is extracted from the three word extended-real format. When moving a value from a floating-point register to global or local registers, the 80-bit value is inserted into the three registers in the three-word format.

5.4.4 Arithmetic Controls

The arithmetic controls are used extensively to control the arithmetic and faulting properties of floating-point operations. Table 5-4 shows the bits in the arithmetic controls that are used in floating-point operations. See Section 6.4 for the encodings of these fields.

Table 5-4. Arithmetic Controls Used in Floating-Point Operations

Arithmetic Control Bits	Function
0 - 2	Condition Code
3 - 6	Arithmetic status field
8	Integer overflow flag
12	Integer overflow mask
16	Floating overflow flag
17	Floating underflow flag
18	Floating invalid-operation flag
19	Floating zero-divide flag
20	Floating inexact flag
24	Floating overflow mask
25	Floating underflow mask
26	Floating invalid-operation mask
27	Floating zero-divide mask
28	Floating inexact mask
29	Normalizing mode flag
30 - 31	Rounding control

The condition code flags are used to indicate the results of comparisons of real numbers, just as they are for integers and ordinals.

The arithmetic status field is used to record results from the classify real (**classr** and **classrl**) and remainder real (**remr** and **remrl**) instructions. These instructions are discussed later in this chapter.

The floating-point flags indicate exceptions to floating-point operations. Here, the term exception refers to a potentially undesirable operation (such as dividing a number by zero) or an undesirable result (such as underflow). The flags provide a means of recording the occurrence of specific exceptions.

The floating-point masks provide a method of inhibiting the invocation of a fault handler when an exception is detected.

Use of the floating-point flag and mask bits are discussed in Section 5.10.

5.4.5 Normalizing Mode

The normalizing-mode flag specifies whether floating instructions operate in normalizing mode (set) or not (clear).

Normalizing mode is the most common mode of operation. Here, the operations are performed on valid floating-point operands, regardless of whether they are normalized or denormalized values.

When not operating in normalizing mode, a reserved-encoding exception is signaled whenever denormalized floating-point value is encountered as a source operand. In either mode, denormalized numbers are be produced if the underflow exception is masked.

There are no flag or mask bits in the arithmetic controls for this exception. When a reserved-encoding exception is detected, a floating reserved-encoding fault is generated and the destination operand is left unchanged.

The unnormalized mode of operation is provided to allow unnormalized arithmetic to be simulated with software. Here, a fault handler routine can be used to perform unnormalized arithmetic whenever a reserved-encoding exception is signaled.

5.4.6 Rounding Control

Often the infinitely precise result of an arithmetic operation cannot be encoded exactly in the format of the destination operand. For example, the following value has a 24-bit fraction. The least-significant bit of this fraction (the underlined bit) cannot be encoded exactly in the real (32-bit) format:

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 011\underline{1}E_2\ 101$$

This result must then be rounded to one of the following two values:

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 011E_2\ 101$$

$$1.0001\ 0000\ 1000\ 0011\ 1001\ 100E_2\ 101$$

A rounded result is called an inexact result. When an inexact result is produced, the floating-point inexact flag bit in the arithmetic controls is set.

Results are rounded according to the destination format (real, long real, or extended real) and the setting of the rounding-mode flags of the arithmetic controls. Four types of rounding are allowed, as described in Table 5-5.

Table 5-5. Rounding Methods

Rounding Mode	Description
Round up (toward +∞)	Rounded result is close to but no less than the infinitely precise result.
Round down (toward -∞)	Rounded result is close to but no greater than the infinitely precise result.
Round toward zero (Truncate)	Rounded result is close to but no greater in absolute value than the infinitely precise result.
Round to nearest (even)	Rounded result is close to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero).

When the infinitely precise result is between the largest positive finite value allowed in a particular format and +∞, the result is rounded as shown in Table 5-6.

Table 5-6. Rounding of Positive Numbers

Rounding Mode	Description
Round up (toward +∞)	+∞
Round down (toward -∞)	Maximum, positive finite value.
Round toward zero (Truncate)	Maximum, positive finite value.
Round to nearest (even)	+∞

When the infinitely precise result is between the largest negative finite value allowed in a particular format and -∞, the result is rounded as shown in Table 5-7.

Table 5-7. Rounding of Negative Numbers

Rounding Mode	Description
Round up (toward +∞)	Minimum, negative finite value.
Round down (toward -∞)	-∞
Round toward zero (Truncate)	Minimum, negative finite value.
Round to nearest (even)	+∞

The rounding modes have no effect on comparison operations, operations that produce exact results, or operations that produce NaN results.

5.4.7 Rounding Precision

Results are rounded according to the destination format (or precision). Since real, long real, and extended real destinations are allowed, no rounding precision mode is necessary. It is feasible to mimic all possible combinations of operand precision without suffering more than one rounding error.

The floating-point instructions allow a result to be stored in a shorter destination than the source operands. For example, the instruction

```
        addr fp1, fp2, g5
```

produces a real (32-bit) result from two extended-real (80-bit) source operands. In all such operations, only one rounding error occurs: the error that occurs when rounding the infinitely precise result to the size of the destination format.

Technically, an operation which computes a narrow result from wide operands is in violation of the IEEE standard. However, systems that are designed to conform to the IEEE standard do not need to use this capability.

5.5 Instruction Format

The instruction format for floating-point instructions is the same as for the other instructions. When programming in assembly language, an assembly language statement begins with an instruction mnemonic and is followed by from one to three operands. For example, the multiply-real instruction **mulr** might be used as follows:

```
        mulr 18, 19, fp3
```

Here, real operands in local registers l8 and l9 are multiplied together and the result is stored in floating-point register fp3.

From the machine level point of view, all floating-point instructions use the REG format. Refer to Chapter 17 for details on the REG format instructions.

5.6 Instruction Operands

Floating point operands for floating-point instructions can be either floating-point literals or registers. Two encodings are recognized for floating-point literals: +0.0 and +1.0.

A number of floating-point instructions contain both floating-point operands and integer/ordinal operands.

All of the general purpose registers (global registers, local registers, and floating-point registers) can be used as operands in floating-point instructions.

When general purpose registers are specified as operands, the instruction mnemonic (or opcode) determines how the values in these registers are interpreted. For example, there are two floating-point divide instructions: divide real (**divr**) and divide long real (**divrl**). When using the **divr** instruction, global- or local-register operands contain real (32-bit) values. When using the **divrl** instruction, global- or local-register operands contain long-real (64-bit) values. Long real and extended real operands need to satisfy the register alignment requirement as defined in Section 6.2.4. With either instruction, floating-point registers (containing extended-real values) can also be used as operands.

5.7 Mixed-Precision Arithmetic

Using floating-point registers as operands allows mixed format or mixed precision arithmetic to be performed with either real and extended-real values or long-real and extended-real values. Mixed-format operations with real and long-real values are not supported, but can be implemented with a sequence of two instructions without introducing extra rounding error.

```
subr   g0,  g1,  g2

subr   g0, fp1,  g2
subr  fp0,  g1,  g2
subr  fp0, fp1,  g2

subr   g0,  g1, fp2
subr   g0, fp1, fp2
subr  fp0,  g1, fp2
subr  fp0, fp1, fp2
```

A single **subr** instruction can be viewed as eight different instructions. Seldomly is it necessary to explicitly convert from one floating-point format to another in expression evaluations.

5.8 Summary of Floating-Point Instructions

Floating-point instructions consist of all instructions for which as least one operand is a real data type.

These instructions can be divided into the following groups:

- Data Movement
- Data Type Conversion
- Basic Arithmetic
- Comparison and Classification
- Trigonometric
- Logarithmic and Exponential

The following sections give a brief overview of the instructions in each group. Detailed descriptions of the operations of these instructions are given in Chapter 18.

5.8.1 Data Movement

The non-floating-point load and store instructions are used to move real values between registers and memory. Once in registers, the non-floating-point move instructions (**mov**, **movl**, and **movt**) are used to move real values between global and local registers without format conversion; whereas, the floating-point move instructions (**movr**, **movrl**, and **movre**) are used to move real values between global and local registers and floating-point registers.

The copy-sign real extended (**cpysre**) and copy-reverse-sign real extended (**cpyrsre**) instructions provide a means of copying the sign of one extended-real value to another, if one of the values is in a floating-point register. This operation is best performed on real and long-real values using the bit instructions **chkbit** and **alterbit**.

All these instructions, with the exception of **movr** and **movre**, are non-arithmetic operations. It is possible for **movr** and **movrl** to generate a real arithmetic fault even if the source and destination operands are of the same format. Real arithmetic faults can be avoided if **mov** and **movl** are used instead.

5.8.2 Data Type Conversion

Two types of data type conversions are provided: conversion from one floating-point format to another (for example, real to extended real) and conversion between integer and real.

Conversion between floating-point formats is handled in either of two ways: explicitly by move real instructions or implicitly by using the floating-point registers as operands in floating-point instructions.

As described earlier in this chapter, the **movr** instruction implicitly converts values from real to extended real, and vice versa, when moving values between global or local registers and floating-point registers. Likewise, the **movrl** instruction implicitly converts values from long real to extended real, and vice versa.

Conversion between real and long-real formats requires the use of both instructions. For example, the following two instructions convert a real value in global register g6 to a long-real value contained in g6 and g7, using a floating-point register for intermediate storage of the value:

```
movr g6, fp1
movrl fp1, g6
```

Implicit format conversion is also provided through the arithmetic, trigonometric, logarithmic, and exponential instructions. For example, the instruction

```
addr l4, l5, fp2
```

adds two real values together and produces an extended-real result.

The following six instructions allow conversion between integers and reals:

cvtir	convert integer to real
cvtilr	convert long integer to long real
cvtri	convert real to integer
cvtril	convert real to long integer
cvtzri	convert truncated real to integer
cvtzril	convert truncated real to long integer

Both the **cvtir** and **cvtilr** instructions can be used to convert an integer to an extended-real value by specifying that the result be placed in a floating-point register.

The convert real-to-integer instructions round off the real value to the nearest integer or long-integer value. For the **cvtri** and **cvtril** instructions, the rounding mode determines the direction the real number is rounded. For the convert truncated real-to-integer instructions (**cvtzri** and **cvtzril**), rounding is always toward zero. The latter two instructions are provided to allow efficient implementation of FORTRAN-like truncation semantics.

Extended-real values can be converted to integers by using a floating-point register as a source operand in either of the convert real-to-integer instructions.

Converting long-real values to integers requires two instructions, as in the following example:

```
movrl g6, fp3
cvtzri fp3, g6
```

The first instruction moves the long-real value to a floating-point register. The second instruction converts the extended-real value to an integer.

5.8.3 Basic Arithmetic

The following instructions perform the basic arithmetic operations specified in the IEEE standard:

addr	add real
addrl	add long real
subr	subtract real
subrl	subtract long real
mulr	multiply real
mulrl	multiply long real
divr	divide real
divrl	divide long real
remr	remainder real
remrl	remainder long real
roundr	round real
roundrl	round long real
sqrtr	square root real
sqrtrl	square root long real

The round instructions round the floating-point operand to its nearest integral value, based on the current rounding mode. These instructions perform a function similar to the convert real-to-integer instructions except that the result is in floating-point format. The remainder real instructions are not the same as defined by the IEEE standard. The IEEE remainder can be implemented easily using the provided instructions.

5.8.4 Comparison, Branching, and Classification

Comparison of floating-point values differs from comparison of integers or ordinals because with floating-point values there are four, rather than the usual three, mutually exclusive relationships: less than, equal to, greater than, and unordered.

The unordered relationship is true when at least one of the two values being compared is a NaN. This additional relationship is required because, by definition, NaNs are not numbers, so they cannot have greater than, equal, or less than relationships with other floating-point values.

The following instructions are provided for comparing floating-point values:

cmpr	compare real
cmprl	compare long real
cmpor	compare ordered real
cmporl	compare ordered long real

All of these instructions set the condition code flags in the arithmetic controls to indicate the results of the comparison. With the compare instructions (**cmpr** and **cmprl**), the condition code flags are set to 000_2 for the unordered condition. With the compare ordered instructions (**cmpor** and **cmporl**), the condition code flags are set to 000_2 and an invalid-operation exception is signaled for the unordered condition.

classr	classify real
classrl	classify real

Two branch instructions (**bo** and **bno**) allow conditional branching to be perfomed on an ordered or unordered condition, respectively.

The classify-real instructions (**classr** and **classrl**) provide a means of determining the class of a floating-point value (zero, denormalized finite, normalized finite, ∞, SNaN, or QNaN). The result of this operation is stored in the arithmetic status field of the arithmetic controls.

5.8.5 Trigonometric

The following instructions provide four common trigonometric functions:

sin	sine real
sinrl	sine long real
cosr	cosine real
cosrl	cosine long real
tanr	tangent real
tanrl	tangent long real
atanr	arctangent real
atanrl	arctangent long real

The arctangent instructions facilitate conversion from rectangular to polar coordinates.

5.8.6 Pi

The following value for π is used in computations:

$$\pi = f * 2^2$$

where:

$$f = 0.\mathrm{C90FDAA2\ \ 2168C234\ \ C}_{16}$$

(The spaces in the fraction above indicate 32-bit boundaries.)

This value has a 66-bit mantissa, which is 2 bits more than is allowed in the significand of an extended-real value.

The extra 2 bits of precision in the internal π avoids the singularities that occur in some trigonometric and complex functions. Since π/2 or its multiples cannot be exactly represented in the input operand, the tangent instructions never have to handle the singularity case at those points. Additionally, with the exception of sine of zero, the sine and cosine instructions never return zero.

If the results of computations that explicitly use π are to be used in the sine, cosine, or tangent instructions, the full 66-bit fraction for π should be used. This insures that the results are consistent with the argument-reduction algorithms that these instructions use. Using a rounded version of π can cause inaccuracies in result values, which if propagated through several calculations, might produce meaningless results.

A common method of representing the full 66-bit fraction of π is to separate the value into two numbers. For example, the following two long-real values (in long real format) added together give the value for π shown above with the full 66-bit fraction:

$$\pi = \text{high}\pi + \text{low}\pi$$

where:

$$\text{high}\pi = \text{400921FB}\ \ \text{54400000}_{16}$$

$$\text{low}\pi\ = \text{3DD0B461}\ \ \text{1A600000}_{16}$$

Here *highπ* gives the most significant 33 bits of π and *lowπ* gives the least significant 33 bits. Similar versions of π can also be written in the extended-real format.

When using this two-part π value in an algorithm, parallel computations should be performed on each part, with the results kept separate. When all the computations are complete, the two results can be added together to form the final result.

5.8.7 Logarithmic, Exponential, and Scale

The following instructions provide three different logarithmic functions, an exponential function, and a scale function:

logbnr	log binary real
logbnrl	log binary long real
logr	log real
logrl	log long real
logepr	log epsilon real
logeprl	log epsilon long real
expr	exponent real
exprl	exponent long real
scaler	scale real
scalerl	scale long real

These instructions are described in detail in Chapter 18. The following is a brief description of their functions.

The log binary instructions compute the IEEE recommended function *logb (X)* which returns the unbiased exponent of X.

The log instructions compute the function $Y * log_2 (X)$, where the log of X is the base-2 logarithm.

The log epsilon instructions compute the function $Y * log_2 (X + 1)$, where the log of X + 1 is a base-2 logarithm.

The exponent instructions compute the value $2^X - 1$.

The scale instructions perform a multiplication of a floating-point value by a power of 2.

5.8.8 Arithmetic Versus Nonarithmetic Instructions

The floating-point instructions can be divided into two groups: arithmetic and nonarithmetic. Nonarithmetic instructions are those which do not generate the invalid operation faults for signalling NaNs.

The five nonarithmetic instructions are move-real extended (**movre**), copy-sign real extended (**cpysre**), copy-reversed-sign real extended (**cpyrsre**), and classify real (**classr** and **classrl**). These nonarithmetic instructions are insensitive to real values and cannot generate any floating-point faults, including the floating reserved encoding faults. The **cmpr** and **cmprl** instructions also does not generate the IEEE floating-point faults, but do generate floating reserved encoding faults. Hence, the IEEE recommended functions, finite(x) and isnan(x), can be implemented as nonarithmetic.

This distinction between arithmetic and nonarithmetic instructions is important because floating-point exceptions and faults can be signaled only during the execution of arithmetic instructions.

5.9 Operations on NaNs

The two types of NaNs are SNaN and QNaN. A SNaN is any NaN value with its most-significant fraction bit set to 0 and at least one other fraction bit set to 1. (If all the fraction bits are set to 0, the value is an ∞.) A QNaN is any NaN value with the most-significant fraction bit set to 1. The sign bit of a NaN is not interpreted.

In general, when a QNaN is used in one or more arithmetic floating-point instructions, it is allowed to propagate through a computation. An SNaN on the other hand causes a floating invalid-operation exception to be signaled.

The floating invalid-operation exception has a flag and a mask bit associated with it in the arithmetic controls. The mask bit determines how an SNaN value is handled. If the floating invalid-operation mask bit is set, the SNaN is converted to a QNaN by setting the most significant fraction bit of the value to a 0. The result is then stored in the destination and the floating invalid-operation flag is set. If the invalid operation mask is clear, a floating invalid-operation fault is signaled and no result is stored in the destination.

When the result is a QNaN, the format of the result is as shown in Table 5-8, depending on the form of the source operands.

Table 5-8. Format of QNaN Results

Source Operands	QNaN Result
Only one operand is NaN, destination is same width	QNaN version of NaN source
Only one operand is NaN, destination is longer	QNaN version of NaN source, with fraction extended with zeros
Only one operand is NaN, destination is shorter	QNaN version of NaN source, with fraction truncated

Table 5-8: Format of QNaN Results (cont.)

Source Operands	QNaN Result
Both operands are NaNs	QNaN version of source whose fraction field has greatest magnitude, with fraction extended or truncated as described above

In some cases, a QNaN result is returned when none of the source operands are NaNs. Here, a standard QNaN is returned. The significand for the standard QNaN is as follows:

1.1000...00

(For real and long-real destinations, the integer bit will be an implied 1.)

Other than the rules specified above, software is free to use the other bits of a NaN for any purpose.

5.10 Exceptions and Fault Handling

Occasionally, a floating-point instruction can result in one of the six following floating-point exceptions being signaled:

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

These exceptions can be divided into two categories:

1. Situations in which one or more source operands are inappropriate for an operation and would cause an exception to be signaled.
2. Situations in which the result of an operation is exceptional.

The reserved encoding, invalid operation, and division-by-zero exceptions fall in the first category; the overflow, underflow, and inexact exceptions fall in the second category.

Except for the floating reserved-encoding exception, each of these exceptions has a flag and a mask bit associated with it in the arithmetic controls. When an exception condition occurs, the one of the following actions is taken:

- If the mask bit for the exception is set, the flag for the exception is set and instruction execution continues, substituting a default value in place of the result.
- If the mask bit for the exception is clear, the flag for the exception is not set and a floating-point arithmetic fault is raised, by saving diagnostic information in the fault information area and diverting instruction execution to a fault handler.

Since the floating reserved-encoding exception does not have a flag or mask bit, it always results in a fault.

NOTE

> The floating-point exception flags are cleared only with explicit instructions. They are not cleared automatically at the next successful floating-point operation.

Under certain circumstances, multiple floating-point exceptions can occur at the same instruction. The two cases are inexact and underflow, and inexact and overflow. If either of these cases occurs, the masked exceptions get their exception flags set, while the unmasked exceptions report the fault without setting the corresponding exception flags.

5.10.1 Fault Handler

When a floating-point fault is signaled, a single fault handler is invoked. This fault handler determines how to handle the specific fault subtype by interpreting the floating-point exception flags and the information in the fault record. This process is described in detail in Chapter 10.

5.10.2 Floating Reserved-Encoding Exception

A reserved encoding exception occurs as a result of either of the following two conditions:

- When a reserved encoding is used as an operand in a floating-point instruction, or
- When a denormalized value is used as an operand in a floating-point instruction and the normalizing-mode bit in the arithmetic controls is clear.

The first condition is rare, and occurs only when an extended-real value has a zero j-bit (integer part) and a non-zero biased exponent.

The second condition was discussed in Section 5.4.5. This condition is also rare, since the normalizing mode is typically enabled.

There is neither a flag nor a mask bit for this exception. When a reserved-encoding exception occurs, floating reserved-encoding fault is raised, and the result is discarded (not stored).

5.10.3 Floating Invalid-Operation Exception

The invalid-operation exception indicates that one of the source operands is inappropriate for the type of operation being performed. The following conditions cause this exception to be signaled:

- Any arithmetic operation on an SNaN
- Addition of infinities of unlike sign
- Subtraction of infinities of like sign
- Multiplication of zero by ∞
- Division of zero by zero or ∞ by ∞
- Remainder of x by y, if y is zero or x is ∞
- Square root of a negative, nonzero value
- Conversion of a NaN from floating-point format to integer format
- Sine, cosine, or tangent of ∞

- y * log (x), if:
  - x is negative and nonzero,
  - y is zero and x is ∞,
  - y and x are zero, or
  - y is ∞ and x is 1
- Log epsilon of (y, x), if y is ∞ and x is 0
- Compare ordered, if a source operand is a NaN

When a floating invalid-operation exception occurs and its mask is set, the following occurs:

- When the result is a floating-point value, the standard QNaN value is stored in the destination and the floating invalid-operation flag is set. (A discussion of how the NaNs are handled is provided in Section 5.9.)
- When the result is an integer, the maximum negative integer is stored in the destination and the floating invalid-operation flag is set.

When the mask is clear, no result is stored; the floating invalid-operation flag is not set; and the floating invalid-operation fault is signaled.

5.10.4 Floating Zero-Divide Exception

The floating zero-divide exception is signaled when an exact infinite result would be produced from finite operands. (Note that a different exception, overflow, is signaled when an infinite result is produced inexactly from finite operands.) The most common example of this exception is a division operation, where the divisor is zero and the dividend is a nonzero, finite value. The default result is an infinite with the sign equals to the exclusive-or of the sign of the two source operands. A zero-divide exception also occurs in log and log binary instructions.

When the floating zero-divide mask is set, a correctly signed ∞ is stored in the destination and the floating zero-divide flag is set. When the mask is clear, no result is stored, the floating zero-divide flag is not set, and a floating zero-divide fault is signaled.

5.10.5 Floating Overflow Exception

The overflow exception occurs when the infinitely precise result of a floating-point instruction exceeds the largest allowable finite value for the specified destination format. Overflow occurs when the infinitely precise result falls outside the range $-1.0 * 2^{Emax+1}$ to $1.0 * 2^{Emax+1}$ (exclusive), where Emax is 127 for real, 1023 for long real and 16383 for extended real.

When the floating overflow mask is set, a rounded result is stored in the destination and the floating overflow flag is set. The current rounding mode determines the method used to round the result. The default result is described in Section 5.4.6.

When the mask is clear, no result is stored in the destination and the floating overflow flag is not set. Instead, the the result is saved in extended-real format in the fault information area. The fraction of the extended-real value is rounded to the instruction's destination precision. For example, if the destination operand's format is real (32 bits), the extended-real fraction is rounded to 23 bits, with the 40 least-significant bits filled with zeros.

If the exponent exceeds the range of the extended-real format (16383 unbiased), then the exponent is divided by 2^{24576} and a flag is set in the fault information area to indicate that the exponent has been bias adjusted. After this fault information is stored, a floating overflow fault is signaled.

When using the scale instructions (**scaler** or **scalerl**), massive overflow can occur, where the infinitely precise result is too large to be represented, even with a bias-adjusted exponent. Here, a properly signed ∞ is stored in the fault record.

The floating overflow exception cannot occur on a conversion from floating-point format to integer format (although an integer overflow exception can occur).

5.10.6 Floating Underflow Exception

An underflow condition occurs when the infinitely precise result of a floating-point instruction is less than the smallest possible normalized, finite value for the specified destination format. Underflow occurs when an infinitely precise result falls in the range $-1.0 * 2^{Emin}$ to $+1.0 * 2*^{Emin}$, where Emin is -126 for real, -1022 for long real, and -16383 for extended real.

When a floating underflow condition occurs, the setting of the floating underflow mask determines how the condition is handled.

If the mask is set when an underflow condition occurs, the the result is denormalized. Then if the result is exact, it is stored in the destination and the floating underflow exception is not signaled, nor is the floating underflow flag set. If, on the other hand, the denormalized result is inexact, the floating underflow flag is set and the the inexact condition is handled (as described in Section 5.10.7).

If the floating underflow mask is clear when an underflow-condition occurs, no result is stored in the destination and the floating underflow flag is not set. Instead, the result is stored in extended-real format in the fault information area, with the fraction of the extended-real value rounded to the instruction's destination precision. For example, if the destination precision is real (23-bit fraction) the 40 least-significant bits of the fraction are set to 0.

If the exponent of the value stored is less than the minimum allowable value in the extended-real format (-16,382 unbiased), then the exponent is multiplied by 2^{24576} and a flag is set in the fault information area to indicate that the exponent has been bias adjusted. After this information is stored, a floating underflow fault is signaled.

The scale instructions can cause massive underflow to occur, where the infinitely precise result is too small to be represented, even with a bias-adjusted exponent. Here, a properly signed zero is stored in the fault record.

Refer to Section 5.10.8 for more information on the interaction of the floating underflow and inexact exceptions.

5.10.7 Floating Inexact Exception

The floating inexact exception occurs when an infinitely precise result cannot be encoded in the format specified for the destination operand. Either of the following two conditions can cause an inexact exception to be signaled:

- When a result is rounded and the result is not exact

- When overflow occurs and the floating overflow mask is set

If the floating inexact mask is set when an inexact condition occurs and an unmasked overflow or underflow condition does not occur, the rounded result is stored in the destination and the floating-point inexact flag is set. The current rounding mode determines the method used to round the result.

If the floating inexact mask is clear when an inexact condition occurs, the floating inexact flag is not set and one of the following operations is carried out:

- If only the inexact condition has occurred, the rounded result is stored in the specified destination and a floating-inexact fault is raised.
- If the inexact condition occurs along with overflow or underflow, no result is stored in the destination. Instead, the result is stored in extended-real format in the fault information area, as described for the floating overflow and underflow exceptions, then a floating inexact fault is raised.

Refer to the following section for more information on the interaction of the floating underflow and inexact exceptions.

5.10.8 Floating-Point Underflow Condition

Two aspects of underflow are important in numeric processings: the "tininess" of a number and "loss of accuracy." A result is tiny when it is nonzero and its exponent is between ± 2^{Emin}, where E_{min} is the smallest unbiased exponent allowed in the destination format. For example, if the destination format is long-real (64-bit format), a result is tiny if it is nonzero and in the range of $+1 * 2^{-1022}$ to $-1 * 2^{-1022}$. The ability to detect a tiny result is important because such a result may cause an exception to be signaled in a later operation (for example, overflow on a division).

Loss of accuracy occurs when a tiny result is approximated as part of the denormalization process so that it will fit into the destination format.

Tininess is detected after rounding as an underflow condition. Loss of accuracy is detected as an inexact condition.

The algorithm in Figure 5-6 shows how the these two conditions are handled when a floating-point operation produces a tiny result.

An important point to note in this algorithm is that if the underflow mask is set, an underflow exception is signaled only if the denormalized result is inexact. If the denormalized number is exact, no flags are set and no faults are signaled.

Underflow is different from all other exceptions in that the condition under which the exception occurs depends on whether the underflow mask is set. The only difference is in the case where a denormalized result is exact. When underflow is masked, neither underflow nor inexact flags are set. When underflow is not masked, an underflow fault is generated.

```
generate infinitely precise result; # exponent and significand;
if exponent < underflow threshold then
   if underflow fault mask clear then
      goto underflow fault handler;
      exit algorithm;
   else  # underflow fault is masked
      generate denormalized number;
      if denormalized significand equals infinitely precise significand then
         store denormalized result in destination;
         # no underflow is signaled;
      else
         set underflow flag in AC;
         if inexact fault mask is clear then
            goto inexact fault handler;
            exit algorithm;
         else  # inexact fault is masked
            set inexact flag in AC;
            store denormalized result in destination;
         end if;
      end if;
   end if;
else  # no underflow, but result may be inexact
   if infinitely precise result is inexact then
      if inexact fault mask is clear then
         goto inexact fault handler;
         exit algorithm;
      else  # inexact fault is masked
         set inexact flag in AC;
         store normalized result in destination;
      end if;
   else  # result is exact
      store normalized result in destination;
   end if;
end if;
exit algorithm
```

Figure 5-6. Interaction of Floating Underflow and Inexact Exceptions

PROTECTION MODEL 7

This chapter describes the subsystem call and protection mechanism. This chapter discusses the domain object, the environment table, the subsystem mechanism, and the supervisor call mechanism.

7.1 Introduction

The processor supports two protection models: the conventional "supervisor" model, and the "subsystem" model. (BiiN™ use the latter.) The supervisor model is principally described in this manual for completeness, and also because the BiiN™ Operating System internally uses the supervisor mode for operations such as interrupt handling.

In the supervisor protection model, a process consists of two address spaces, one for the user (application program), and one for the supervisor (the operating system). In supervisor mode, the operating system has complete access to both address spaces.

In the subsystem protection model, a process consists of any number of distinct address spaces. Unlike the supervisor model, no address space has implicit access to any other address space. Procedures may be invoked across address space boundaries in order for a process to gain access to those data structures and procedures that are otherwise inaccessible to an address space. A "domain" object represents the "public" interface of an address space. A domain object defines the address space and the "procedure table" of entry points into that address space.

The term "subsystem" transfer (call/return) describes the subsystem protection mechanism. The term "supervisor" transfer describes the supervisor protection mechanism.

7.2 Supervisor Protection Model

The two modes of execution, "User" mode and "Supervisor" mode, support the efficient emulation of conventional operating systems. (See Chapters 15 and 16 for additional details.) Note however that there are no "privileged instructions"; that is, all instructions can be executed in either mode. A program gains privilege by nature of its access rights and its execution mode. The page rep rights in the current linear address space are interpreted differently depending on the execution mode (see Chapter 9). Operating system storage generally has page rep rights which do not allow user access, but may be read-only or read/write in the Supervisor mode.

In systems where tagging is disabled (see Chapter 16), the tag bit is not available to distinguish between data and ADs. Since all operands have tag bits of zero, any attempt to execute instructions or operand specifiers which require an AD will fault in User mode. In the supervisor mode, the fault is disabled and the data is treated as an AD. Supervisor mode allows the execution of instructions which use ADs as operands (for example, the SEND instruction requires an AD to a port object).

In systems where tagging is enabled (as in the BiiN™ Operating System), the only difference between the user and supervisor modes is the page rep rights interpretation. The automatic interpretation of data as ADs in supervisor mode is not supported. Instructions that require ADs can be executed only if the specified operand is an AD.

In an untagged system, the **calls** instruction is the only way to change the execution mode to supervisor without faulting. The system domain contains a set of entry procedures for the operating system.

The supervisor procedure call (via the **calls** instruction) is similar to a local call. When a supervisor procedure is specified in the procedure table in a domain object (see Chapter 7), the domain object specifies the new supervisor stack pointer. If the process is in user mode, the supervisor stack pointer becomes the new frame pointer. If the process is already in supervisor mode, the stack pointer in the current frame is aligned to the next 64-byte boundary to form the new frame pointer. This allows calling supervisor procedures from a supervisor procedure. The supervisor stack is required to be frozen (that is, locked in memory). This is an intra-address-space transfer with the exception that the execution mode (and trace enable) of the process can be changed as part of the call. Typically, the data, instructions, and stack of the supervisor is located in the processor-specific object of the current address space.

The return status field signals a supervisor return on a return from the frame. A supervisor return is performed only if the process is in supervisor mode at the beginning of the instruction. Otherwise, a local return is performed. This prevents the modification of the trace control and the selection of either fault or interrupt return by a procedure in user mode.

The supervisor procedure mechanism is intended for a simple untagged operating system. This mechanism should be sufficient for an operating system that requires only two stacks (user stack and supervisor stack) sharing the same linear address space and two protection levels.

7.3 Subsystem Based Protection

The target address space in a subsystem call is defined by the contents of a "domain" object. The "procedure table" contains a list of valid entry points into the domain; that is, a procedure table entry is a pointer for the first instruction of a procedure.

Two or more domain objects can specify the same address space, but with different procedure table entries. With different entries, each domain object provides a different kind or level of service. By limiting which programs or users get which domains, different service levels can be granted to different users or programs.

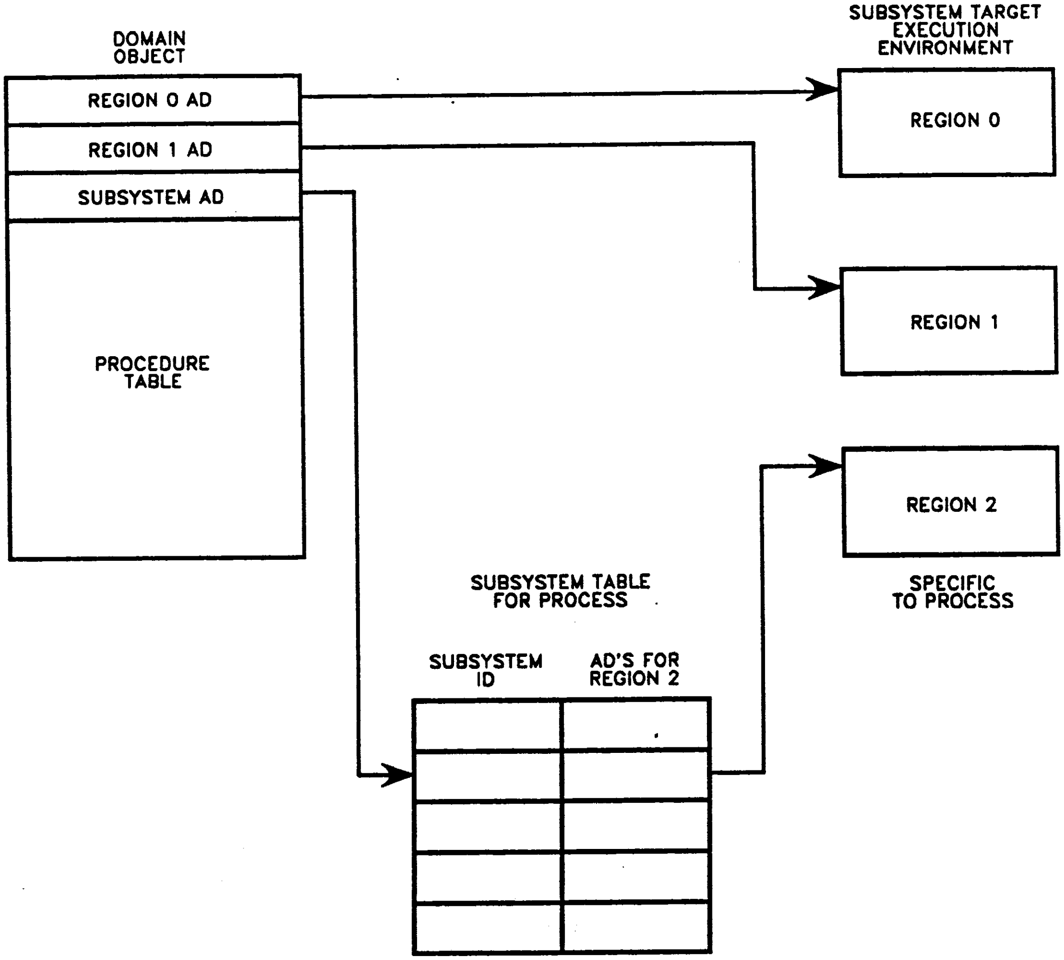


Figure 7-1. Subsystem Transfer

Private information or objects associated with an address space are not directly accessible via a domain from other address spaces. As part of a subsystem call operation, objects that define the address space are made accessible inside the address space.

A calld instruction requires an AD for the domain object with read rep rights. Except for the domain object type manager, a domain object AD should not have write rights. Otherwise, protection can be bypassed by modifying a domain object.

7.3.1 Target Address Spaces

A domain object specifies the target address space of an subsystem call. A subsystem transfer may change one or more of the objects that define the current address space. With appropriate placement of static data, stack frames, and instructions, a subsystem transfer may not need to change all three objects at the same time. A typical (but not necessary) partitioning (such as the one used in the BiiN™ Operating System) might be:

- **"Data" Object.** Region 0 contains static data and private variables. A subsystem call/return changes at least region 0.

- **"Instruction" Object.** Region 1 contains instructions. This allows sharing the instruction part of a domain by copying a single AD without having to use page table sharing. This object may remain unchanged during a subsystem call/return.
- **"Stack" Object.** Region 2 contains stack frames. This object is process-specific and sharing among processes is not possible. This object may remain unchanged during a subsystem call/return if the subsystem caller and callee trust each other.

If the fault handler for virtual-memory faults is specified in the fault table (Chapter 10) as a subsystem, the OTEs for the objects of this subsystem must be marked as valid. Failing to do so will lead to a system error or to an incorrect frame or stack pointer when the fault handler is invoked.

7.3.2 Subsystem ID and Subsystem Table

A domain does not directly specify region 2 AD of the target address space, but indirectly with a subsystem ID. The "subsystem table" locates all the stack (region 2) objects and their corresponding topmost frame and stack pointers associated with a process. A subsystem ID can be mapped to different region 2 ADs, each associated with a different process. The subsystem ID selects a subsystem entry in the subsystem table in the environment table object associated with the current process. A subsystem entry, in turn, specifies the region 2 AD and the topmost stack frame in the object.

A subsystem ID can be can be either an AD or data. If the subsystem ID is an AD, the object index field of the AD provides a system-wide unique ID for the subsystem. Otherwise, software needs to assign unique IDs to each subsystem within a single process. The subsystem ID together with the subsystem table serves the following functions:

- **Stack object sharing among domains.** Domains for the same subsystem use the same subsystem ID to allow stack object sharing within the same process. This also allows subsystems that trust one another to share the same stack.
- **Reentrancy of a stack object.** When a subsystem is exited on a call to another subsystem, the linear address of the topmost frame is saved in the subsystem entry. This allows a call from subsystem A to subsystem B which in turn calls subsystem A without returning from subsystem B first.
- **Trusted subsystems.** A subsystem ID of 0 indicates the current stack object is unchanged. This allows two mutually suspicious subsystems to share the same trusted library module (for example, a run-time library).
- **Guarantee stack resource.** Since the stack resource associated with a subsystem need not be shared with other subsystems, it is possible under some situations for software to guarantee that stack resource exhaustion never occurs. This allows certain faults be handled synchronously.
- **Domain object sharing among processes.** The subsystem ID also allows the same domain for different processes, but the same domain is mapped to different region 2 ADs using different process-specific stacks.

7.3.3 Control Stack

The subsystem call/return mechanism maintains a "control stack" (in the environment table object associated with a process) for the subsystem linkage information. A control stack is an array of control stack entries. Each control stack entry contains the saved state of an address space (to be restored on a return).

7.3.4 Extended Subsystem Environment

The subsystem model requires the following objects:

- **Environment Table Object.** This includes both the subsytem table and the control stack.
- **Current Subsystem ID.** The process is associated with a subsystem ID which is saved in the process together with related information (like the current subsystem table offset for the current subsystem entry).

These fields are ignored when not operating with the subsystem model.

7.3.5 Interrupt Environment

When the process is in the interrupted state, subsystem calls consult an interrupt environment table object. All subsystem calls in the interrupted state are handled as intra-subsystem calls (where the region 2 and stack are not changed). Thus, the interrupt environment table does not need a subsystem table at the beginning. The control stack in the interrupt environment must start at a fixed offset.

7.4 Domain Objects

A domain object has object type 0011_2.

The type rights in a domain object AD are uninterpreted.

An AD to a domain object should have read rep rights.

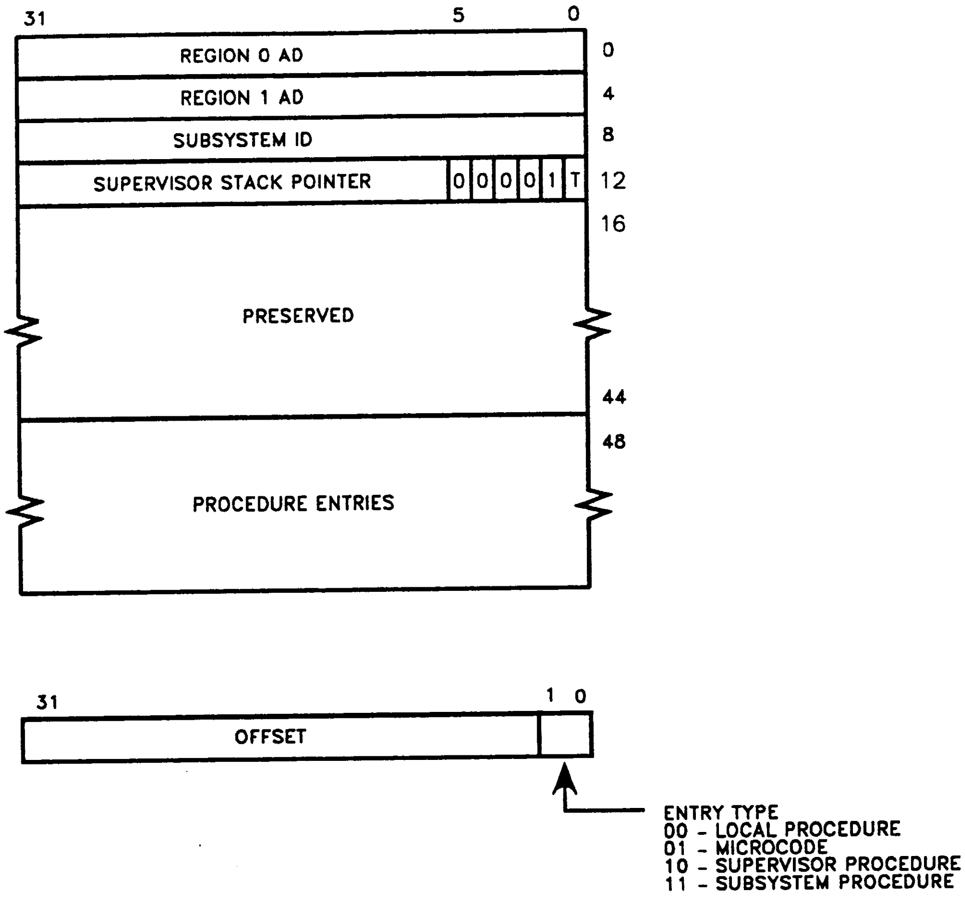


Figure 7-2. Domain Object

The structure of a domain object is given in Figure 7-2. The fields of a domain object are defined as follows:

- **Region 0 AD** (bytes 0-3). This AD references the object which defines region 0 of the target address space for a subsystem call. If the tag bit is zero, an *invalid AD* fault is raised.
- **Region 1 AD** (bytes 4-7). This AD references the object which defines region 1 of the target address space for a subsystem call. If the tag bit is zero, an *invalid AD* fault is raised.
- **Subsystem ID** (bytes 8-11). This mixed value is the subsystem ID that selects an entry in the subsystem table in the environment table associated within the process object. A subsystem entry in the subsystem table specifies the object that defines region 2 of the target address space and the frame pointer of the topmost stack frame in the address space. If this field is a data word of zero, the current region 2 remains unchanged and the current frame is the topmost stack frame. Bits 6-31 of the subsystem ID produce a hash value into the subsystem table.
- **Trace Control, T** (byte 12, bit 0). This bit specifies the trace enable bit of the process controls after a subsystem or supervisor call via this object. This bit disables or enables tracing inside the new address space. This bit is ignored during an explicit call to a supervisor procedure in supervisor mode. This bit has the same encoding as that in the process controls (see Chapter 15).

- **Supervisor Stack Pointer** (bytes 12-15, bits 2-31). This is a linear address (word-aligned) for the supervisor stack. Supervisor calls locate the new frame with this field (in the user mode) instead of the stack pointer in the current frame.

  The process distinguishes between a user stack (in user mode) and a supervisor stack (in supervisor mode). If the supervisor stack pointers in different domains contain different values, all the stacks must be big enough to handle the needs of all the supervisor procedures. Hence, the process supervisor stack pointers should be the same. Since the fault table is associated with a processor, all the processes sharing the same processor need to have a supervisor stack as specified by the supervisor fault handling procedures. Hence, the supervisor stack pointer should be a system wide constant.

- **Procedure Entries** (from byte 48 to the end of the object). A procedure entry specifies the type and address of the target procedure. The fields of a procedure entry are defined as follows:

  - **Procedure Entry Type** (bits 0-1). This field indicates the type of procedure to be invoked. The encodings of this field are as follow:

00	local procedure
01	reserved
10	supervisor procedure
11	subsystem procedure

  - **Offset** (bits 2-31). This 30-bit field is a word offset into the target address space to the first instruction of the target procedure.

  The domain object may be bigger than the last entry of the procedure table. If so, the unused entries should point at a local procedure that reports an error, to prevent compromise of the protection mechanism.

7.5 Environment Table Object

An "environment table object" contains two data structures: a subsystem table and a control stack. This object contains information necessary for all subsystem transfers within a single process; thus, there is an one-to-one correspondence between a process object and an environment table object.

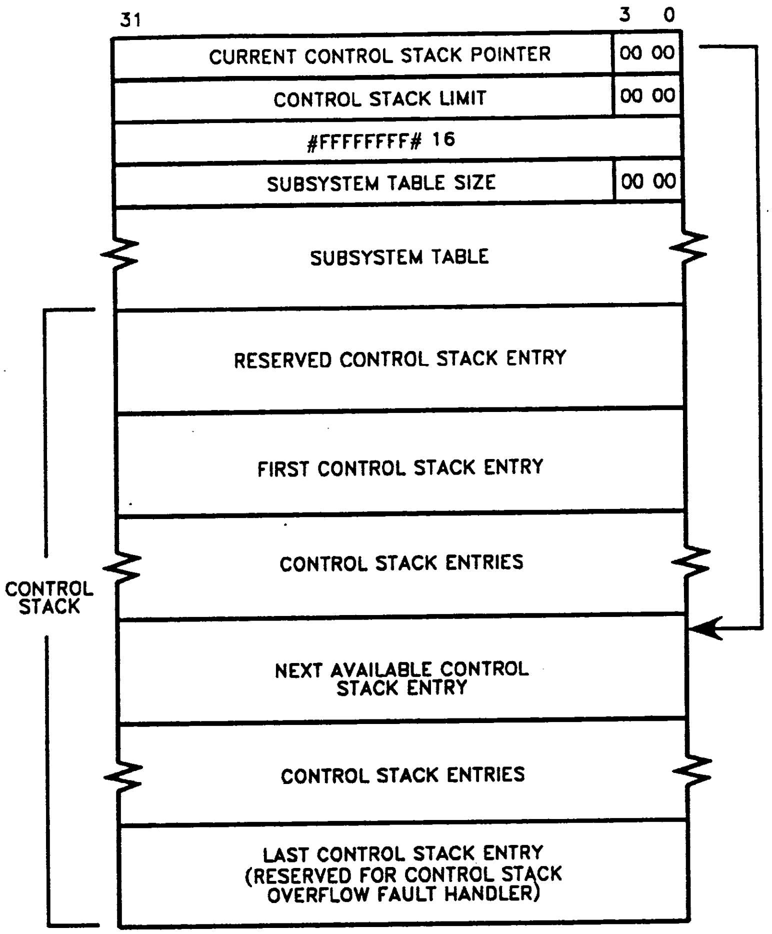

31
3
0
CURRENT CONTROL STACK POINTER
00 00
CONTROL STACK LIMIT
00 00
#FFFFFFFF# 16
SUBSYSTEM TABLE SIZE
00 00
SUBSYSTEM TABLE
RESERVED CONTROL STACK ENTRY
FIRST CONTROL STACK ENTRY
CONTROL STACK ENTRIES
CONTROL STACK
NEXT AVAILABLE CONTROL STACK ENTRY
CONTROL STACK ENTRIES
LAST CONTROL STACK ENTRY (RESERVED FOR CONTROL STACK OVERFLOW FAULT HANDLER)

Figure 7-3. Environment Table Object

An environment table object does not have a predefined object type. Some parts of the environment table may be held inside the processor. The fields of an environment table object are defined as follows:

- **Subsystem Table.** This area is described in the following section. The first entry stores the current control stack pointer, control stack limit, and subsystem table size.
- **Control Stack.** This area is described in the following section.

7.5.1 Subsystem Table

During a subsystem call, a domain object directly specifies only two of the three objects that define the new address space. The domain object contains a subsystem ID which indirectly specifies the third object of the new address space. A subsystem table is a data structure within an environment table object which provides the mapping of a subsystem ID to region 2 AD of the new address space.

The first entry of the subsystem table is a dummy entry with the following defined fields:

- **Current Control Stack Pointer, CCSP** (bytes 0-3, bits 4-31). This is a quad-word index for this object to the next available CSE. This field is incremented on a subsystem call and decremented on a subsystem return.

- **Control Stack Limit,** CSL (bytes 4-7, bits 4-31). This is a quad-word index for this object to the first CSE reserved for the control stack overflow fault handler (that is, not for regular uses). When CCSP = CSL after the completion of a subsystem call, a *Control-Stack Overflow* fault is generated.
- **Subsystem Table Size** (bytes 12-15, bits 4-29). This field contains one less than the size (in units of subsystem entries) of the subsystem table. The size of a subsystem table must be a power of 2; thus, this field contains a bit mask of ones in the least significant bits. Otherwise, the behavior is unpredictable. Thus, this field may be interpreted as the index of the last subsystem table entry.

7.5.2 Subsystem ID to Subsystem Entry Mapping

A subsystem ID selects the corresponding subsystem entry as follows:

- If the specified subsystem ID is zero or equal to the current subsystem ID, the current subsystem ID is selected. Otherwise, search the subsystem table as specified below.
- Bits 6-31 of the specified subsystem ID are logically-ANDed with the subsystem table size to form the initial subsystem entry index.
- Repeat the following,
  - If the subsystem ID in the selected subsystem entry is zero, a *Subsystem Not Found* fault is raised.
  - If the subsystem ID in the selected subsystem entry matches (as if a **cmpm** instruction were executed) that of the specified subsystem ID, exit from the search.
  - Otherwise, select the previous subsystem entry by searching backward linearly through the table. Entry 0 wraps around to the last entry pointed to by the system table size.
  - If this is the initial subsystem entry and the process is not in interrupted state, raise a *Subsystem Not Found* fault. If the process is in interrupted state, the current subsystem is selected.

7.5.3 Subsystem Entries

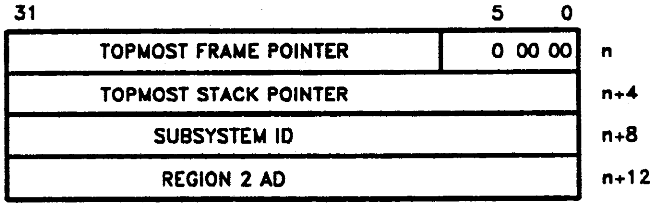


Figure 7-4. Subsystem Entry

The structure of a subsystem entry is shown in Figure 7-4. The fields of a subsystem entry are defined as follows:

- **Topmost Frame Pointer** (bytes 0-3, bits 6-31). This field contains the frame pointer of the topmost stack frame. During a subsystem call into this address space, this field defines the previous frame pointer in the new frame. During a subsystem call from this address space, the current frame pointer is saved here. During a subsystem return to this address space, this defines the target frame pointer. During a subsystem return from this address space, the previous frame pointer in the current frame is saved here.

- **Topmost Stack Pointer** (bytes 4-7). This field contains the stack pointer in the topmost stack frame. During a subsystem call into this address space, this field defines the frame pointer of the new frame. During a subsystem call from this address space, the current stack pointer is saved here. During an inter-subsystem return from this address space, the current frame pointer (that is, the rounded stack pointer in the previous frame) is saved here. During other inter-subsystem returns from this address space, the current frame pointer minus 64 (that is, the rounded stack pointer in the previous frame) is saved here. During a return to this address space, this field is ignored.
- **Subsystem ID** (bytes 8-11). This ID identifies the subsystem the target address space is associated with. This ID is the key for a matching subsystem ID to a region 2 object; thus it is unique within the subsystem table. A value of zero indicates this subsystem entry is unallocated. Subsystem table entry 0 (with a subsystem ID of all ones) stores control stack information.
- **Region 2 AD** (bytes 12-15). This AD references the object that defines the region 2 of the target address space partially specified by this entry. This AD must contain read/write rights, otherwise, a *Rep Rights* fault is raised.

7.5.4 Control Stack

The organization of the control stack is shown in Figure 7-3. A control stack entry is pushed on the control stack on a subsystem call, and popped off the control stack on a subsystem return. The control stack is delimited on the low end by a reserved control stack entry. The control stack is delimited on the high end by the control stack limit plus a few reserved entries (for the control stack overflow fault handler). The number of entries to be reserved for stack overflow fault handler is software-defined.

7.5.5 Control Stack Entries

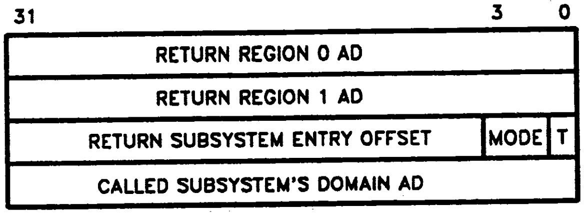


Figure 7-5. Control Stack Entry

The format of a control stack entry is shown in Figure 7-5. The fields of a control stack entry are defined as follows:

- **Return Region 0 AD** (bytes 0-3). This AD references the object that defines region 0 of the calling address space of the corresponding subsystem call. On a subsystem return, region 0 is restored to this object. This AD must contain read/write rights; if not, a *Rep Rights* fault will be raised.
- **Return Region 1 AD** (bytes 4-7). This AD references the object that defines the region 1 of the calling address space of the corresponding subsystem call. On a subsystem return, region 1 is restored to this object. This AD must contain read/write rights; if not, a *Rep Rights* fault will be raised.
- **Trace Control, T** (byte 8, bit 0). This bit contains the trace enable bit in the process controls during the corresponding normal subsystem calls. During a subsystem return, the trace control is restored to this bit.

- **Return Mode,** MMM (byte 8, bits 1-3). This 3-bit field indicates the type of entries. This field is encoded as follows:

000	Normal intra-subsystem
001	Normal inter-subsystem
100	Fault intra-subsystem
101	Fault inter-subsystem
x1x	reserved; *Control-Stack Underflow* fault is raised

  Fault procedures are described in Chapter 10.

- **Return Subsystem Entry Offset** (bytes 8-11, bits 4-31). This field contains the entry index into the subsystem table (in the environment table) for the subsystem entry that defines the region 2 of the calling address space.

  When a subsystem table is expanded and rehashed, the subsystem entry offset changes and needs to be updated.

- **Callee's Domain AD** (bytes 12-15). This AD references this subsystem call's domain object. This is initialized during a call, but it is ignored on a return.

7.6 Domain CALL/RETURN

calld
calls
ret
ldcsp

The **calld** instruction invokes the procedure specified by the procedure number in the specified domain object, and changes the address space as specified by the domain object. The specified domain AD must have read rights. The procedure number is a word index into the procedure table in the specified domain object for a procedure entry.

The **calls** instruction calls a procedure in the system domain. The system domain is a domain referenced by the processor object. This instruction is necessary to allow supervisor calls in an untagged system. In a tagged system, the system domain can map supervisor calls of an untagged operating system to a tagged operating system.

The **ret** instruction returns to the caller. The particular return action is determined by the return status field in the previous frame field of the current frame. This allows procedures to be invoked from both inside or outside of their associated domains, even though different actions are taken.

The **ldcsp** instruction returns the current control stack pointer of the process.

OBJECT ADDRESSING 8

This chapter describes objects, and how they are addressed.

8.1 Address Spaces

The three address spaces are linear, virtual, and physical. A linear address space is mapped onto a virtual address space which is, in turn, mapped onto a physical address space. An address in each space has a unique structure.

Physical addresses define specific memory and I/O locations outside the processor. Virtual addresses define objects and locations within objects; the operating system manages virtual addresses to support multiprocessing and multiprogramming. Linear addresses provide a flat address space to be used by programs operating in a "conventional" manner.

8.1.1 Physical Address Space

The physical address space consists of 2^{32} (4G) bytes. The physical address space covers read-write memory, read-only memory, and memory-mapped I/O. The last 16M bytes of the physical address space (from $FF000000_{16}$ to $FFFFFFFF_{16}$) are reserved; see Appendix B for details.

The physical address space is byte addressable and must guarantee atomic and indivisible access (read or write) for memory addresses that fall within 16-byte boundaries. An **indivisible** access guarantees a processor reading or writing a set of memory locations will complete the operation before another processor can read or write the same location. An **atomic** operation allows a processor to read and modify a set of memory locations (within a aligned 16-byte block) with the guarantee that another processor doing an atomic operation on the same block will be delayed.

The physical medium representing an area of the physical address space may have more restrictions on the alignment and the length of an individual access. These restrictions are not necessarily detected.

Multiple processors may share a single, common physical address space.

8.1.2 Virtual Address Space

The system-wide virtual address space is actually a collection of **objects**. Given that any datum within an object is located by a simple offset, the **virtual address** of the datum is specified by two components: an **object index** that selects the desired object and an **object offset** of the datum within the object. The size of the virtual address space is the product of the number of objects allowed and the maximum size of each object. A maximum of 2^{26} (64M) objects are allowed, with a maximum of 2^{32} (4G) bytes per object, yielding a total virtual address space of up to 2^{58} bytes.

An object is also defined as the unit of protection. To control access within the virtual address space, the generation of object indices is protected. A special type of pointer, called an **access descriptor** (AD), contains an object index. (Ordinary data can not be used where an access descriptor is required.) An access descriptor can point to any of the 64 million objects in the virtual address space. It is more accurate, therefore, to say that a **virtual address** is specified by a protected object index (an access descriptor) and an unprotected offset.

The use of a system-wide virtual address space allows different processors and processes to communicate with each other without the conventions and restrictions imposed by conventional architectures on shared address spaces.

8.1.3 Instantaneous Address Space

Access descriptors, directly or indirectly accessible, are conceptually assembled in sets to form yet a third type of address space called the instantaneous address space. The instantaneous address space defines the visibility of the execution environment. The execution environment is further described in Chapter 6. Addresses are mapped onto the single virtual address space. For maximum flexibility, there are two types of instantaneous addresses.

The first type, called a **linear address**, is defined by the four objects that form the execution environment. A linear address is used to represent the conventional notion of a process address space. Linear addresses, interpreted within a given environment, are mapped onto the virtual address space. The mapping of linear addresses to virtual addresses is a fundamental part of the instruction interpretation process. In a linear address, an operand specifier supplies only an offset; the current linear address space is implied. The upper two bits of a linear address implicitly selects one of the four objects that define the execution environment, while the remaining 30 bits are an offset into the selected object.

The second type, called the **structured address**, is defined by a virtual address (that is, AD plus offset). The structured address is used to access the more advanced object-oriented protection features. In a structured address, an operand specifier supplies a virtual address. Since an AD cannot be directly specified in the instruction stream, the AD part of the virtual address must be specified indirectly using an AD selector in an operand specifier. An AD selector specifies a register (global or local) that holds an AD.

8.2 Objects and Object Addressing

8.2.1 Access Descriptors and their Rights

An access descriptor is a protected pointer into the object space. Access descriptors are protected from accidental or malicious creation and modification.

A program cannot address an object directly, but only indirectly via an access descriptor. Since a program cannot reference an object without an access descriptor to it, nor can a program create an arbitrary access descriptor, a program's visibility is restricted to those objects it needs to access.

An access descriptor contains the following information:

- **Object Index.** This selects the object.
- **Rights.** An AD contains read rights, write rights and type rights. These rights indicate the permissible operations on the object. Rights are described in Chapter 9. Note that rights

are associated with an access descriptor and not with the object itself. It is thus possible to have different rights to the same object by selecting different access descriptors.

- **Lifetime.** This bit indicates the lifetime of the object this AD references. Lifetime is described in Section 8.4.

8.2.2 AD to Object Mapping

Objects are referenced using system-wide protected pointers called access descriptors. Each access descriptor contains a 26-bit object index; thus, there are 2^{26} (64M) possible objects. An object index selects an object table entry in the system-wide object table object. An object table entry specifies the location, size, type, and so on of the referenced object. Figure 8-1 introduces the mapping process.

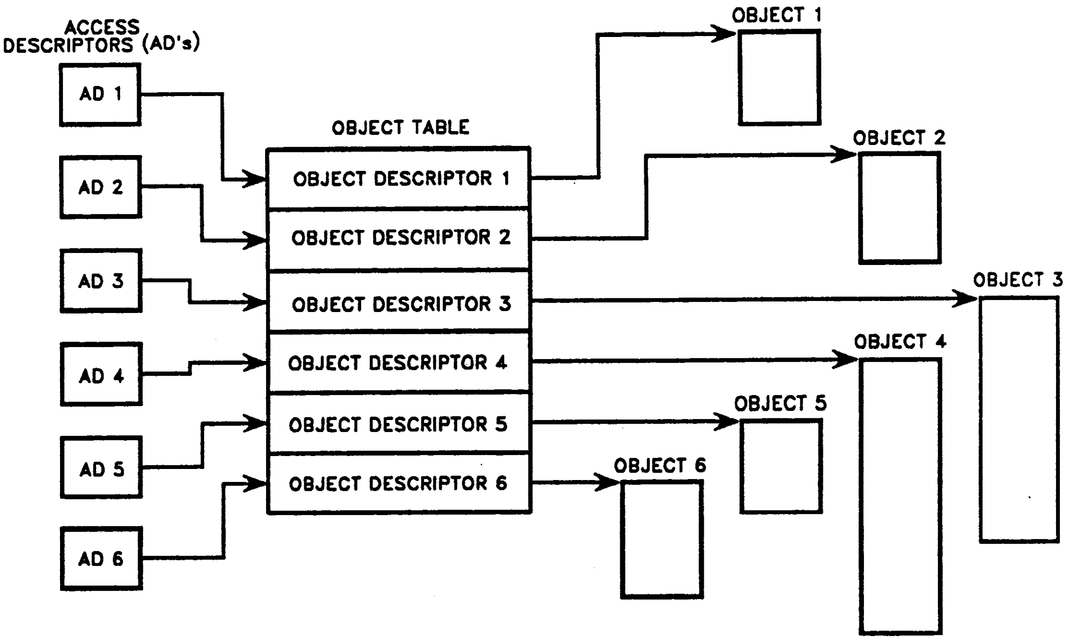


Figure 8-1. AD to Object Table to Object Mapping

8.2.3 Storage Blocks and Pages

Physical memory is partitioned into pages and suballocated pages. Suballocated pages are further subdivided into storage blocks.

An object is physically composed of a storage block and/or a set of pages. A block is a contiguous area in the physical address space. A block can be used to represent a simple object, a page table, or a page table directory.

The base address of a storage block points to the first byte of the block. The base address of a storage block must be aligned on a 64-byte physical address boundary. The length of a block varies from 64 bytes to 4096 bytes. A block cannot span across a 4K-byte boundary.

An object can also be represented by a set of pages with one or two levels of page tables. The first level table can be a storage block instead of a page. The pages that define an object are described by a page table. A page is a fixed size block of 4K bytes with base address aligned on a 4K-byte boundary.

8.2.4 Tagging

An object contains access descriptors and/or data, that is, any binary information. Access descriptors and data can reside in the same object and can be interleaved in any arbitrary order.

In some systems (such as BiiN™ Systems), a tag bit is associated with each 4-byte aligned word in memory to indicate whether the word is data or an access descriptor. An access descriptor must be aligned to 4-byte boundary with a tag bit of one. A tag bit of zero indicates data.

In other systems, the tag bit is not used. The interpretation of a word as data or an access descriptor depends on the operation; see Chapter 16.

In a word-aligned read or write of the whole word, the tag bit is either preserved or set to zero depending on the operation. In an non-word aligned read, or a partial read of a word, the tag bit of the returned value is always forced to zero. Similarly, in an non-word aligned write, or a partial write of a word, the tag bit of any of the modified words is always forced to zero. The data manipulation (arithmetic or logical) instructions require source operands with zero tag bits, and generate values with zero tag bits.

8.2.5 Typed Objects

Certain objects have a predefined internal organization. These objects play a key role in the protection system, the interprocess/interprocessor communication system, and the storage management system. To recognize these objects and to control their use, each one must be identified by a proper **object type**. This object type is maintained with the object's address mapping information. Additional object types may be assigned with their own type codes. Object typing is described in Chapter 9.

8.2.6 Object Offset

An object offset is a 32-bit ordinal used to specify a datum within an object. The offset is capable of pointing to either data or access descriptors in an object. An object offset is divided into a number of fields. The interpretation of these fields are dependent on the object representation (described in later sections).

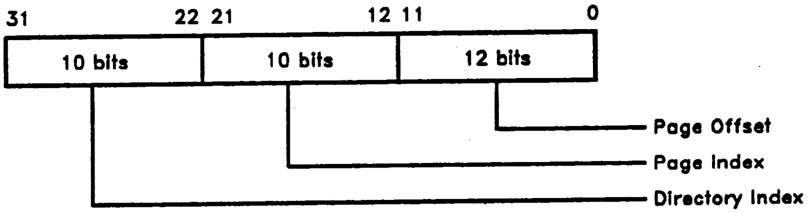


Figure 8-2. Object Offset

8.2.7 Object Size

The maximum size of an object is 2^{32} (4G) bytes. The size of an object is specified in an object table entry. The object offset in a virtual address plus the operand size is compared with the size of the referenced object on every address translation. This operation is called **bounds checking** and prevents reference beyond the specified object of a datum which may belong to another object. For a 2^{32} byte object, the offset wraps around instead of raising a fault. The granularity of an object varies from 64 bytes to 64M bytes, depending on the object representation.

8.3 Object Representation

An object is described by an object table entry. The object table entry provides provides the mapping information for the physical addresses of the storage blocks and pages. These blocks and pages represent the physical object. Three different mapping schemes are used for different maximum object sizes and to minimize object representation overheads.

- **Simple Objects.** A simple object is represented by a block in physical address space directly. The physical base address is stored directly in the object table entry. Such an entry is called a simple object descriptor. Simple objects are objects between 64 and 4K bytes.
- **Paged Objects.** A paged object is represented by a set of physical pages using a single-level page table. The object table entry for a paged object, called a paged object descriptor, contains the physical address of a page table, which is an array of page table entries for the pages. Paged objects are objects between 4K and 4M bytes.
- **Bipaged Objects.** A bipaged object is represented by a set of physical pages using two levels of page tables. The object table entry for a bipaged object, called a bipaged object descriptor, contains the physical address of a page table directory, which is an array of page table entries for page tables. Bipaged objects are objects between 4M and 4G bytes.

8.3.1 Simple Objects

A simple object is defined by a simple object descriptor and represented by a single block. The maximum size of a simple object is 2^{12} (4K) bytes. This size of a simple object is in units of 64 bytes. A simple object descriptor contains the physical base address and the block length. A simple object cannot span across a 4K-byte physical address boundary.

A simple object offset is partitioned as follows:

- **Directory Index DI** (bits 22-31). This 10-bit field must be zero. Otherwise an *Object Length* fault is raised.
- **Page Index PI** (bits 12-21). This 10-bit field must be zero. Otherwise an *Object Length* fault is raised.
- **Block Offset SO** (bits 0-11). This 12-bit field is the byte displacement added to the base address of the block to form the physical address for the first byte of the operand.

8.3.2 Paged Objects

A paged object is described by an object table entry called a paged-object descriptor. Paged objects are implemented with one level of page table. The maximum size of a paged object is 2^{22} (4M) bytes. The size of a paged object is in units of 4K bytes. The page table of a paged object is aligned on 64-byte physical address boundary and the size is in multiples of 64 bytes. A paged object is composed of 4K-byte pages and thus does not require contiguous physical address space of more than 4K bytes. Each page is individually swappable and relocatable, thus not all pages of a paged object need be present in physical address space at the same time. To access an item of a paged object, only the page table (64-4096 bytes) and the selected page (4K bytes) need to be located in the physical address space.

A paged-object descriptor contains the object length, but does not contain the base addresses of the pages which represent the object. The base address field of a paged-object descriptor contains the base address of the page table block. The length of the page table block is defined by the object length of the object.

A paged object offset is partitioned as follows:

- **Directory Index,** DI (bits 22-31). This 10-bit field must be zero. Otherwise, an *Object Length* fault is raised.
- **Page Index,** PI (bits 12-21). This 10-bit field is used to index into the selected page table for a page table entry.
- **Page Offset,** PO (bits 0-11). This 12-bit field is the byte displacement appended to the base address (in the page table entry) of the page to form the physical address for the first byte of the operand.

8.3.3 Bipaged Objects

A bipaged object is described by an object table entry called a bipaged object descriptor. Bipaged objects are implemented with two levels of page tables. The second level of page tables are always page aligned and 4K bytes in size. The maximum size of a bipaged object is 2^{32} (4G) bytes. The size of a bipaged object is in units of 4K bytes. The page table directory of a bipaged object is aligned on 64-byte physical address boundary and the size is in multiples of 64 bytes. A bipaged object uses 4K-byte page tables and 4K-byte pages and thus does not require contiguous physical address space of more than 4K bytes. Each page or page table is individually swappable and relocatable; thus, not all pages or page tables of a bipaged object need be present in physical address space at the same time. To access an item of a bipaged object, only the page table directory (64-4096 bytes), the selected page table (4K bytes), and the selected page (4K bytes) need to be located in the physical address space.

A bipaged object descriptor contains the object length, but does not contain the base addresses of the pages nor page tables which represent the object. The base address field of a bipaged object descriptor contains the base address of the page table directory block. The length of the page table directory block is defined by the object length of the object.

A bipaged object offset is partitioned as follows:

- **Directory Index,** DI (bits 22-31). The directory index selects a page table entry in the page table directory specified by the bipaged object descriptor.
- **Page Index,** PI (bits 12-21). The page index selects a page table entry in the specified page table.
- **Page Offset,** PO (bits 11-0). The page offset is used as an offset into the page. The page offset is appended to the base address (in a page table entry) to form the physical address for the first byte of the operand.

8.4 Object Lifetime

To support the implicit deallocation of certain objects while preventing dangling references, the object lifetime concept is supported. The lifetime of an object can be local or global. "Local" objects have a lifetime that is tied to a particular program execution environment (such as a "job" within the BiiN™ Operating System). "Global" objects are not associated with a particular execution environment.

Each job has a distinct set of local objects. No two jobs can have ADs that reference the same local object. The processor does not allow an AD for a local object to be stored in a global object. Thus, when a job terminates, all the local objects associated with a job cab be safely deallocated, and there cannot be any dangling pointers.

The local bits in access descriptors, object descriptors, and page table entries are the means by which the lifetime of an object is determined and prevents a potential dangling reference (AD) from being stored.

8.4.1 Local Bits

A local bit is associated with each object or page to denote its relative lifetime. The local bit is located in the object descriptor for a simple object, and the PTEs for a page. A value of 0 indicates a global object or page with unbound object lifetime. A value of 1 indicates a local object or page with bound object lifetime.

8.4.2 Lifetime Checking

The object lifetime check is performed every time an AD is stored. Since this requires the lifetime of the source object and destination location to be compared, the operation is called **lifetime checking.** If the source AD is valid and the local bit is 1 and the lifetime of the destination location (in an object table entry or a page table entry) is global, a *Lifetime* fault is raised.

In the implicit manipulation of system objects, lifetime checking is ignored unless explicitly specified.

8.5 Garbage Collection

The software implementation of a parallel garbage collector is supported, as described in "On-the-Fly Garbage Collection: An Exercise in Cooperation," by E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.M.F. Steffens, in Communications ACM 21(11) p. 966-5 (November 1978).

A gray and black boolean are associated with each object table entry to support parallel garbage collection. The gray boolean is represented as a byte in the gray table, while the black boolean is software defined. Since there are 2^{26} objects (maximum), the size of the gray table area can be up to 2^{26} (64M) bytes.

The gray boolean (byte) associated with an object is set to one whenever a new AD for the object is generated, while the mutator enable bit in either the processor object or current process object is set. See Sections 15.3 and 16.1.

The gray boolean is not necessarily set (marked) until the AD is stored in memory. Thus, in certain circumstances, the mutator operation can be delayed. The garbage collector software has to synchronize with these exceptions. Gray marking may be delayed until process suspension for:

- The content of local registers (Chapter 6)
- The content of global registers (Chapter 6)
- The current subsystem ID (Chapter 7)

- The current region 0-2 AD (Chapter 7)
- The current process AD (Chapter 15)

Gray marking is never perform for the:

- Object Table (Chapter 8)
- Process Object AD (Chapter 15)
- Region 3 AD (Chapter 6)
- System Domain AD (Chapter 7)
- Default TDO ADs (Chapter 9)
- Dispatching Port ADs (Chapter 14)

8.6 Mapping Tables

8.6.1 Object Table Objects

An object table object (OT) serves as the root of the virtual address mapping. An object table is an array of 16-byte object table entries. The object index field in an access descriptor selects an object table entry (OTE) in the object table. Object table entries are described in later sections.

Object tables do not have a predefined system type.

Although an AD to an OT has a global lifetime, the OTE/PTEs of an OT must have a local lifetime. This is necessary to support TDO ADs in OTEs.

One system-wide object table exists for all processors that share a single system-wide virtual address space.

The swappable part of the PTEs of an OT should be cacheable to allow caching of addressing translation information in an external cache.

8.6.1.1 Predefined Object Indices

The following object indices are predefined, and should not be used for any other purpose:

Object Indices	Purpose
0	Reserved for empty AD
(special for syn* instructions and the empty queue representation for port and semaphore operations)	
1-7	Preserved
8	Object Table
9-15	Preserved
16-31	Default TDO ADs for object types 0-15 (see Chapter 9)

8.6.2 Page Tables or Page Table Directories

Page tables provide one or two level(s) of mapping for paged objects and bipaged objects. Page table directories provide the first level of mapping for bipaged objects. Page tables (or page table directories) contain page table entries (or page table directory entries) which define the base address of a page, and other information for virtual memory management and garbage collection.

Page tables and page table directories are predefined, but are not objects and do not have a system type.

A page table is an array of page table entries, each of which is 4 bytes in length. Each page table entry in a page table describes a page in a paged object or a bipaged object. Each page table entry in a page table directory describes a page table for a bipaged object.

The page table of a paged object or the page table directory of a bipaged object can be variable in size and aligned on any 64-byte boundary. The page tables of a bipaged object must be 4K bytes in size and aligned on 4K-byte boundaries.

Page tables and page table directory are not objects and thus cannot be accessed directly in the virtual address space. One approach is to access them using physical addresses. Another approach is to map the page tables to part of the object they are defining. In the second approach, the physical address of the page table directory or the page table must be duplicated. If so, the software is responsible to guarantee the physical address alias is updated during swapping.

8.6.3 Gray Table Area

A gray table area supports parallel garbage collection. The system-wide gray table area is in region 3, and shared by all processors. The gray table area starts at offset $C0000000_{16}$ (in region 3) of any linear address space.

The gray byte for object index *i* is located in linear address $C0000000_{16} + i$.

The gray table must be frozen (defined in Section 8.9.1) while either the mutator enable bit in the processor object is 1 or the mutator enable bit in the current process object is 1.

The gray table can be swapped while the garbage collector is not active, or allocated before a garbage collection cycle and deallocated after a garbage collection cycle.

8.7 Descriptor Formats

8.7.1 Data Words

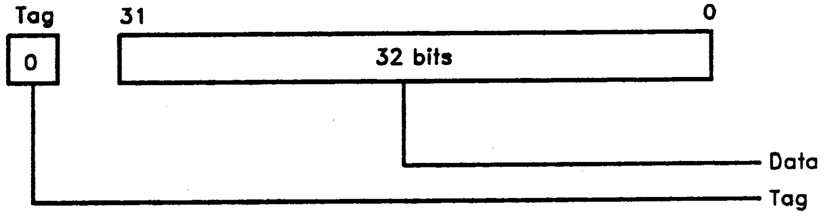


Figure 8-3. Data Word

The fields of a data value are defined as follows:

- Data (bits 0-31). This field contains any data value.
- Tag (Tag Bit). This bit is 0 for data values.

8.7.2 Access Descriptors

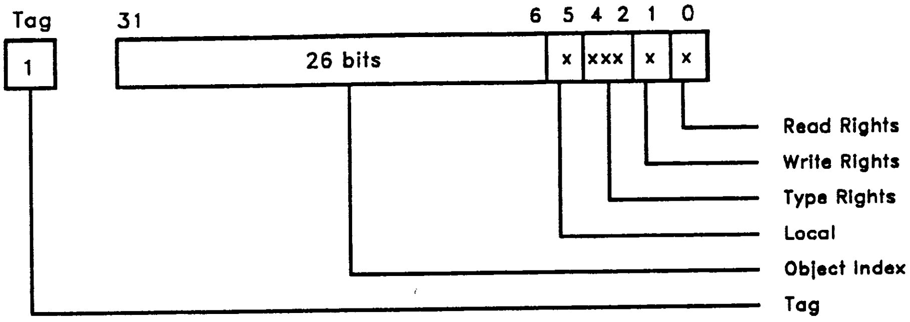


Figure 8-4. Access Descriptor Format

The fields of an access descriptor are defined as follows:

- **Read Rights** (bit 0). This bit indicates reading the contents of the object referenced by this access descriptor is allowed. Read rights are further described in Chapter 9.
- **Write Rights** (bit 1). This bit indicates whether writing the contents of the object referenced by this access descriptor is allowed. Write rights are further described in Chapter 9.
- **Type Rights** (bits 2-4). The interpretation of this 3-bit field is determined by the object type of the referenced object. Type rights are further described in Chapter 9.

Name of the bit	**Bit position in the AD**
Use	bit 2
Modify	bit 3
Control	bit 4

- **Local** (bit 5). This bit indicates the object's lifetime. This bit is 0 for a global object and 1 for a local object.
- **Object Index** (bits 6-31). This 26-bit field selects an object table entry in the object table.
- **Tag** (Tag Bit). This bit must be 1 for a valid access descriptor.

8.7.3 Mixed Words

A mixed word can be viewed as either a data word or an access descriptor depending on the context.

The values of a mixed word are divided into the following classes:

1. **[Valid] Access Descriptor.** A valid access descriptor has the tag bit set to 1. This can be dereferenced (used to reference the content of the object) if the object for the corresponding index is defined. The *Invalid AD* fault is raised when the tag bit is 0 and an AD is expected.
2. **Data.** A data word has the tag bit set to zero. When a data value is generated, the tag bit is always set to zero. When a data value is expected, the tag bit is ignored and interpreted as zero.

The instructions to manipulate access descriptors or mixed words are described in Section 8.8.

8.7.4 Virtual Addresses

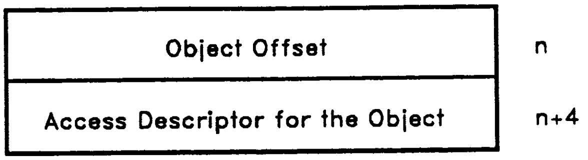


Figure 8-5. Virtual Address Format

The fields of a virtual address are defined as follows:

- **Object Offset** (bytes 0-3). This 32-bit field contains an ordinal offset into the object referenced by the access descriptor in the virtual address.
- **Access Descriptor** (bytes 4-7). This AD specifies the object referenced by this virtual address. The AD also specifies the permissible operations using this virtual address.

8.7.5 Object Table Entries

An object table can contain the following types of object table entries. All object table entries are 16 bytes in size.

Specific object table entries are identified by the entry type field (bits 96-98) of each object table entry as follows:

000	Invalid Object Table Entry
001	Embedded Descriptor (such as Semaphores)
010	Invalid Simple Object Descriptor
011	Simple Object Descriptor
100	Invalid Paged Object Descriptor
101	Paged Object Descriptor
110	Invalid Bipaged Object Descriptor
111	Bipaged Object Descriptor

The following object table entries are collectively called storage descriptors:

- [Invalid] Simple Object Descriptor
- [Invalid] Paged-Object Descriptor
- [Invalid] Bipaged Object Descriptor

Valid storage descriptors contain physical addresses. Invalid storage descriptors, where the base address field may not necessarily be valid, are used to indicate that the selected object cannot be accessed.

The term object descriptor and object table entry are no longer distinguished.

8.7.5.1 Storage Descriptors

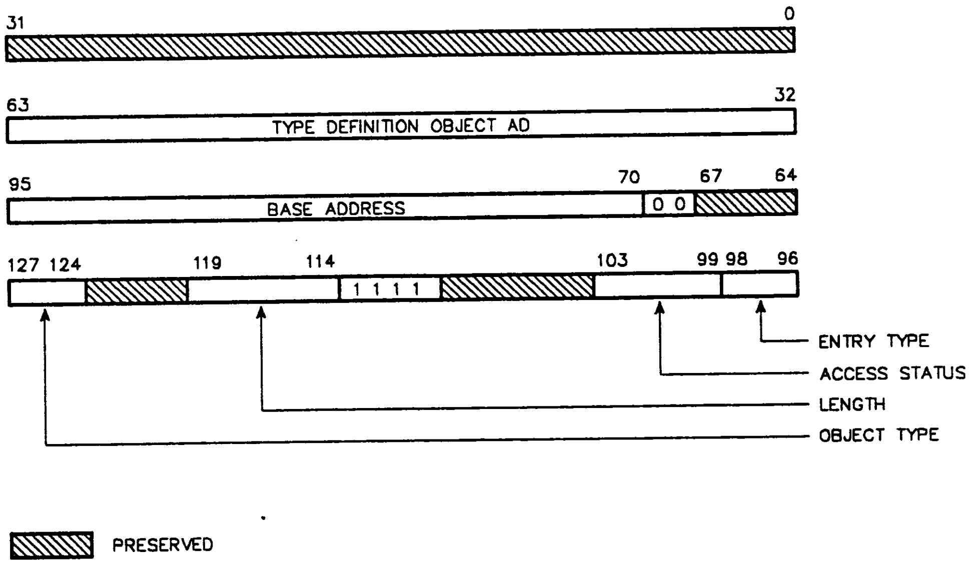


Figure 8-6. Storage Descriptors

The fields of a [invalid] simple object descriptor, a [invalid] paged object descriptor, or a [invalid] bipaged object descriptor are defined as follows:

- **TDO AD** (bits 32-63). This field contains the type definition object AD associated with this object descriptor. The TDO is described in Chapter 9.
- **Reserved** (bits 68-69). This field must be zero.
- **Base Address** (bits 70-95). This 26-bit field contains the physical base address (in units of 64 bytes) of the block, page table or page table directory. This provides a 2^{32} byte physical address space. This field is uninterpreted in an invalid storage descriptor.
- **Entry Type** (bits 96-98). This 3-bit field indicates the type of object table entries and the definition of the rest of the descriptor. The (binary) encodings for the three types of OTE are:

010	Invalid Simple Object Descriptor
011	Simple Object Descriptor
100	Invalid Paged Object Descriptor
101	Paged Object Descriptor
110	Invalid Bipaged Object Descriptor
111	Bipaged Object Descriptor

- **Access Status** (bits 99-103). This 5-bit field is described in Section 8.7.5.2. This field is defined only in a simple object descriptor. This field is preserved for other entry types.
- **Object Length** (bits 114-119). This field contains one less than the length in units of 64 bytes of the storage block referenced by the base address field.

  In a simple object, this field contains one less than the length in units of 64 bytes defined by this descriptor.

  In a paged object descriptor, this field contains one less than the length in units of 64K bytes defined by this descriptor.

  In a bipaged object descriptor, this field contains one less than the length in units of 64M bytes defined by this descriptor.

- **Object Type** (bits 124-127). This 4-bit field contains the object type of the object. Object types are described in Chapter 9.

8.7.5.2 Access Status

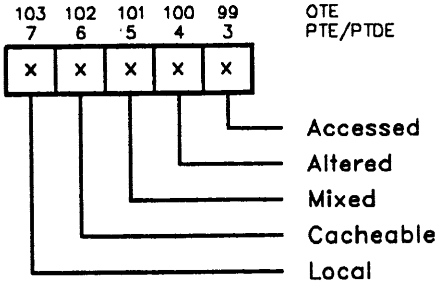


Figure 8-7. Access Status

An access status contains information for the management of blocks and pages. It is found in simple object descriptors and valid page table entries. This field does not appear in an invalid object descriptor, a paged/bipaged object descriptor, nor a page table directory entry.

The fields of an access status are defined as follows:

- **Accessed** (bit 99 in OTE, Bit 3 in PTE). This bit indicates the object or page defined with this descriptor has been referenced (read or written). This bit is ensured to be 1 before the associated storage is referenced. Once set, this bit remains set until cleared by software.
- **Altered** (bit 100 in OTE, Bit 4 in PTE). This bit indicates the object or page defined with this descriptor has been overwritten. This bit is ensured to be 1 before the associated storage is overwritten. Once set, this bit remains set until cleared by software.
- **Mixed** (bit 101 in OTE, bit 5 in PTE or PTDE). This bit indicates that an AD has been written in the object or page defined by this descriptor. This bit is ensured to be 1 before the associated storage is overwritten with a non-zero tag bit. Once set, this bit remains set until cleared by software.
- **Cacheable** (bit 102 in OTE, Bit 6 in PTE). This bit indicates the object or page defined with this descriptor can be cached. The encodings of the cacheable bit are as follows:

0	Do Not Cache
1	Can Be Cached

  Cacheability is described in Chapter 9.
- **Local** (bit 103 in OTE, bit 7 in PTE). This bit indicates the lifetime of the object or page defined by this descriptor. This is 0 for a global object or page and 1 for a local object or page.

8.7.5.3 Embedded Descriptors (Semaphores)

An embedded descriptor holds special predefined data structures. The only such predefined data structure is a semaphore, defined in Chapter 7.

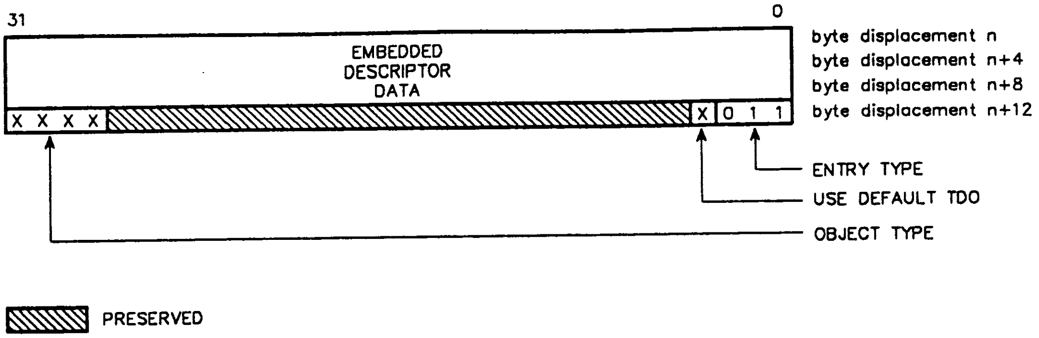


Figure 8-8. Embedded Descriptor

The fields of an embedded descriptor are defined as follows:

- **Storage Area** (bits 0-95). This 12-byte area contains the data structure.
- **Entry Type** (bits 96-98). This field is 001_2 for an embedded descriptor.
- **Use Default TDO** (bit 99). If set, this descriptor has an associated default TDO (see Chapter 9). If clear, the TDO AD is assumed to be in bits 32-63 of the descriptor. Typically, this bit would be set.
- **Type** (bits 124-127). This field is the 4-bit type value. The only predefined value is 0100_2, which indicates that the first three words hold a semaphore.

8.7.5.4 Invalid Object Table Entry

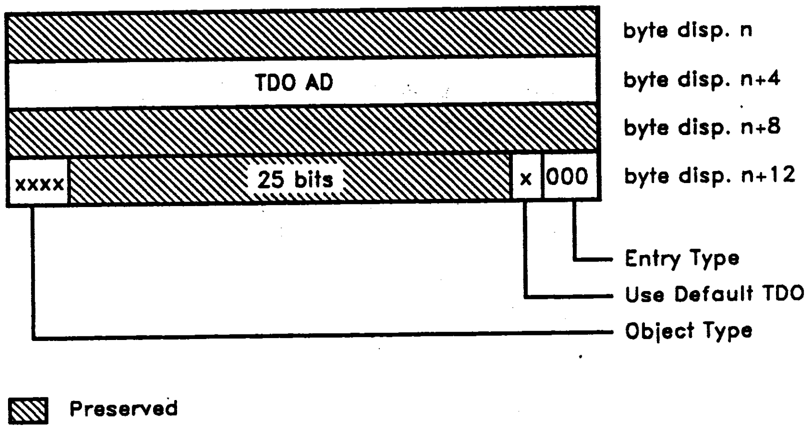


Figure 8-9. Invalid Object Table Entry

The fields of an invalid object table entry are defined as follows:

- **TDO AD** (bits 32-63). This field has the same interpretation as in a storage descriptor, but only if flag *use-default-TDO* is clear.
- **Entry Type** (bits 96-98). This field is 000_2 for an invalid object table entry.
- **Use Default TDO** (bit 99). If set, this descriptor has an associated default TDO (see Chapter 9). If clear, the TDO AD is assumed to be in bits 32-63 of the descriptor.

- **Object Type** (bits 124-127). This field has the same interpretation as in a storage descriptor.

8.7.6 Page Table Entries

A page table or page table directory contains an array of 4-byte page table [directory] entries of similar format. Page table entries in a page table directory specify page tables while page table entries in a page table specify pages.

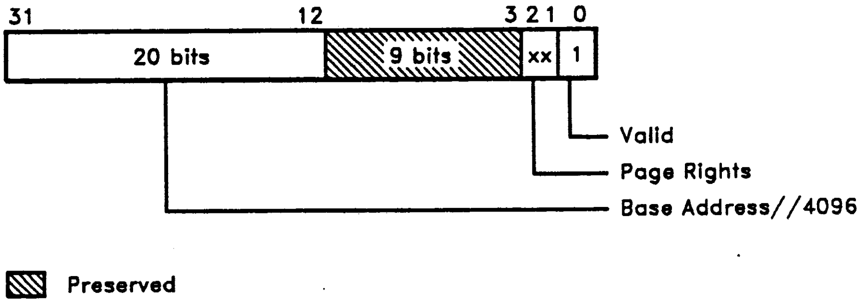


Figure 8-10. Page Table Directory Entry

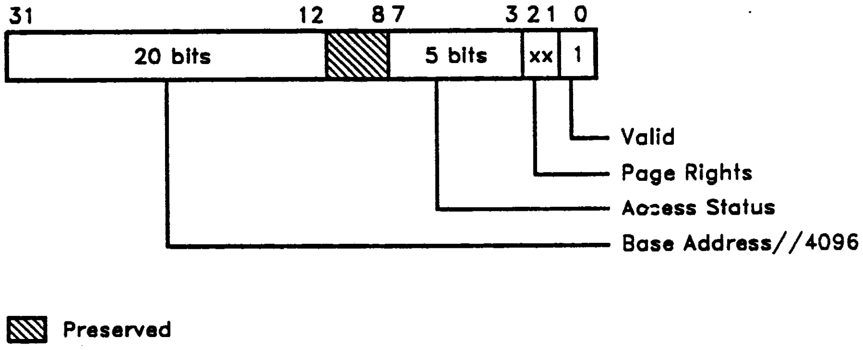


Figure 8-11. Page Table Entry

The fields of a valid page table entry or page table directory entry are defined as follows:

- **Valid** (bit 0). This bit is 1 to indicates a valid page table entry or page table directory entry.
- **Page Rights** (bits 1-2). This 2-bit field encodes the permissible operations (read or write) in different execution mode on the content of this page (in a page table entry) or for the pages defined by this page table (in a page table directory entry). Since a page may be controlled by more than one set of page rights, the effective rights is the minimum of all page rights. See Chapter 9.
- **Access Status** (bits 3-7). This 5-bit field is similar to that in a storage descriptor and is defined in the Section 8.7.5.2. This field is defined for a page table entry and is preserved for a page table directory entry.
- **Base Address** (bits 12-31). This 20-bit field contains the physical base address (in units of 4K-byte pages) of the page.

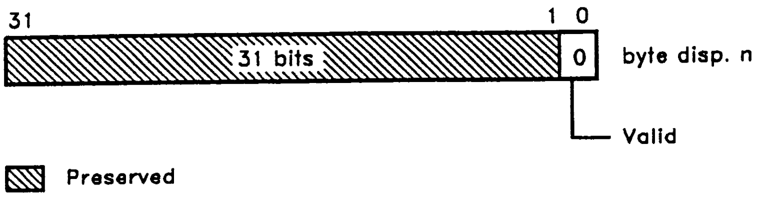


Figure 8-12. Invalid Page Table [Directory] Entry

The field of an invalid page table [directory] entry is defined as follows:

- **Valid** (bit 0). This bit is 0 to indicate an invalid page table [directory] entry.

8.8 Instructions

8.8.1 Access Descriptor Manipulation

Access descriptors, as opposed to data words, can be moved or copied only by a set of special instructions that guarantee the integrity of the object index and the access rights, and satisfy the object lifetime and the mutator function. Before an AD is copied, a lifetime check is performed. If the AD is valid and the mutator function is enabled at either the processor or process level, gray marking is performed.

ldm
ldml
ldmq
stm
stml
stmq
movm
movml
movmq

The move instructions copy an operand from register(s) to register(s). The load instructions read an operand from memory. The store instructions write the content of registers to memory; lifetime checking and gray marking are performed.

An AD in any accessible object will prevent the referenced object from being garbage collected. ADs which are no longer needed can be deleted by overwriting the AD with any data value.

The addresses of all mixed operands must be word aligned. Otherwise, a load mixed instruction becomes the equivalent of the corresponding load instruction, and so on.

These instructions can be used to store data values too, but will always be slower than the corresponding store instructions. When a mixed record (contains both data and ADs) exists where the internal layout is unknown, these instructions should be used.

In a record with both access descriptors and data, the ADs should be grouped together at the beginning of the record if possible. This avoids introducing extra gaps when ADs are intermixed with data. This also allows using a mixed instructions for the access parts and ordinal instructions for the data parts.

8.8.2 Object Reference Testing

cmpm
chktag

The **cmpm** instruction compares both ADs or data for equality. If both operands are ADs, the instruction tests whether they reference the same object. If both operands are data, the instruction tests whether the data value are equal. The **chktag** instruction checks for the tag bit.

In an untagged system in supervisor mode, the tag bit is assumed to be set in the **cmpm** instruction.

In the Ada expression "access_type_variable = null" where an AD is used to represent an access variable, the **chktag** instruction should be used instead of the **cmpm** instruction with zero because any non-zero data values cannot be used to reference an object.

8.8.3 Access Descriptor Creation

To facilitate proper lifetime checking, it is presumed that access descriptors are not haphazardly created.

However, in certain controlled situations (such as object allocation), system software needs to create an AD for an existing object.

cread

The **cread** instruction converts a data word to an AD.

8.8.4 Object Addressing Instructions

ldphy

The **ldphy** instruction returns the physical address of the operand.

8.9 Object Characteristics

8.9.1 Frozen

A number of predefined objects are required to be **frozen**, which means that no virtual memory fault can be raised during the access of some predefined part of these objects. The object table entry, page table directory entries and page table entries associated with the predefined area cannot be set to make the object inaccessible. This includes the object table's object descriptor, the page table directory, the page table entry and the page when the object descriptor is located. The software defined part of the predefined objects need not be frozen. In some cases, the objects are required to be frozen only when the object is in certain states. When these objects are relocated in memory, the normal mechanism for object relocation does not work for these objects. The following are a list of objects required to be frozen:

- Gray Table in region 3 (Chapter 8)
- Environment Table Object (Chapter 7)
- Port Object (Dispatching) (Chapter 14)

- Process Object (Chapter 15)
- Interrupt Environment Table (Chapter 16)
- Interrupt-related stack/code/data PTE (Chapter 16)

For any process in the executing, ready, or blocked state (see Chapter 15), the object descriptors of its regions must be marked as valid (that is, the V flags must be set).

8.10 Virtual Address Translation

8.10.1 Object Offset Translation

A memory request specifies the following information:

- Access Descriptor
- Object Offset
- Read/Write
- Length of Request

A single operand in an instruction may require one or more memory requests to fetch the operand from memory. An example of this is the **movstr** instruction where the number of memory requests depends on one of the source operands. Each of these memory requests goes through the same address translation operation described below.

The following describes the address translation of a virtual address to a physical address. Bounds and rights checking are included.

1. Compute the last byte of the memory request by adding the request length to the object offset.
2. If the memory request spans a 16-byte address boundary, perform the following:
   a. Split the request into two requests which do not span a 16-byte boundary.
   b. Perform the request as two separate memory requests.
3. Determine the rep rights needed by the request type.
4. Raise a *Rep-Rights* fault if the rights needed are not presented in the read and write rights of the AD.
5. Read the object table entry selected by the object index of the AD.
6. Raise an *Invalid Descriptor* fault if the entry type is 000_2 or 001_2.
7. Raise one of the *Virtual Memory* faults if the object table entry is not a valid storage descriptor.
8. If the object table entry is a simple object descriptor,
   a. Set the accessed bit of the simple object descriptor atomically if the accessed bit is 0.
   b. Set the altered bit of the simple object descriptor atomically if this is a write/RMW operation and the altered bit is 0.
   c. Set the mixed bit of the simple object descriptor atomically if this is a write/RMW operation and the memory request is mixed.

d. Raise an *Object Length* fault if the offset of the memory request is greater than the object length in the object descriptor.

e. Add the object offset to the base address in the object descriptor to form the physical address of the memory request.

If the object table entry is a paged object,

a. Raise an *Object Length* fault if the directory index and page index of the memory request is greater than the object length in the paged object descriptor.

b. Scale the page index (bits 12-21) by 4 and add it to the base address in the object descriptor to form the physical address of the selected data page table entry.

c. Read the page table entry and raise an *Invalid PTE* fault if the page table entry is invalid.

d. Raise a *Page Rights* fault if the rights needed is greater than the page rights.

e. Set the accessed bit of the page table entry atomically if the accessed bit is 0.

f. Set the altered bit of the page table entry atomically if this is a write/RMW operation and the altered bit is 0.

g. Set the mixed bit of the page table entry atomically if this is a write/RMW operation and the memory request is mixed.

h. Concatenate the page offset (bits 0-11) to the base address in the page table entry to form the physical address of the memory request.

If the object table entry is a bipaged object,

a. Raise an *Object Length* fault if the directory index of the offset of the memory request is greater than the object length in the bipaged object descriptor.

b. Scale the directory index (bits 22-31) by 4 and add it to the base address in the object descriptor to form the physical address of the selected page table directory entry for a page table.

c. Read the page table directory entry and raise an *Invalid PTDE* fault if the page table directory entry is invalid.

d. Raise a *Page Rights* fault if the rights needed is greater than the page rights.

e. Set the accessed bit of the page table directory entry atomically if the accessed bit is 0.

f. Set the mixed bit of the page table directory entry atomically if this is a write operation and any tag bit is 1.

g. Scale the page index (bits 12-21) by 4 and concatenate it to the base address in the page table directory entry to form the physical address of the selected page table entry for a page.

h. Read the page table entry and raise an *Invalid PTE* fault if the page table entry is invalid.

i. Raise a *Page Rights* fault if the rights needed is greater than the page rights.

j. Set the accessed bit of the page table entry atomically if the accessed bit is 0.

k. Set the altered bit of the page table entry atomically if this is a write operation and the altered bit is 0.

l. Set the mixed bit of the page table entry atomically if this is a write operation and the memory request contains valid access descriptors.

m. Concatenate the page offset (bits 0-11) to the base address in the page table entry to form the physical address of the memory request.

8.10.2 Caching of Address Translation Information

All valid/defined fields in valid page table entries and storage descriptors with storage allocated can be cached for read references. See Chapter 9 on caching of the contents of an object. Concurrent updates to these memory locations are not guaranteed to be reflected in the cached copies within a finite period of time. Interprocessor messages and instructions are provided to support software management of address translation information, as described in Chapter 16.

Note that caching of addressing translation information is independent of the cacheable bits for the object table.

8.10.3 Spanning Page Boundaries

When an access spans across the 2^{32} boundary, the address wraps around to zero.

Page boundaries are completely transparent. If a memory write overlaps a page boundary, that page is not re-read after it is written because of a virtual memory fault or a protection fault. (The correct support of copy-on-write semantics requires the page boundaries to be completely transparent.)

TYPE MANAGEMENT AND ACCESS CONTROL 9

This chapter defines object typing and access control.

9.1 Typed Objects

The two typing mechanisms are system-type and extended-type. They are not mutually exclusive. System types can be viewed as predefined properties of the object, while the extended type provides an unique identifier of the type and allows for type-specific software defined functions. The system-type mechanism provides predefined type-specific instructions, which require operands of these specific system types. The extended-type mechanism supports user defined types. This allows software-defined type-specific operations and extended-typed-specific operations to verify an object's type before carrying out their prescribed functions. This facility supports the type manager style of programming.

The type definition object contains a reference to a domain that provides some type-specific procedures.

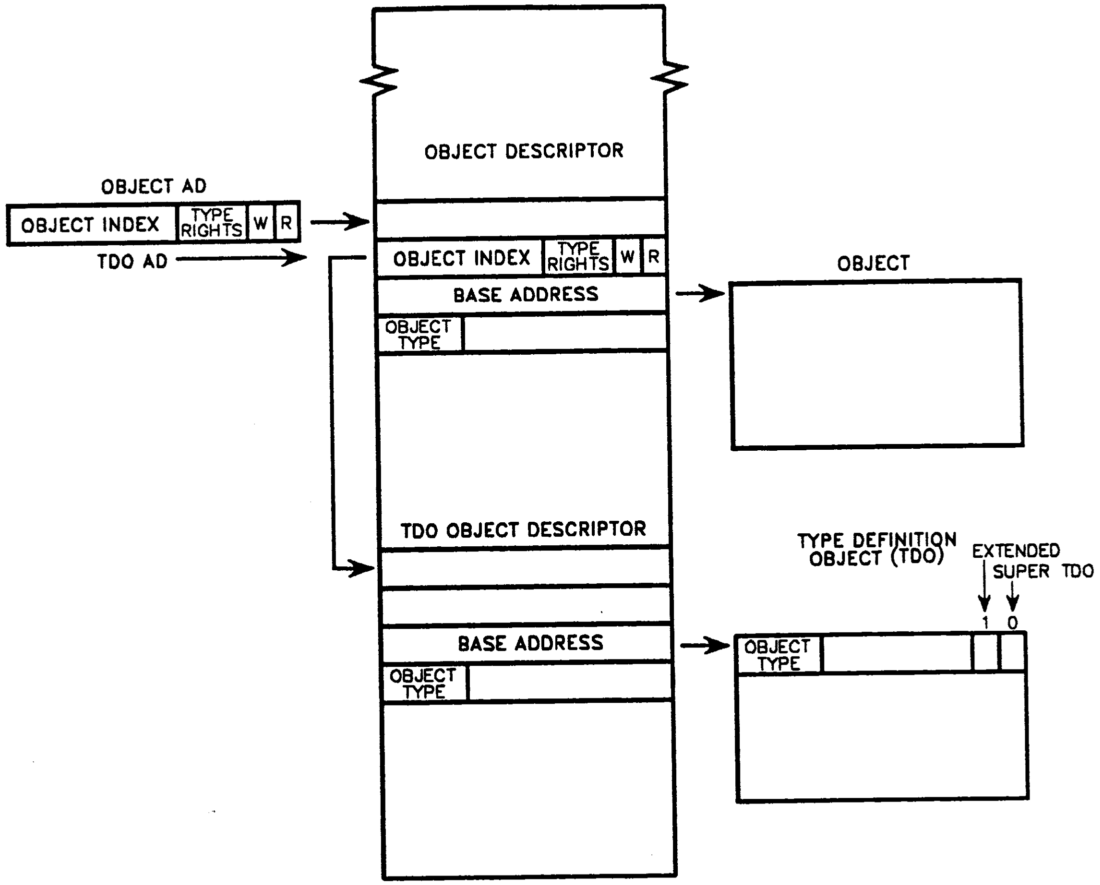


Figure 9-1. Object Typing

9.1.1 Object Type Field

The system type of an object is specified by the object type field in its object descriptor (see Chapter 8). One of the system types is generic that contains no predefined fields.

The encodings for the Object Type field are as follows:

Encoding (in binary)	**Object type**
0000	Generic
0001	Type Definition Object
0010	Process Object
0011	Domain Object
0100	Semaphore (Embedded Object)
0101	Port Object
0110-0111	(reserved)
1000-1111	available for system software

9.1.2 Type Definition Object (TDO)

The extended type of an object is specified by the type definition object's access descriptor associated with the object descriptor, either explicitly or via a default (see Chapter 8). If a default TDO is to be used, the object index for the default TDO is 16 plus the type encoding of the object or embedded descriptor. For instance, for a semaphore, the default TDO is entry 20 in the object table.

A type definition object contains information to manage the instances of a particular object type.

The object index of the TDO can also be used as a unique identifier for the extended type. In this manner, up to 2^{26} (64M) extended types may be defined.

For system objects, while the object type is used to indicate the set of type specific instructions, the type definition object can be used to provide software (that is, procedural) extensions to the predefined type-specific operations.

Different type definition objects may be associated with different instances of the same system object type. This permits software extensions to the predefined instructions to be defined on a per instance basis.

ldtdo

The **ldtdo** instruction copies the type definition object AD associated with the object referenced by the source AD into a register destination. If the object specified has a default TDO, an AD (with no type nor rep rights and specifying global lifetime) to the default TDO is returned.

9.2 Rights

The architecture uses various rights bits to restrict the way in which an object or an AD may be manipulated. There are two sets of rights bits: one set is found in an AD, and the other set is located in the various page table entries in the access (that is, address translation) path.

Different ADs can have different access rights to the same object. Rights on an access path are shared among all users of the access path. An AD can be easily duplicated with the same or less rights than the source AD. Since an access path cannot be similarly duplicated, any change to rights in the access path affect all users of the access path.

An AD contains the following rights bits:

- Type Rights
- Read Rights
- Write Rights

A page table entry in an access path contains the following rights bits:

- Page Rights

9.2.1 Type Rights

The type rights of an AD define the permissible type-specific operations for the object referenced by the AD. The interpretation of the type rights bits of an AD depends on the type of object referenced. The type rights bits for a generic object are uninterpreted. The interpretation of the type rights bits for each system object are predefined. They are described in the individual system object descriptions. The uninterpreted type rights bits are preserved for software-defined type rights and interpreted by individual software-level type managers.

9.2.2 Read and Write Rights in Access Descriptors

The read rights control reading the content of the referenced object, while the write rights control writing. The actual permissible operations are also optionally determined by the page rights in page tables. This allows the support of inaccessible objects, read-only objects, write-only objects, and read-writeable objects.

There are no "execute" rights; only read rights are required to execute an instruction in the execution environment.

9.2.3 Page Rights

The page rights in a valid page table entry define the read/write rights of the page in a bipaged object or a paged object. Page rights are used to permit software-defined areas or regions of the execution environment to have different access protection.

Page rights are interpreted differently depending on the execution mode of the current process. Page rights are defined as follows:

	Execution Mode	
Rights	User Mode	Supervisor Mode
00	no access	read-only
01	no access	read-write
10	read-only	read-write
11	read-write	read-write

Page rights, instead of AD read/write rights, are used to protect the instruction areas of a linear address space from accidental modification.

When the processor is in physical-addressing mode (address translation is disabled), rights checking is disabled.

9.2.3.1 Effective Access Rights

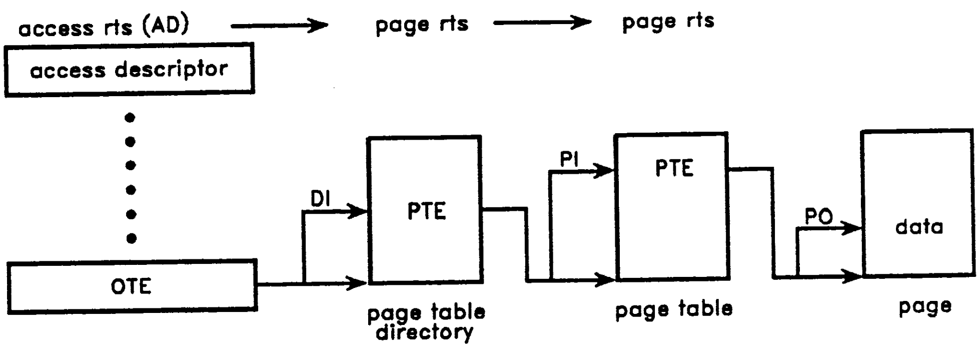


Figure 9-2. Effective Rights

As described above, the access rights for an item are defined by rights fields in each level of the access path. Each access path contains the following:

1. Read and write rights in an AD.
2. Page rights in the page table directory entry for a bipaged object.

3. Page rights in the page table entry for a paged object or a bipaged object.

The effective rights for an object address are the minimum of the rights in the access path.

inspacc

The **inspacc** instruction returns the effective page rights of the access path specified by the source address.

9.3 Type Definition Object

A type definition object is used to control access rights amplification. A type definition object has a predefined system type.

The type rights in an AD for a type definition object are defined as follows:

Use — Unintepreted.

Modify — **Amplify Rights:** If the bit is 1, the TDO may be used in the **amplify** instruction. (In general, only the type manager for a type has an AD with this right enabled.)

Control — **Create Rights.** If the bit is 1, the TDO may be used in the **cread** instruction. (In general, only the memory manager of an operating system should have an AD for a TDO with this right enabled.)

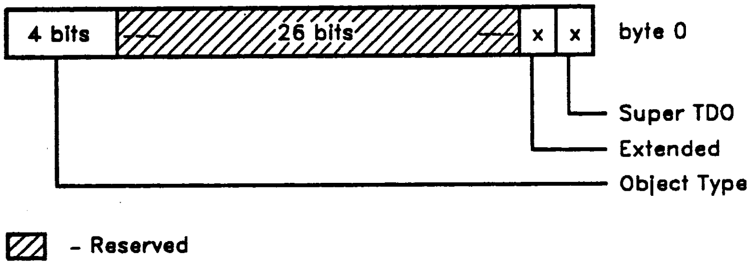


Figure 9-3. Type Definition Object

The fields of a type definition object are defined as follows:

- **Super TDO** (bit 0). This bit is interpreted during rights amplification as follows:

  0 — This TDO can be used to amplify ADs for objects whose type matches that specified by this TDO.

  1 — This TDO can be used to amplify any AD. Thus, this TDO should be available only to the memory manager of an operating system.

- **Extended** (bit 1). This bit is 1 if the TDO is used to manage objects having this object as their TDO ADs. This bit is 0 if the TDO is used to manage objects having the same object type as specified in the TDO.

  This bit also allows a single TDO to manage objects of the specified object type independent of their TDO ADs.

- **Object Type** (bits 28 - 31). This 4-bit field is defined to have the same format as the corresponding field in an object descriptor. In the **amplify** instruction, this field is compared against the object type of the referenced object if the extended type and the super TDO bits are zero.

9.3.1 Rights Manipulation

Some instructions increase or decrease the access rights of an AD. A rights mask is specified during rights amplification and restriction. A rights mask has the same format as an access descriptor, except that the local bit, the object index field, and the tag bit are not used. Access rights are amplified by logically-ORing the access rights in the AD with those in the rights mask. Access rights are reduced by logically-ANDing the access rights in the AD with the complement of that in the rights mask.

amplify
restrict

The **amplify** instruction requires a type definition object AD with amplify rights. The amplify instruction amplifies the source AD and stores the amplified AD in the destination. If the super-tdo bit is 0, and the extended bit is 1 (as it is typically), the object referenced by the source AD must reference an OTE that contains a matching TDO AD. Thus, a type manager may amplify only ADs of objects managed by that type manager.

The **restrict** instruction removes the rights of the source AD specified by the rights mask and stores the restricted AD in the destination.

The accessibility of an addressing environment may be restricted to those objects with the minimum required access rights needed for the execution of the program. Instructions are provided to remove (or restrict) access rights that are not needed. Access restriction is typically performed before an AD is passed as a parameter to a procedure or returned as a result.

Certain (system or extended) typed objects are manipulated exclusively by their corresponding type managers. An AD to a typed object, outside the domain of its type manager, usually has limited access rights (for example, no access or read-only) to prevent access to or modification of the object without the knowledge of the type manager. The type manager restricts (removes) the access rights in the ADs for objects of the type(s) under its control before passing them outside its domain. When such an AD is returned to the type manager, the access rights are amplified (increased) while inside the domain to allow modification of the referenced object by the type manager.

FAULTS 10

This chapter describes the fault handling facilities. The subjects covered include the fault-handling data structures, the software support required for fault handling, and the fault handling mechanism. A reference section that contains detailed information on each fault type is provided at the end of the chapter.

10.1 Overview of the Fault-Handling Facilities

Various conditions may arise that require exceptional software treatment. These situations, detected in programs or the machine state, are called faults. Some of these faults may represent error conditions in the programs: for example, overflow that occurs during both integer and floating-point arithmetic. Other faults may represent infrequent events that require software intervention: for example, a virtual memory fault to invoke the operating system software to bring the page in from swapping devices. In general, these faults are associated with the execution of an instruction. A few of these faults are independent of the execution of the current instruction: for example, the time-slice fault and event-notice fault can occur anytime.

A fault is generally handled with a fault-handling procedure, called the "fault handler". The fault handler is invoked through an implicit procedure call. Information about the state of the process and the fault are made available to the fault handler in a data structure called the fault record.

If the fault handler is able to recover from the fault, the process can be restored to its state prior to the fault and resumed. If, on the other hand, the fault represents a program error, the faulting record allows the debugger or the language-defined exception handler to gain control of the program.

10.2 Fault Types

All of the faults are divided into types and subtypes. The fault type selects a fault handler. The fault subtype may be used by the fault handler to select a specific fault-handling action.

Table 10-1 lists the faults by type and subtype. For convenience, individual faults are referred to in this manual by their fault-subtype name. Thus a *machine bad-access fault* is referred to as simply a *bad-access fault*, or a *virtual-memory, invalid page-table-directory-entry fault* is referred to as an *invalid PTDE fault.*

The fault encoding column of Table 10-1 shows each fault type/subtype word as it appears in the fault record (at offset FP-8).

Table 10-1. Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Encoding
Hex	Name	No./Bit Position	Name	Hex
1	Trace	Bit 1	Instruction Trace	xx01 xx02
		Bit 2	Branch Trace	xx01 xx04
		Bit 3	Call Trace	xx01 xx08
		Bit 4	Return Trace	xx01 xx10
		Bit 5	Prereturn Trace	xx01 xx20
		Bit 6	Supervisor Trace	xx01 xx40
		Bit 7	Breakpoint Trace	xx01 xx80
2	Operation	1	Invalid Opcode	xx02 xx01
		4	Invalid Operand	xx02 xx04
		6	Subsystem Not Found	xx02 xx06
3	Arithmetic	1	Integer Overflow	xx03 xx01
		2	Arithmetic Zero-Divide	xx03 xx02
4	Floating Point	Bit 0	Floating Overflow	xx04 xx01
		Bit 1	Floating Underflow	xx04 xx02
		Bit 2	Floating Invalid-Operation	xx04 xx04
		Bit 3	Floating Zero-Divide	xx04 xx08
		Bit 4	Floating Inexact	xx04 xx10
		Bit 5	Floating Reserved-Encoding	xx04 xx20
5	Constraint	1	Constraint Range	xx05 xx01
		2	Invalid AD	xx05 xx02
6	Virtual Memory	1	Invalid Object-Table-Entry	xx06 xx01
		2	Invalid Page-Table-Directory-Entry (PTDE)	xx06 xx02
		3	Invalid Page-Table-Entry (PTE)	xx06 xx03
7	Protection	Bit 0	Lifetime	xx07 xx00
		Bit 1	Object Length	xx07 xx01
		Bit 2	Page Rights	xx07 xx02
		Bit 3	Rep-Rights	xx07 xx03
		Bit 4	Type Rights	xx07 xx04
8	Machine	1	Bad Access	xx08 xx01
9	Structural	1	Control	xx09 xx01
A	Type	1	Type Mismatch	xx0A xx01
		2	Contents	xx0A xx01
B	Control Stack	1	Control-Stack Overflow	xx0B xx01
		2	Control-Stack Underflow	xx0B xx02
C	Process	1	Time Slice	xx0C xx01
D	Descriptor	1	Invalid Descriptor	xx0D xx01
E	Event	1	Event Notice	xx0E xx01

For some fault types, each subtype is assigned a separate bit. It is possible for more than one of these bits be set if they are detected at the same time. This does not guarantee that all possible faults will be detected and reported at the same time.

10.3 Fault Masking

Certain faults have associated masks that can prevent the fault from being signaled. Some of these faults have associated "sticky" flags which are set when a masked fault has been occurred. (A "sticky" flag is a flag that is reset only by explicit instructions to the word in which the flag is located.) Table 10-2 lists these masks and sticky flags, the system data structures in which they are located, and the fault subtype they affect.

Table 10-2. Fault Flags or Masks

Flag or Mask Name	Location	Fault Affected
Integer Overflow Mask	Arithmetic Controls	Integer Overflow
Floating Overflow Mask	Arithmetic Controls	Floating Overflow
Floating Underflow Mask	Arithmetic Controls	Floating Underflow
Floating Invalid Operation Mask	Arithmetic Controls	Floating Invalid Operation
Floating Zero-Divide Mask	Arithmetic Controls	Floating Zero-Divide
Floating-point Inexact Mask	Arithmetic Controls	Floating Inexact
No Imprecise Faults	Arithmetic Controls	All Imprecise Faults
Trace-Enable	Controls	All Trace Faults
Trace-Mode Flags	Trace Controls	All Trace Faults
Event-Fault Mask	Environment Table	Even Notice Fault

The integer and floating-point mask bits inhibit faults from being signaled for specific fault conditions (that is, integer overflow and floating-point overflow, underflow, zero divide, invalid operation, and inexact). The use of these masks is discussed in Section 10.11. Section 5.10 describes the floating-point fault masks.

The no-parallel-faults flag controls the synchronization of faults. See Section 10.10.4 for details.

The trace-enable bit in the process controls can disable all trace faults. The trace modes in the trace controls enable only the selected trace faults if they are detected. See Chapter 11 for details.

The event-fault disable bit in the subsystem ID field of the subsystem table temporarily postpones the signaling of an event fault while executing in a subsystem with the event-fault disabled.

10.4 Fault Information

10.4.1 Fault Record

When a fault condition is detected, execution of the current instruction is terminated. The state and the cause of the fault condition are placed in a fault record. The fault record can be considered as the parameters for the fault handling procedure.

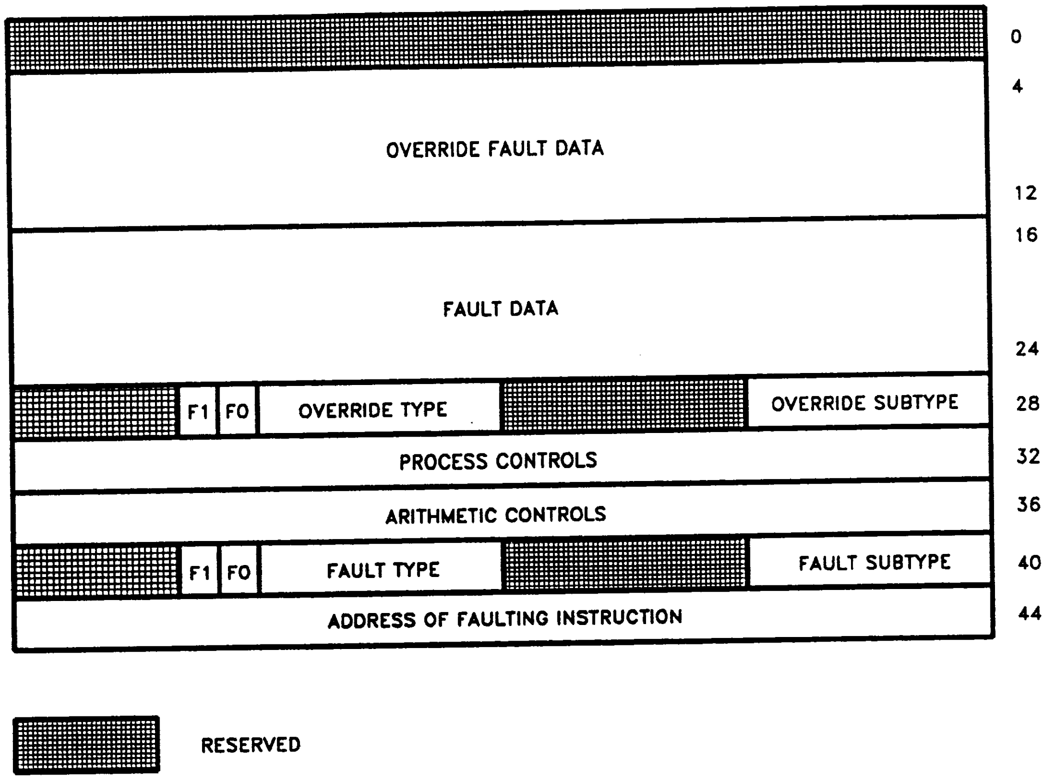


Figure 10-1. Fault Record

The fault information is a function of the type of the fault. Only the fields necessary to describe a fault are stored for a particular fault. The fault and resumption records are stored "beneath" the stack frame of the fault handler (at lower memory addresses). The fault-specific interpretation of this record is given in Section 10.11. The general format of the format record is shown in Figure 10-1. The fields of a fault record are defined as follows:

- **Address of Faulting Instruction** (bytes FP-4 to FP-1). This field contains the linear address of the faulting instruction (the instruction that caused the fault or that was being executed when the fault occurred). In some cases, this field may be undefined.
- **Fault Flags** (byte FP-5). Depending on the type of fault, these flags specify further information about the fault. The only flags defined are F0 (bit 24) and F1 (bit 25). If a fault is not defined as altering these flags, their value is undefined.
- **Fault Subtype** (byte FP-6). Depending on the type of fault, this byte ordinal specifies further information on the cause of the fault; that is, the subtype.
- **Fault Type** (byte FP-8). This byte ordinal indicates the fault type.
- **Process Controls** (bytes FP-12 to FP-9). The value of the process controls at the time when the fault is generated. The process controls are restored to this value on a return from the fault handler.
- **Arithmetic Controls** (bytes FP-16 to FP-13). The value of the arithmetic controls at the time when the fault is generated. The arithmetic controls is restored to this value on a return from the fault handler.

- **Override Flags** (byte FP-17). See Section 10.6.2.
- **Override Subtype** (byte FP-18). See Section 10.6.2.
- **Override Type** (byte FP-20). See section 10.6.2.
- **Fault Data** (bytes FP-32 to FP-21). This field depends on the type of the fault and is defined in detail in later sections. Any part of this field that is not specified for a particular fault has an undefined value.
- **Override Fault Data** (bytes FP-44 to FP-33). See Section 10.6.2.
- **Resumption Record** (bytes FP-64 to FP-47). If an instruction is suspended (neither completed nor aborted) as the result of a fault, 16 bytes of additional information are saved with the fault record, called the resumption record. This is similar to the data associated with an interrupt record. The size and content of the resumption record are processor-defined.

10.4.2 Saved Instruction Pointer

The saved IP, the RIP in L2 of the stack frame where the fault occurred, is also part of the saved fault information. This points to the instruction to be executed on the returned from the fault handler. The saved IP either points to the faulted instruction, the next instruction to be executed if the fault had not occurred, or undefined. Normally, if the execution of the faulting instruction has completed (like an arithmetic fault), the saved IP points to the next instruction. If the execution of the faulting instruction is aborted (like a virtual memory fault), the saved IP points to the faulting instruction so it will be re-executed on return from the fault handler.

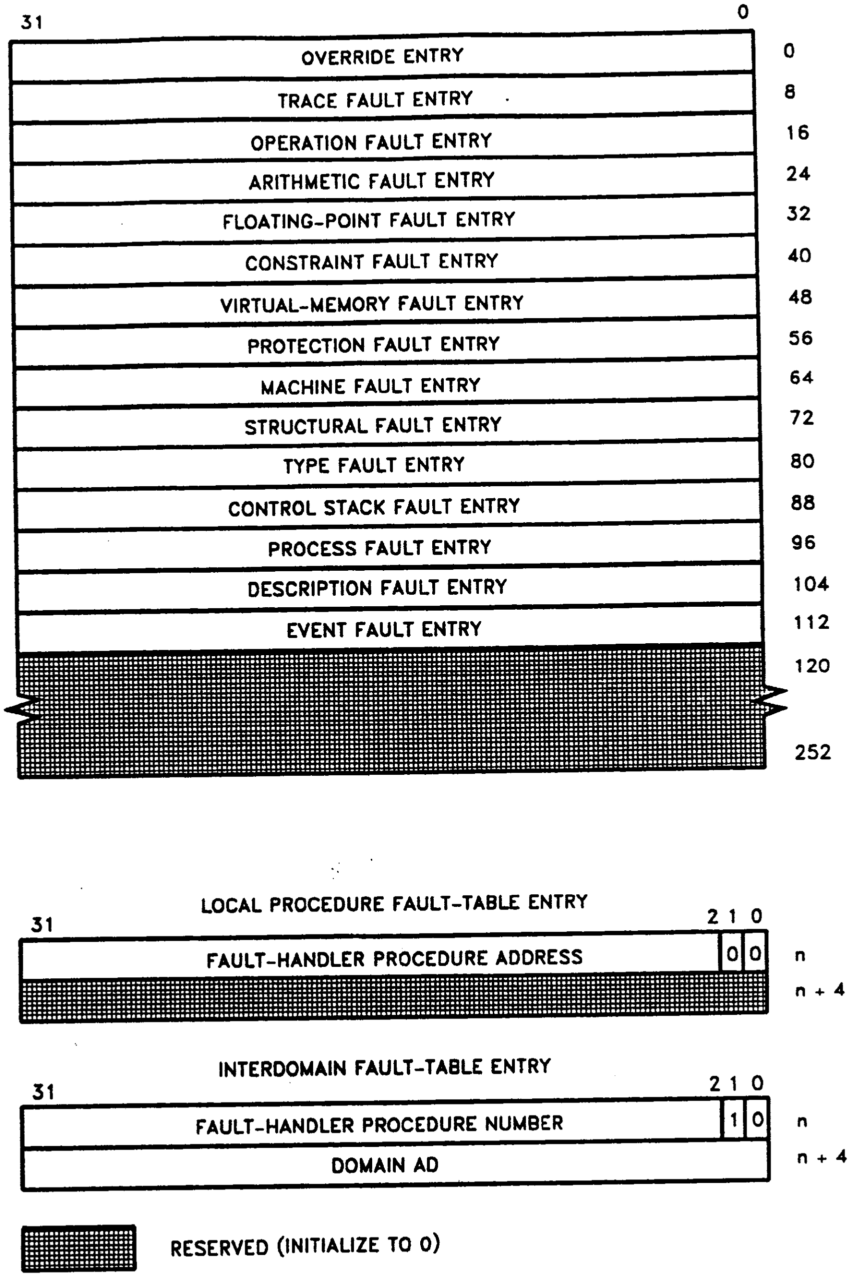
10.4.3 Fault and Resumption Records in Process and Processor Objects

Normally, the fault and resumption records are stored below the stack frame of the fault handler. When a system-error interrupt occurs, the system-error fault records are stored in the processor object at byte offset 128-175 as shown in Figure 10-1. Additionally, the system-error-fault field (at offset 72-75 in the processor object) contains the fault type/subtype of the system-error fault.

The fault record in the process and processor objects are processor defined. The resumption record in these objects is used on implicit returns (fault and interrupt returns). The fault and resumption records in the processor object are used when there is no process associated with the processor.

10.5 Fault Table

The fault table directly or indirectly specifies the fault handler for each fault type. The physical address of the fault table is specified in the processor object. As shown in Figure 10-2, the fault table contains one entry for each fault type plus an entry for overrides. When a fault occurs, the fault type is used to select a fault table entry, which selects the different types of implicit fault call mechanisms to be used in reporting this fault.



31 — 0	
OVERRIDE ENTRY	0
TRACE FAULT ENTRY	8
OPERATION FAULT ENTRY	16
ARITHMETIC FAULT ENTRY	24
FLOATING-POINT FAULT ENTRY	32
CONSTRAINT FAULT ENTRY	40
VIRTUAL-MEMORY FAULT ENTRY	48
PROTECTION FAULT ENTRY	56
MACHINE FAULT ENTRY	64
STRUCTURAL FAULT ENTRY	72
TYPE FAULT ENTRY	80
CONTROL STACK FAULT ENTRY	88
PROCESS FAULT ENTRY	96
DESCRIPTION FAULT ENTRY	104
EVENT FAULT ENTRY	112
▨	120
▨	252

LOCAL PROCEDURE FAULT-TABLE ENTRY

31	2	1 0	
FAULT-HANDLER PROCEDURE ADDRESS		0 0	n
▨			n + 4

INTERDOMAIN FAULT-TABLE ENTRY

31	2	1 0	
FAULT-HANDLER PROCEDURE NUMBER		1 0	n
DOMAIN AD			n + 4

▨ RESERVED (INITIALIZE TO 0)

Figure 10-2. Fault Table and Fault-Table Entries

10.5.1 Fault-Table Entries

Each entry in the fault table is two words long, as shown at the bottom of Figure 10-2. The fields of a fault type entry are defined as follows:

- **Entry Type** (bits 0-1). This field specifies the type of implicit call used to reported the fault. The encodings of this field are as follows:

00	perform an implicit call_extended operation.
01	reserved.
10	perform an implicit call_domain operation.
11	reserved.

- **Procedure Offset/Number** (bits 2-31). If the entry type is 0, this field contains the word address of the fault handler procedure. If the entry type is 2, this field contains the procedure number to be used in the implicit **calld** operation. This field is reserved for the other entry types.
- **Domain AD** (bits 32-63). If the entry type is 2, this field contains the domain AD to be used in the implicit **calld** operation. This field is reserved for the other entry types.

10.6 Fault Levels

Faults are handled at one of the following levels dependent on when the fault is detected:

- Implicit procedure call to the primary fault handler
- Implicit procedure call to the override fault handler
- Implicit interrupt call to the system-error interrupt handler
- Halt

The four fault levels provide a mechanism for recovering from faults or for gradually degrading processing activity when serious or catastrophic fault conditions are encountered. The scenario for handling faults with this mechanism is as follows.

10.6.1 Primary Fault Handling

When a fault occurs during the execution of an instruction, a fault handler for that fault type is selected and invoked from a data structure called the *fault table*.

As a result of calling primary fault handler, a fault record is created. This record includes the type and subtype of the fault and information on the state of the process when the fault occurred. If the fault occurred while in the midst of executing an instruction, a resumption record for the instruction may also be included with the fault record. This fault record is stored on the stack of the fault handler.

10.6.2 Overrides

If a fault occurs while performing the implicit call to the primary fault handler, an override occurs. When an override faults, the fault record contains additional information associated with the override faults.

A common override condition is a virtual-memory fault on the fault handler's stack while trying to store the fault record or allocating a stack frame for the fault handler. The override fault data contains the address of the stack, and the override fault handler performs the actions in the virtual memory fault handler to swap in the missing page. On the return from the override fault handler, the primary fault is automatically refaulted so that it can then be handled.

10.6.3 System-Error Interrupt

If another fault occurs while performing the implicit call to the override fault handler, the second fault is handled by means of a system-error interrupt. The fault and resumption records for the primary and override faults are stored in the designated area in the processor object. The fault type and subtype of the system-error fault is also stored in the processor object. Interrupt number 248 is reserved for the system-error interrupt.

10.6.4 Halt

If another fault occurs while performing the implicit interrupt to the system-error interrupt handler, the processor enters the stopped state and halts.

There is no fault information associated with the fault that causes the processor to halt. The fault and resumption information associated with the primary, override and system-error faults in the processor object may be undefined if the last fault occurs before these information can be stored.

10.7 Fault-Handler Procedures

The fault-handling mechanism supports four types of fault-handler procedures:

- Local procedures (either through a local procedure entry in the fault table or a domain call that points to a local procedure entry in the domain object)
- Supervisor procedures
- Intrasubsystem procedures
- Intersubsystem procedures

Local and supervisor procedures are generally located in region 3 of the linear address space, so they are always accessible, regardless of whether a process is bound to the processor or not. Otherwise, regions 0-2 are not defined when a fault occurs in a interrupt handler while the processor is idle (or does not have a process).

Subsystem fault-handler can be either intrasubsystem or intersubsystem procedure (see Chapter 7).

Supervisor or subsystem fault-handler procedures must be used if the execution of the normal instruction stream is to be continued after a return from the fault handler. A local fault handler does not modify the trace enable bit on call, nor does it restore the process controls on the return from a local fault handler.

10.7.1 Fault-Handler Invocation

When a fault occurs, the following steps are taken:

1. If system-error reporting is in progress, put the processor into the stopped state.
2. If override bit in the saved process controls is set, store the fault records into the system-error fault record in the processor object, and the system-error fault type/subtype into the system-error-fault field in the processor object. System-error reporting is now in progress. Perform an implicit system-error interrupt.

3. If primary fault reporting is in progress, the override fault handler entry is used. The overrides fields of the fault record composed according to the specific override fault. The refault and resume bit in the saved process controls are set.
4. If primary fault reporting is not in progress, the fault type is used as an index to an entry in the fault table. The process controls and arithmetic controls are saved. A fault record is composed according to the specific fault. Primary fault reporting is now in progress.
5. If the fault was a virtual-memory, object-length, page-rights, event-notice, or a time-slice fault occurred within an instruction that was suspended as a result, the resume flag is set in the saved process controls. Some bits in the processor-defined field may also be set in the saved process controls to specify different type of resumption actions.
6. Depending on the selected fault table entry, an implicit *callx* or *calld* operation is performed (see detail descriptions of *callx* and *calld* in Chapter 18), but with the following exceptions:
7.

- The return status of the fault-handler frame is different.

	Normal	Implicit
Local	000	001
Domain		
local	000	001
supervisor	01T	001
intrasubsystem	100	100
intersubsystem	101	101

- The return mode in the control stack entry of the fault-handler subsystem call is different.

	Normal	Implicit
Intrasubsystem	000	100
Intersubsystem	001	101

- An extra 64 bytes are allocated below the fault-handler stack frame. So instead of adding 63 to the stack pointer of the target stack where the fault-handler will run) before rounding, 63 plus the size of the combined fault and resumption record is added to the stack pointer. Note that the total size of the fault and resumption records is processor-defined, and may vary from release to release.
- When the fault-handler stack frame is allocated, the fault record is stored at NFP-48 to NFP-1 where NFP is the frame pointer of the fault-handler stack frame.
- If there is resume bit in the saved process controls is set, the resumption record is stored at NFP-64 to NFP-49.

10.7.2 Fault Return

The return operation performs the following additional actions if the frame status is 001:

The arithmetic-controls field at FP-12 is stored in the process's arithmetic controls. If the execution mode is supervisor, the process-controls field at FP-16 is stored in the process's process controls, and if the resume flag is set, the resumption record at FP-48-M is copied into the process's resumption record.

The restoring of the process controls affects how tracing occurs for this instruction; see Section 11.4.5. If the refault flag was set in the process controls in the fault record, additional actions may be performed (see Section 10.9).

10.7.3 Subsystem Fault Return

The return operation performs the following additional actions if the return status is $10x_2$ (subsystem return) and the return mode in the control-stack-entry is $10x_2$.

Same as the above, except that all is done regardless of the execution mode. If the refault flag was set in the process controls in the fault record, additional actions may be performed (see Section 10.9).

10.7.4 Returning Without Resumption

A fault handler may return to other than the point of the fault by first altering the return IP in the previous frame. This could lead to unpredictable behavior if resumption information is present with the fault. To perform such a return predictably, one should clear the following information in the process-controls field in the fault record before the return: resume flag, refault flag, trace-fault-pending flag, internal-state field.

10.7.5 System-Error Interrupt Action

When a system-error interrupt occurs, data is collected on the faults that caused the condition and the system-error interrupt fault handler is invoked. No mechanism, however, is provided for resuming the process, once the handling of the interrupt is complete.

When a system-error interrupt occurs as the result of a second override fault, the following actions are taken:

1. The fault records for both the primary fault and the override fault are stored in the system-error-fault-record field in the processor object.
2. The type and subtype of the system-error fault are stored in the system-error fault field in the processor object.
3. The interrupt stack is selected.
4. An implicit call operation to vector 248 (or $F8_{16}$, the predefined system-error interrupt vector) in the interrupt table is initiated.

10.7.6 Halt Action

When a fault occurs while reporting the system-error interrupt, the following actions are taken:

1. The processor places itself in the stopped state and asserts the #FAILURE pin.

10.8 Process State After a Fault

As described earlier, faults can occur prior to the execution of the faulting instruction, during the instruction, or after the instruction. When the fault occurs before the faulting instruction is executed, the instruction can theoretically be executed on the return from the fault handler. So, the fault is not accompanied by a change in process state.

When a fault occurs during or after the instruction that caused a fault, the fault may be accompanied by a change in the process state such that the faulting instruction cannot be reexecuted. For example, when an integer-overflow fault occurs, the overflow value is stored in the destination. If the destination register was the same as one of the source registers, the source value is lost, making it impossible to reexecute the faulting instruction.

In general, process changes never accompany the following fault types or subtypes:

- All Operation Subtypes
- Arithmetic Zero-Divide
- All Floating-Point Subtypes Except Floating Inexact
- Constraint Range
- Prereturn Trace
- Control Stack Underflow
- All Descriptor Subtypes

Process state changes always accompany the following fault types and subtypes:

- All Trace Subtypes Except Prereturn Trace
- Integer Overflow
- Floating Inexact
- Control Stack Overflow

Process state changes may or may not accompany the following fault types and subtypes:

- All Virtual Memory Subtypes
- Time Slice
- Event Notice
- Invalid AD
- All Structural Subtypes
- Bad Access
- All Protection Subtypes
- All Type Subtypes

If a fault occurs while an object is locked as part of the operation, the object is unlocked before the fault is handled to prevent deadlocks.

The effect that specific fault types have on a process is given in Section 10.11.

10.9 Refault Operation

The resume and refault bits in the process controls are set in the saved process controls when an override fault handler is invoked. These bits are used to indicate to the return from fault operation to report the primary fault before the faulted instruction stream is to be resumed.

If the resume and refault bits are set in the saved process controls on a fault or subsystem fault return, the fault and resumption records are saved before the fault-handler's frame is deallocated. At the end of a fault (from supervisor mode) or a subsystem return, the refault bit (and under certain conditions, the resume bit) is cleared. The primary fault record in the fault record is reported before any instructions in the faulted stack frame is executed.

The refault mechanism can be used by software explicitly. A primary fault handler may decide that a different fault handler should be used to handle a specific fault. For example, a page rights protection fault may signify a copy-on-write operation which is more appropriately handled by the virtual memory fault handler. To perform an explicit refault, a fault hander must change the primary fault type and set the resume and refault bits of the saved process controls. The return from fault handler operation will signal the new primary fault with the rest of the fault record.

If a refault must be signaled from outside a fault handler, do the following:

1. Call a supervisor or subsystem procedure, so it is possible to fake a return from fault operation.
2. Call a local procedure, so the space for the previous frame can be used for a fault record.
3. Execute the **flushreg** instruction.
4. Copy the previous frame pointer of the previous frame to L0 to delete the previous frame.
5. Change the return status in L0 to fault call (if this operation is performed in supervisor mode) or the return mode in the control stack entry to the corresponding fault return (if this operation is performed in a subsystem procedure).
6. Execute the **flushreg** instruction.
7. Compose the fault record. Set the resume and refault bit in the saved process controls.
8. Execute the **ret** instruction.

10.10 Multiple Events

10.10.1 Faults and Interrupts

If an interrupt occurs when a fault is being reported the interrupt may be handled in either of the following ways:

- The fault information is recorded as part of the resumption record, and the interrupt is serviced immediately. On the return from the interrupt, the fault is reported as though the interrupt has not occurred.
- The interrupt is delayed until the fault reporting has completed, but before the first instruction in the fault handler is executed.

10.10.2 Control Stack Overflow or Trace Fault on a Fault Call

A control-stack overflow fault detected at the end of a subsystem fault call is not considered an override condition because the subsystem fault call is considered completed.

A trace fault signaled as a result of a fault call is not considered an override condition because the fault call is considered completed. fault call is considered completed.

10.10.3 Multiple Fault Conditions

It is possible for multiple fault conditions to be associated with a single instruction. This should be distinguished from parallel fault conditions associated with different instructions when they are executed in parallel. In some cases when the subtype field allocates a bit position for each subtype, multiple faults of the same fault type may be reported at the same time. In other cases when the multiple faults have different fault type, the fault mechanism selects any one of the multiple faults and ignores the rest. The fault mechanism is only required to report a single fault at a time unless explicit specified by the specific fault, like floating-point faults.

10.10.4 Parallel Faults

In some instances, the processor is able to execute instructions concurrently. When faults occur, faults may be signaled out of order, making it different from a serial instruction execution model. When two instructions are being executed concurrently, it is also possible for them to generate faults simultaneously.

Two mechanisms are provided to allow the circumstances under which faults are signaled to be controlled. These mechanisms are: (1) the no parallel faults flag in the arithmetic controls and (2) the syncf instruction.

Faults are grouped into the following categories: synchronized, parallel, and asynchronous.

Synchronized faults are those that are intended to be recoverable by software. For any instruction that can generate a synchronized fault, the processor will (1) not execute the instruction if an unfinished prior instruction will fault and (2) not execute subsequent out-of-order instructions that will fault. The following faults are always synchronized:

- trace
- virtual memory
- protection
- control stack
- descriptor faults

Parallel faults are those that in some instances are allowed to occur and be signaled out of order. These faults include the following:

- operation
- arithmetic
- floating-point
- constraint
- structural
- type

Asynchronous faults have no direct relationship to the current executing instruction. This category includes the machine, event, and process faults.

The no-parallel-faults flag controls whether or not parallel faults are allowed. When this flag is set, all faults must be synchronized. In this mode, the ability to execute instructions concurrently is essentially disabled. All faults that occur are signaled.

When the flag is clear, faults in the parallel category can in some instances occur in parallel or out of order. In this mode, the following conditions hold true:

1. When a parallel fault occurs, the saved IP points to the instruction to be executed next after all the parallel faults are handled.

The **syncf** instruction guarantees no faults associated with all previous instruction execution can be signaled after the **syncf** instruction. One use is to force faults to be synchronized when the no-parallel-fault flag is clear. The other use is to insure that all instructions are complete and all faults signaled in one block of code before execution of another block of code (for example, on Ada block boundaries when the blocks have different exception handlers).

The intent of these fault-generating modes is that compiled code should execute with the no-parallel-faults flag is clear, using the **syncf** instruction where necessary to ensure that faults occur in order.

10.11 Fault Reference

This section describes each of the fault types and subtypes and gives detailed information about what is stored in the various fields of the fault record. The section is organized alphabetically by fault type.

Each of the fault descriptions includes the following topics:

- **Fault Type and Subtype**
  The fault-type section gives the number entered in the fault-type field of the fault record for the given fault type. The fault-subtype section lists the fault subtypes and their associated number or bit position in the fault-subtype field of the fault record.
- **Function**
  The function section gives a general description of the purpose of the fault type, then describes the purpose of each of the fault subtypes in detail. It also describes how each fault subtype is handled.
- **Fault Flags**
  Fault Data
  Addr. Fault. Inst.
  The fault record section describes how the flags, fault-data, and address-of-faulting-instruction fields of the fault record are used for the fault type and subtypes.
- **Saved IP**
  The saved IP section describes what value is saved in the RIP register (L2) of the stack frame in which the fault occurred.
- **State Changes**
  The process state changes section describes the effects that the fault subtypes have on the state of the process.

10.11.1 Arithmetic Faults

Fault Type: 3_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Integer Overflow
2	Arithmetic Zero-Divide
3-F	Reserved

Function: This fault type applies only to integer, ordinal, floating-point to integer/ordinal conversion instructions, but not floating-point-only instructions.

The integer-overflow fault occurs when the result of an integer instruction exceeds the range of the destination and the integer-overflow mask in the arithmetic-controls register is cleared. Normally, the *n* least significant bits of the result are stored in the destination, where *n* is the destination size.

The arithmetic zero-divide fault occurs when the divisor of an ordinal/integer divide/remainder/modulo instruction is zero.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

State Changes: When an integer-overflow fault occurs, the instruction has been completed and the truncated result has been stored in the destination before the fault is signaled.

There is no state changes associated with a zero-divide fault.

10.11.2 Constraint Faults

Fault Type: 5_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Constraint Range
2	Invalid AD
3-F	Reserved

Function: The constraint-range fault occurs when a fault-if instruction is executed and the condition code in the arithmetic controls matches the condition specified by the instruction.

The invalid-AD fault occurs when an instruction attempts to reference a object by means of an AD with the tag bit zero, or the execution mode is user when tagging is disable.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP of the instruction which faulted.

Saved IP: Not used.

State Changes: There is no state changes associated with a constraint-range fault.

There is no state changes associated with an invalid-AD fault if the AD is specified as an operand or in a domain. Otherwise, the accomplished state changes are unpredictable. The latter could only occurs if system data structures are incorrectly setup.

10.11.3 Descriptor Faults

Fault Type: D_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Invalid Descriptor
2-F	Reserved

Function: when an AD points to a object descriptor that has an invalid type

when an AD points to a object descriptor that is an embedded type, but the descriptor is not being used in a semaphore operation.

Fault Flags: Not used.

Fault Data: The virtual address is stored in the first two words of the fault-data field.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: Same as the address-of-faulting-instruction field.

State Changes: If there is any state changes associated with an invalid-descriptor fault, sufficient resumption information will be saved to make the fault recoverable on a return from the fault handler.

10.11.4 Event Faults

Fault Type: E_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Event Notice
2-F	Reserved

Function: when a process is dispatched and the event-fault-request flags in the process object are setwhile the event-fault disable bit is 0.

when a process notice IAC is received and the event-fault-request flags in the process object are set while the event-fault disable bit is 0.

when an intersubsytem call/return which changes the event-fault disable bit from 1 to 0 and the event-fault-request flags (may be cached) are set.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: Not used.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

State Changes: If this fault occurs while a process is being dispatched, the fault is signaled before work on the process begins. This allows the fault handler to either never begin work on the process or to return to the process and begin work on it.

If this fault occurs while an instruction is being executed, the processor does one of the following: (1) terminates the instruction as if it had not yet begun execution, (2) completes execution of the instruction, or (3) suspends the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

If there is any state changes associated with an event fault, sufficient resumption information will be saved to make the fault recoverable on a return from the fault handler.

10.11.5 Floating-Point Faults

Fault Type: 4_{16}

Fault Subtype:	**Bit Number**	**Name**
Bit 0	Floating Overflow	
Bit 1	Floating Underflow	
Bit 2	Floating Invalid-Operation	
Bit 3	Floating Zero-Divide	
Bit 4	Floating Inexact	
Bit 5	Floating Reserved-Encoding	
Bits 6 and 7	Reserved	

Function: Each floating-point fault is assigned a bit in the fault-subtype field. Multiple floating-point faults can occur simultaneously, but only the floating-inexact faults can occur with either the floating-overflow or floating-underflow faults.

The floating-point faults are described in detail in Chapter 5. The following paragraphs give a brief description of each floating-point fault.

A floating-overflow fault occurs when the floating-point overflow mask is 0 and the rounded infinitely precise result of a floating-point instruction exceeds the largest finite value of the destination format. This fault can occur with the floating-inexact fault (as described in Chapter 5).

A floating-underflow fault occurs when the floating-point underflow mask is 0 and the rounded infinitely precise result of a floating-point instruction is less than the smallest normalized value of the destination format. This fault can occur with the floating-inexact fault (as described in Chapter 5).

The floating invalid-operation fault occurs when the floating-point invalid-operation mask is 0 and one of the source operands for a floating-point instruction is a NaN or inappropriate for the type of operation being performed.

The floating zero-divide fault occurs when the floating-point zero-divide mask is 0 and an exact infinite result would be produced from finite operands.

The floating-inexact fault occurs when the floating-point inexact mask is 0 and an infinitely precise result cannot be encoded in the format specified for the destination operand. This fault interacts with the floating-overflow and floating-underflow faults (as described in Chapter 5).

The floating reserved-encoding fault occurs when the normalizing-mode bit in the arithmetic controls is 0 and a denormalized value is used as an operand in a floating-point instruction, or an unnormalized extended-real value is used.

Fault Flags: **F0** -- Used if inexact fault occurs in conjunction with overflow or underflow fault. If set, F0 indicates that the adjusted result has been rounded toward +∞; if clear, F0 indicates that the adjusted result has been rounded toward -∞.

F1 -- Used with overflow and underflow faults only. If set, F1 indicates that the adjusted result has been bias adjusted, because its exponent was outside the range of the extended-real format.

Fault Data: Used only with overflow and underflow faults. Adjusted result is stored in this field in extended-real format.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

State Changes: State changes accompany the floating-overflow, floating-underflow, and floating-inexact faults, because a result is stored in the destination before the fault is signaled. The faulting instruction can thus not be reexecuted.

No state changes do not accompany the floating invalid-operation, floating zero-divide, and floating reserved-encoding faults, because the faults occur before the execution of the faulting instruction.

10.11.6 Machine Faults

Fault Type: 8_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Bad Access
2-F	Reserved

Function: Indicates that the processor has detected a hardware or memory-system error.

The bad-access fault is the only one of this fault type. This fault occurs whenever an unrecoverable memory error occurs on a physical memory operation.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: Not used.

Saved IP: Not used.

State Changes: This fault may occur at any time. When it does occur, the accompanying state of the process is undefined. As a result, the processor is not able to return predictably from the fault handler to the point in the process where the fault occurred.

If this fault occurs during an atomic operation, there is no guarantee that the locking mechanism that the memory subsystem uses for synchronization is unlocked.

10.11.7 Operation Faults

Fault Type: 2_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Invalid Opcode
2 - 3	Reserved
4	Invalid Operand
5	Reserved
6	Subsystem Not Found
7 - F	Reserved

Function: The invalid-opcode fault occurs when the processor attempts to execute an instruction that contains an undefined opcode or addressing mode.

The invalid-operand fault occurs when the processor attempts to execute an instruction for which one or more of the operands have special requirements and one or more of the operands do not meet these requirements. This fault subtype is not generated on floating-point instructions.

The subsystem-not-found fault occurs when during an intersubsystem call, the target subsystem is not found in the subsystem table. This fault will not occur if in the in terrupted state because there is all intersubsystem calls are handled as intrasubsystem calls.

Fault Flags: Not used.

Fault Data: For the subsystem-not-found fault, the first word contains the subsystem ID; not used for other faults.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: For subsystem-not-found fault, IP for the faulting instruction; not used for other faults

State Changes: No state changes associated with this fault.

10.11.8 Process Faults

Fault Type: C_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Time Slice
2-F	Reserved

Function: The time-slice fault occurs when an end-of-time-slice event occurs and the time-slice-reschedule flag in the process-controls is 0.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: Not used.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

State Changes: Since this fault often occurs while an instruction is being executed, it is often accompanied by a process-state change. However, when the state does change, sufficient resumption information will be saved to make the fault recoverable on a return from the fault handler.

When the fault occurs, the processor does one of the following: (1) terminates the instruction as if it had not yet begun execution, (2) completes execution of the instruction, or (3) suspends the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

10.11.9 Protection Faults

Fault Type: 7_{16}

Fault Subtype:	Bit Number	Name
Bit 0	Lifetime	
Bit 1	Object Length	
Bit 2	Page Rights	
Bit 3	Rep Rights	
Bit 4	Type Rights	
Bit 5 - 7	Reserved	

Function: Indicates that an instruction has attempted to violate the addressing-protection rules. Each protection fault is assigned a bit in the fault-subtype field. When multiple protection faults occur at the same time, the architecture is permitted, but not required, to indicate each fault.

The lifetime fault occurs when an attempt is made to copy an AD with its local bit set to 1 into an object or page with a local flag of 0.

The object-length fault occurs when a reference is made to object which falls beyond the length of the object, or when the OTE referenced by an AD falls beyond the length of the object table.

The page-rights fault occurs a reference is made to storage in a paged or bipaged object and the associated page-table-directory entry or page-table entry do not have the access rights needed by the reference under the current execution mode.

A rep-rights fault occurs when the AD explicitly or implicitly used to reference storage within an object does not have the representation rights needed by the reference.

The type-rights fault occurs when an object-operation instruction references an object with an AD with insufficient type rights for the operation.

The action that the processor takes when these faults occur allows the fault handler to modify the object table, page-table-directory, or page-table when appropriate to correct the fault condition, then resume work on the process from the point where the fault occurred.

Fault Flags: **F0** -- Used with page-rights and rep-rights faults only. If set, F0 indicates that an attempted write operation caused the fault; if clear, F0 indicates that an attempted read operation caused the fault.

F1 -- Not used.

Fault Data: For a page-rights or object-length fault, the first two words contains the virtual address of the attempted memory operation.

Addr. Fault. Inst.: IP of the instruction that faulted

Saved IP: Same as the address-of-faulting-instruction field.

State Changes: A process-state change accompanies each of the protection faults, however, sufficient state information is saved to permit either reexecution or completion of the faulting instruction on a return from the fault handler.

These faults occur while the faulting instruction is being executed. When the fault occurs, the processor will either (1) terminate the instruction as if it had not yet begun execution or (2) suspend the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

10.11.10 Structural Faults

Fault Type: 9_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Control
2-F	Reserved

Function: Indicates that the state of one of the system data-structures is preventing the processor from performing a system operation. Examples of things that can cause a structural fault include a pointer in one data structure to a non-existent data structure or invalid state information in a data-structure field. These faults often occur while the processor is performing an internal (implicit) operation and may not be related to a particular instruction.

The control fault occurs when the invalid contents of a data structure are preventing a fault or interrupt from being handled or when a fault occurs during the process of invoking an interrupt handler.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: Not used.

State Changes: When a structural fault occurs, the accompanying state of the process is undefined. The processor is thus not able to return predictably from the fault handler to the point in the process where the fault occurred.

10.11.11 Trace Faults

Fault Type: 1_{16}

Fault Subtype:

Bit Number	Name
Bit 0	Reserved
Bit 1	Instruction Trace
Bit 2	Branch Trace
Bit 3	Call Trace
Bit 4	Return Trace
Bit 5	Prereturn Trace
Bit 6	Supervisor Trace
Bit 7	Breakpoint Trace

Function: Indicates that the processor has detected one or more trace events. The processor's event tracing mechanism is described in detail in Chapter 11.

A trace event is the occurrence of a particular instruction or type of instruction in the instruction stream. The processor recognizes seven different trace events (instruction, branch, call, return, prereturn, supervisor, and breakpoint). It detects these events, however, only if a mode bit is set for the event in the process trace-controls word, which is cached in the processor chip. If, in addition, the trace-enable flag in the process controls is set, the processor generates a fault when a trace event is detected.

The fault is generated following the instruction that causes a trace event (or prior to the instruction for the prereturn trace event).

The following trace modes are available:

- **Instruction** -- Generate trace event following any instruction.
- **Branch** -- Generate trace event following any branch instruction when branch is taken.
- **Call** -- Generate trace event following any call or branch-and-link instruction, or implicit procedure call (i.e., call to fault or interrupt handler).
- **Return** -- Generate trace event following any return instruction.
- **Prereturn** -- Generate trace event prior to any return instruction.
- **Supervisor** -- Generate trace event following any call-system 6r call-domain instruction.
- **Breakpoint** -- Generate trace event following any processor action that causes a breakpoint condition.

There is a trace fault subtype and a bit in the fault-subtype field associated with each of these modes. Multiple fault subtypes can occur simultaneously, with the fault-subtype bit set for each subtype that occurs.

When a fault type other than a trace fault occurs during the execution of an instruction that causes a trace event, the non-trace-fault is handled before the trace fault. An exception to this rule is the prereturn trace fault. The prereturn trace fault will occur before the processor has a chance to detect a non-trace-fault, so it is handled first.

Likewise, if an interrupt occurs during an instruction that causes a trace event, the interrupt is serviced before the trace fault is handled. Again, the prereturn trace fault is an exception. Since it occurs before the instruction, it will be handled before any interrupt that might occur during the execution of the instruction.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP of the instruction that caused the trace event.

Saved IP: IP for the instruction that would have been executed next, if the fault had not occurred.

State Changes: A process state change accompanies all the trace faults (except the prereturn trace fault), because the events that can cause a trace fault occur after the faulting instruction is completed. As a result, the faulting instruction cannot be reexecuted upon returning from the fault handler.

Since the prereturn trace fault occurs before the **return** instruction is executed, a process state change does not accompany this fault and the faulting instruction can be executed upon returning from the fault handler.

10.11.12 Type Faults

Fault Type: A_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Type Mismatch
2	Contents
3-F	Reserved

Function: Indicates that the contents of a system-data structure or its descriptor are inconsistent with the operation that the processor is trying to perform.

The type-mismatch fault occurs when the type information in a object descriptor does not match the operation the processor is being asked to perform. For example, a type-mismatch fault occurs when the AD given in a resume-process instruction (**resumprcs**) does not point to a process object.

The contents fault occurs when the information in a object is not defined or is inconsistent.

Fault Flags: Not used.

Fault Data: Not used.

Addr. Fault. Inst.: IP of the instruction that faulted.

Saved IP: Not used.

State Changes: When a type fault occurs, the accompanying state of the process is undefined. The processor is thus not able to return predictably from the fault handler to the point in the process where the fault occurred.

10.11.13 Virtual-Memory Faults

Fault Type: 6_{16}

Fault Subtype:

Number	Name
0	Reserved
1	Invalid Object-Table-Entry
2	Invalid Page-Table-Directory-Entry (PTDE)
3	Invalid Page-Table-Entry (PTE)
4-F	Reserved

Function: Indicates that an address or an AD in an instruction cannot be translated into a physical address, because the object or page being referenced is not in physical memory.

The invalid-object-table-entry fault occurs when the valid flag in a object descriptor is 0, which can mean that the object, the page-table directory, or the page table that the object descriptor points to is not in physical memory.

The invalid-PTDE fault occurs when the valid flag in a page-table-directory entry is 0, which means that the page table that the entry points to is not in physical memory.

The invalid-PTE fault occurs when the valid flag in a page-table entry is 0, which means that the page that the entry points to is not in physical memory.

The action that the processor takes when these faults occur allows the fault handler to copy the missing object or page from the disk into physical memory, then resume work on the process from the point where the fault occurred.

Fault Flags: Not used.

Fault Data: The virtual address of the attempted memory operation.

Addr. Fault. Inst.: IP of the instruction that faulted

Saved IP: Same as the address-of-faulting-instruction field.

State Changes: A process-state change accompanies each of the virtual-memory faults, however, sufficient state information is saved to permit either reexecution or completion of the faulting instruction on a return from the fault handler.

These faults occur while the faulting instruction is being executed. When the fault occurs, the processor will either (1) terminate the instruction as if it had not yet begun execution or (2) suspend the instruction, saving the intermediate state in the resumption record. The instruction being executed determines which action is taken.

DEBUGGING AND TRACING SUPPORT 11

This chapter describes the tracing facilities, which allow the monitoring of instruction execution.

11.1 Overview of the Trace-Control Facilities

Trace events allow processor activity to be monitored. Some trace events occur just after having executed a particular instruction, while others occur after having executed one of a class of instructions, while still others occur just before executing a particular instruction.

By monitoring trace events, debugging software is able to display or analyze the activity of a program. This analysis can be used to locate software or hardware bugs or for general system monitoring during the development of system or applications programs.

The typical way to use this tracing capability is to enable certain trace events either by means of the trace-controls word or a set of breakpoint registers. An alternate method of creating a trace event is with the **mark** and **fmark** instructions. These instructions cause an explicit trace event to be generated whenever they are executed.

If tracing is enabled, a trace fault occurs at each trace event. The fault handler for trace faults can then call the debugging monitor software to display or analyze the state when the trace event occurred.

11.2 Required Software Support for Tracing

To use the tracing facilities, software must provide trace-fault handling procedures, perhaps interfaced with a debugging monitor. Software must also manipulate several control flags to enable the various tracing modes and to enable or disable tracing in general. These control flags are located in the system-data structures described in the next section.

11.3 Trace Controls

The following flags or fields control tracing:

- Trace controls
- Trace-enable flag in the process controls
- Trace-fault-pending flag in the process controls
- Trace flag (bit 0) in the return-status field of register l0
- Trace-control flag in the supervisor-stack-pointer field of the domain object

11.3.1 Trace-Controls Word

The trace-controls word is located in the PCB for the current process. When a process is bound to the processor, the contents of the trace-controls word are cached internally in the processor.

The trace controls allow software to define the conditions under which trace events are generated. Figure 11-1 shows the structure of the trace-controls word.

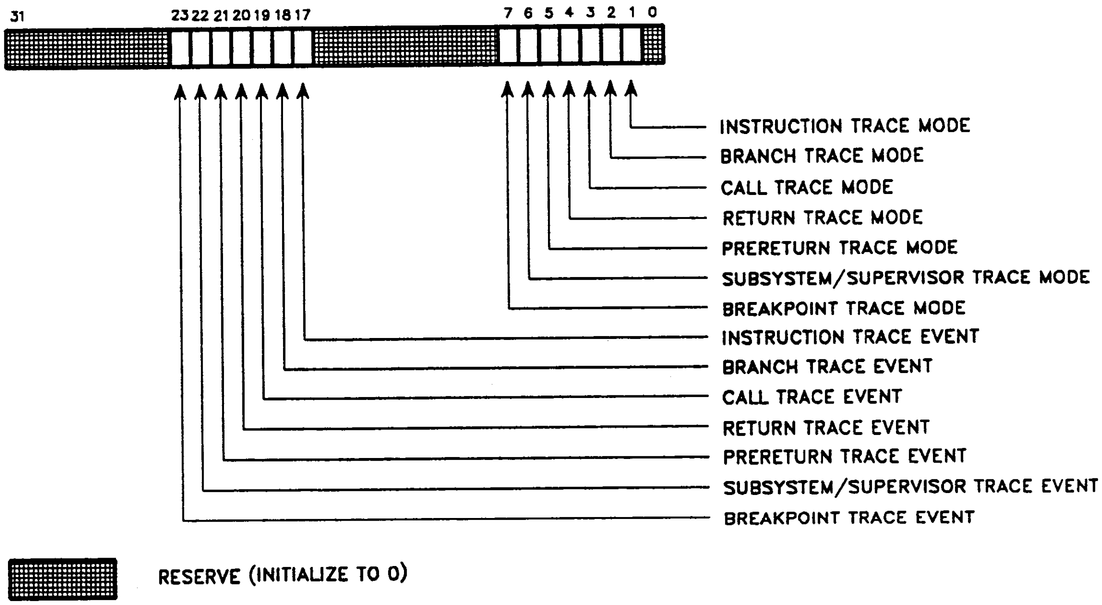


Figure 11-1. Trace-Controls Word

This word contains two sets of bits: the mode flags and the event flags. The mode flags define a set of trace modes that the processor can use to generate trace events. A mode represents a subset of instructions that will cause trace events to be generated. For example, when the call-trace mode is enabled, the processor generates a trace event whenever a call or branch-and-link operation is executed. To enable a trace mode, the kernel sets the mode flag for the selected trace mode in the trace controls. The trace modes are described later in this chapter.

The event flags keep track of which trace events (for those trace modes that have been enabled) have been detected.

A **modtc** instruction sets or clears flags flags in the trace controls. On initialization, all the flags in the internal trace controls are cleared. The **modtc** instruction can then set or clear trace mode flags as required. (This instruction does not affect the trace controls word in the PCB for the current process.)

Software can access the event flags using the **modtc** instruction. However, this is unnecessary, because these flags are modified directly by the trace-handling mechanism.

Bits 0, 8 through 16, and 24 through 31 of the trace controls are reserved. These bits should be initialized to zero and not accessed or modified after initialization.

11.3.2 Trace-Enable and Trace-Fault-Pending Flags

The trace-enable flag and the trace-fault-pending flag (in the process controls) control tracing. The trace-enable flag enables the tracing facilities. When this flag is set, trace faults are generated for all trace events.

Typically, software selects the trace modes to be used through the trace controls. It then sets the trace-enable flag when tracing is to begin. This flag is also altered as part of some of the call and return operations, as described at the end of this chapter.

The trace-fault-pending flag keeps track of the fact that an enabled trace event has been detected. This flag is used as follows. An enabled trace event sets this flag. Before executing each instruction, this flag is checked, and if set, a trace fault is generated. By providing a means of recording the occurrence of a trace event, the trace-fault-pending flag allows interrupts or other faults to be handled before handling the trace fault.

11.3.3 Trace Control on Subsystem and Supervisor Calls

The trace flag and the trace-control flag allow tracing to be enabled or disabled when a subsystem call (interdomain call) or a supervisor call is executed. This action occurs independent of whether or not tracing is enabled prior to the call.

When a subsystem or supervisor call is executed, the current state of the trace-enable flag (from the process controls) is saved into the trace flag (bit 0) of the return-status field of register l0.

Then, when the subsystem or supervisor procedure is selected from the procedure table in the domain object, the trace-enable flag in the process controls is copied from the setting in the trace-control flag in the domain object (bit 0 of the word that contains the supervisor-stack pointer).

On a return from the subsystem or supervisor procedure, the trace-enable flag in the process controls is restored to the value saved in the return-status field of register l0.

Thus, the trace-enable flag is established by the called domain, and not by the caller. However, the caller's trace flag is restored upon return from the called domain.

11.4 Trace Modes

The following trace modes can be enabled through the trace controls:

- Instruction trace
- Branch trace
- Call trace
- Return trace
- Prereturn trace
- Supervisor trace
- Breakpoint trace

These modes can be enabled individually or several modes can be enabled at once. Some of these modes overlap, such as the call-trace mode and the supervisor-trace mode. Section 11.7 describes what happens when multiple trace events occur.

The following sections describe each of the trace modes.

11.4.1 Instruction Trace

When the instruction-trace mode is enabled, each instruction generates an instruction-trace event before the instruction is executed. This mode can be used within a debugging monitor to execute a program a single step at a time ("single-step" the program).

11.4.2 Branch Trace

When the branch-trace mode is enabled, each branch instruction that successfully branches (not including unsuccessful conditional branches) generates a branch-trace event. Branch-and-link, call, and return instructions do not cause generate branch-trace events.

11.4.3 Call Trace

When the call-trace mode is enabled, every call instruction (**call**, **callx**, or **calls**), or a branch-and-link instruction (**bal** or **balx**), or an implicit call (such as invoking a fault or interrupt handler) generates a call-trace event.

During a call-trace event, the prereturn-trace flag (bit 3 of register l0) is set in the new frame created by the call operation or in the current frame if a branch-and-link operation was performed. This flag controls whether or not to signal a prereturn-trace event on a **ret** instruction.

11.4.4 Return Trace

When the return-trace mode is enabled, every **ret** instruction generates a return-trace event.

11.4.5 Prereturn Trace

The prereturn-trace mode causes any **ret** instruction to generate a prereturn-trace event prior to its execution, but only when the prereturn-trace flag in l0 is set. (Prereturn tracing cannot be used without enabling call tracing.)

The prereturn-trace flag is set whenever a call-trace event is detected. This flag performs a prereturn-trace-pending function. If another trace event occurs at the same time as the prereturn-trace event, the prereturn-trace flag allows the the non-prereturn-trace event to fault first, then the prereturn-trace event is faulted. The prereturn trace is the only trace event that can cause two successive trace faults to be generated between instruction boundaries.

11.4.6 Subsystem/Supervisor Trace

When the subsystem/supervisor-trace mode is enabled, the subsystem/supervisor-trace event is generated any time (1) an implicit or explict domain call is executed; (2) an explicit domain call is made, where the procedure table entry is a supervisor procedure; or (3) when a **ret** instruction is executed and the return-status field is set to 010_2 or 011_2 (that is, return from supervisor mode).

This trace mode allows a debugging program to determine the boundaries of operating-system calls within the instruction stream.

11.4.7 Breakpoint Trace

The breakpoint-trace mode allows trace events to be generated at places other than those specified with the other trace modes. This mode is used in conjunction with the **mark** and **fmark** instructions, and the breakpoint registers.

The **mark** and **fmark** instructions allow breakpoint-trace events to be generated at specific points in the instruction stream. When the breakpoint-trace mode is enabled, breakpoint-trace event are generated by every **mark** instruction. The **fmark** generates a breakpoint-trace event regardless of whether the breakpoint-trace mode is enabled or not.

The processor has two, one-word breakpoint registers, designated as breakpoint 0 and breakpoint 1. Using the set-breakpoint-register IAC, one instruction pointer can be loaded into each register. A breakpoint trace is then generated any time an instruction referenced by a breakpoint register is executed. (The BiiN™ Operating System does not provide access to the breakpoint registers.)

11.5 Trace-Fault Handler

A fault handler is a procedure that is called to handle faults. The requirements for fault handlers are given in Section 10.7.

A trace-fault handler has one additional restriction. It must be called with an implicit interdomain call, and the trace-control flag in the domain object entry must be clear. This restriction insures that tracing is turned off when a trace fault is being handled, which is necessary to prevent an endless loop.

11.6 Signaling a Trace Event

To summarize the information presented in the previous sections, a trace event occurs when one of the following conditions occurs:

- An instruction included in a trace-mode group is executed or about to be executed (in the case of a prereturn trace event) and the trace mode for that instruction is enabled.
- An implicit call operation has been executed and the call-trace mode is enabled.
- A **mark** instruction has been executed and the breakpoint-trace mode is enabled.
- An **fmark** instruction has been executed.
- An instruction specified in a breakpoint register is executed and the breakpoint-trace mode is enabled.

When the trace-enable flag is set in the process controls, the trace event generates the following actions:

1. The appropriate trace-event flag is set in the trace controls. If a trace event meets the conditions of more than one of the enabled trace modes, a trace-event flag is set for each trace mode condition that is met.
2. The trace-fault-pending flag is set in the process controls.

Note that the trace-event flag and the trace-fault-pending flag may be set before the instruction that triggered the event is complete. However, the trace event is generated only "between" the execution of instructions.

If the trace-enable flag is clear, and a trace event is detected, the corresponding event flag is set, but the trace-fault-pending flag is not set (and thus, the trace event does not trigger a fault).

11.7 Handling Multiple Trace Events

Multiple trace events are resolved according to the following precedence (from most significant to least significant):

1. Subsystem/supervisor-trace event
2. Breakpoint- (from **mark** or **fmark** instruction, or from a breakpoint register), branch-, call-, or return-trace event
3. Instruction-trace event

If two or more trace events occur at the same time, only the most significant event is necessarily generated. Other events may or may not be generated.

11.8 Trace Handling Action

Trace events may or may not fault, depending on the trace-enable and trace-fault-pending flags, and on other events occuring at the same time, such as an interrupt or a non-trace fault.

The following sections describe how trace events are handled for various situations.

11.8.1 Normal Handling of Trace Events

Before each instruction is executed, the trace-fault pending flag is checked. If the flag is clear, the trace-enable flag is checked. If the trace-enable flag is clear, all trace event flags are cleared before executing the next instruction. If the trace-enable flag is set, a trace fault is raised, and the fault is handled as described in Chapter 10.

11.8.2 Prereturn Trace Handling

A prereturn-trace event is handled as described above, unless the event occurs at the same time as a non-trace fault, in which case the non-trace fault is handled first.

On returning from the fault handler for the non-trace fault, the the prereturn-trace flag is checked in register l0. If this flag is set, a prereturn-trace is generated and handled as described above.

11.8.3 Tracing and Interrupt Handlers

Tracing is disabled during an interrupt handlers automatically, by saving the current state of the process controls, and then clearing the trace-enable and trace-fault-pending flags in the current process controls.

Since the process controls are restored upon return from the interrupt handler, a trace fault may be signaled depending on the trace-enable and trace-fault-pending flags in the restored process controls.

11.8.4 Tracing and Fault Handlers

Fault handlers may be invoked with either an implicit local call or an implicit interdomain call. On a local call, the trace-enable and trace-fault-pending flags are neither saved on the call nor restored on the return. The state of these flags on the return is thus dependent on the action of the fault handler.

On a interdomain call, the trace-enable and trace-fault-pending flags are saved, as part of the saved process controls, and restored on the return. So, if these two flags were set prior to calling the fault handler, a trace fault will be signaled on the return from the fault handler.

On a return from an interrupt handler or a fault handler (other than the trace-fault handler), the trace-fault-pending flag is restored. If this flag is set as a result of the handler's **ret** instruction, the detected trace event is lost.

INTERRUPTS 12

This chapter describes the interrupt handling facilities. It also describes how interrupts are signaled.

12.1 Overview of the Interrupt Facilities

An interrupt is a temporary break in the control stream of a process so that the processor can handle another task. Interrupts are generally requested from an external source. The interrupt request either contains a vector number or else points to a vector that tells the processor what task to do while in the interrupted state. When the processor has finished servicing the interrupt, it generally returns to the process that it was last working on when the interrupt occurred, and resumes execution where it left off.

The mechanism for servicing interrupts uses an implicit procedure call to a selected interrupt handling procedure, called an *interrupt handler*.

When an interrupt occurs, the current state of the process is saved. If the interrupt occurs during an instruction that requires many machine cycles, the instruction state is also saved and execution of the instruction is suspended.

A new frame is then created on the interrupt stack and the interrupt handler (selected with the interrupt vector) is called. As part of the implicit call, the processor switches to a pseudo-process.

Upon returning from the interrupt handler, the processor switches back to the process that was running when the interrupt occurred, restores this process to the state it was in when the interrupt occurred, and resumes work on the process.

Interrupts may be prioritized. If an interrupt is signaled that has the same or a lower priority than the process that the processor is currently working on (and the priority of the interrupt is below 31), the interrupt request is saved for service at a later time. Interrupts that are waiting to be serviced are called pending interrupts.

12.2 Software Requirements for Interrupt Handling

The following items must be present in memory to use the interrupt handling facilities:

- Interrupt Table
- Interrupt Handler Routines
- Interrupt Stack
- Interrupt Environment Table

These items are generally established in memory as part of the initialization procedure. Once these items are present in memory and pointers to them have been entered in the appropriate system data structures, interrupts are then handled automatically and independently from software.

The requirements for these items are given in following sections of this chapter.

12.3 Vectors and Priority

Each interrupt vector is 8 bits in length, which allows up to 256 unique vectors to be defined. In practice, vectors 0 through 7 cannot be used, and vectors 244 through 247 and 249 through 251 are reserved and should not be used by software. Vector 248 is reserved for a processor-generated interrupt called a *system-error interrupt.* This interrupt is described in Section 10.6.3.

Each vector has a predefined priority, which is equal to the vector number divided by 8 (discarding any remainder). Thus, at each priority level, there are 8 possible vectors (for example, vectors 8 through 15 have a priority of 1, vectors 16 through 23 a priority of 2, and so on to vectors 246 through 255, which have a priority of 31).

The priority of an interrupt determines whether the interrupt will be serviced immediately. If the interrupt priority is greater than the priority of the current process, the interrupt receives immediate service; if the interrupt priority is equal to or lower than the priority of the current process, the interrupt vector is saved as a pending interrupt so that the interrupt can be serviced after work on the current process is complete.

A priority-31 interrupt is always serviced immediately.

Note that the lowest process priority allowed is 0. If the current process has a 0 priority, a priority-0 interrupt will never be accepted. This is why vectors 0 through 7 cannot be used. In fact, there are no entries provided for these vectors in the interrupt table.

12.4 Interrupt Table

The interrupt table contains instruction pointers to interrupt handlers. This table is located in physical memory and must be aligned on a word boundary. The processor object contains the location of the interrupt table as a physical address.

As shown in Figure 12-1, the interrupt table contains one entry (that is, one pointer) for each allowable vector. The structure of an interrupt-table entry is given at the bottom of Figure 12-1. Each interrupt procedure must begin on a word boundary, so only the 30 most-significant bits of the pointer are given.

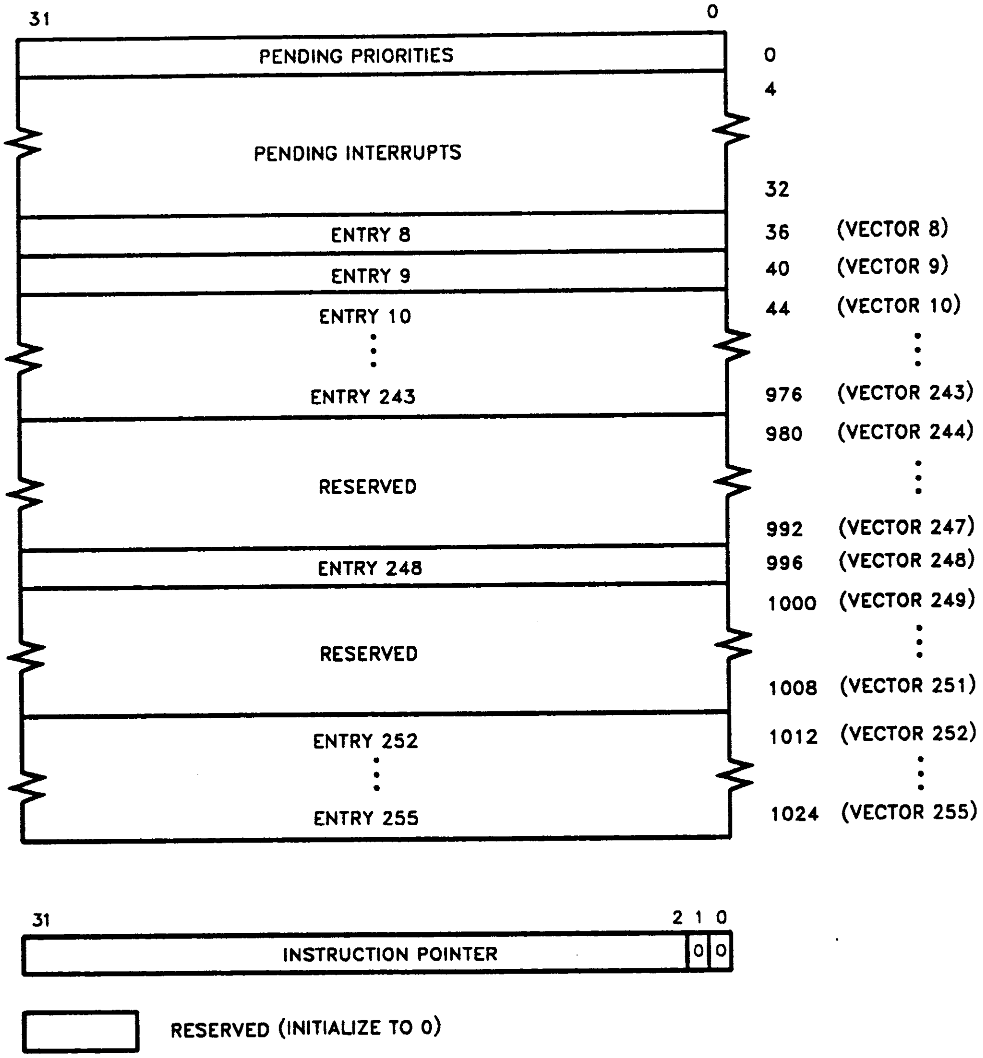


Figure 12-1. Interrupt Table

The instruction pointers point to an address in the current linear address space of the processor (local procedure).

The first 36 bytes of the interrupt table record pending interrupts. This section of the table is divided into two fields: pending priorities (byte-offset 0 through 3) and pending interrupts (byte-offset 4 through 35).

The pending-priorities field contains a 32-bit string in which each bit represents an interrupt priority. The bit number in the string represents the priority number. A pending interrupt is posted in the interrupt table by setting the corresponding bit. For example, if an interrupt with a priority of 10 is posted in the interrupt table, bit 10 is set.

The pending-interrupts field contains a 256-bit string in which each bit represents an interrupt vector. For example, byte-offset 4 is reserved, byte-offset 5 is for vectors 8 through 15, byte-offset 6 is for vectors 16 through 23, and so on. When a pending interrupt is logged, its corresponding bit in the pending-interrupts field is set.

For proper operation, these fields should be cleared during interrupt initialization, and then left undisturbed during normal system operation.

12.5 Interrupt Table Sharing

The interrupt table is located in physical memory so that systems that use multiple processors can share the interrupt table. When one processor receives an interrupt and posts it as a pending interrupt in the interrupt table, another processor can service the interrupt.

12.6 Interrupt Handler Procedures

An interrupt handler is a procedure that is designed to perform a specific action that has been associated with a particular interrupt vector. For example, a typical job for an interrupt handler is to read a character from a keyboard.

12.6.1 Location of Interrupt Handler

An interrupt-handler procedure can be located in either the the current linear address space of the process or within a subsystem. If an interrupt-handler procedure is located in the linear address space, it is generally located in region 3. This makes the procedure available to all processes and all subsystems. Procedures located in the current linear address space are accesssed with local procedure entries in the interrupt table. As stated in the previous section, each procedure must begin on a word boundary.

The interrupt handler procedures can also be located in a subsystem, with access to these procedures provided through intersubsystem calls. Here, a local procedure call is made from the interrupt table to a procedure in the linear address space (as described in the previous paragraph). This procedure then issues a **calld** instruction to the selected interrupt procedure in the interrupt-handler subsystem.

When an interrupt handler makes a subsystem call, the *interrupt environment table* is used. The AD for this table is located in the processor object. Using the *interrupt environment table* protects the stack of the interrupted process, and permits the subsystem protection model to be used during interrupt handling.

12.6.2 Interrupt Handler Restrictions

All interrupts are processed in supervisor mode. The pages that contain interrupt handler routines may have their page rights set for supervisor-only access.

When an interrupt-handler procedure is called, the states of the process controls and arithmetic controls for the interrupted process are saved. However, the interrupt handler shares the other resources of the interrupted process, in particular the global registers and the address space. This sharing of resources imposes two important restrictions on the interrupt handler procedures.

First, the interrupt handler procedures must preserve and restore the state of any needed resources. Local registers need not be saved, because a new local stack frame is already established for the interrupt handler. If the interrupt handler needs to use the global or floating-point registers, however, it should save their contents before using them and restore them before returning from the interrupt.

Second, the interrupt handler should not do anything that would cause the interrupted process to be unbound from the processor and rescheduled, because doing so would leave the processor in an indeterminate state. To avoid rescheduling the process, an interrupt handler should not use the **sendserv**, **receive**, or **wait** instructions. Also, the interrupt handler should not enable timing (set the timing flag in the process controls register), since this can result in an end-of-time-slice event that can also cause the interrupted process to be rescheduled.

The **resumprcs** instruction (resume process) can be used; however, the state of the interrupted process will be lost.

An interrupt-handler procedure can also be called from within a pseudo-process, where there is no process object. One example of this situation is when the processor receives an interrupt while it is servicing another interrupt. Here, execution of the **ldtime** instruction (load process time), **ldglobals** (load-from-process-globals), or the **condrec** instruction (conditional receive) returns an undefined result.

12.7 Interrupt Stack

The interrupt stack is usually located in region 3 of the address space. The location of the interrupt stack is defined by a pointer in the processor object. To avoid raising a fault while processing an interrupt, the interrupt stack must be frozen in physical memory, meaning that the pages that contain the stack must always be valid.

The interrupt stack has the same structure as the local procedure stack described in Section 6.7.

12.8 Signaling Interrupts

The processor can be interrupted in any of the following six ways:

- Signal on its interrupt pins
- Signal on its interrupt pins from an external interrupt controller
- An IAC message from external source
- An IAC message from a program in the processor
- A system-error fault interrupt
- A pending interrupt (described at the end of this chapter)

12.8.1 Interrupts From Interrupt Pins

The processor has four interrupt pins, called INT0, INT1, INT2, and INT3. These pins can be configured in either of the following three ways:

- as four interrupt-signal inputs;
- as two interrupt inputs and two pins for handshaking with an interrupt controller such as the Intel 8259A Programmable Interrupt Controller; or
- as one IAC input and three interrupt inputs.

A 32-bit, interrupt-control register in the processor determines how these pins are used. Each interrupt pin is associated with one 8-bit field in the register, as shown in Figure 12-2.

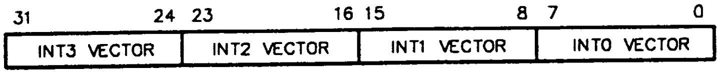


Figure 12-2. Interrupt-Control Register

If the interrupt pins are to be used as four inputs, a different interrupt vector is stored in each of the four fields in the interrupt-control register. Then when an interrupt is signaled on one of the pins, the vector from the pin's associated field is read into the register. For example, if an interrupt is signaled on pin INT0, the vector is derived from bits 0 through 7.

Interrupt vectors in the interrupt register are arranged in descending order from the INT0 field to the INT3 field (that is, the priority of INT0 ≥ INT1 ≥ INT2 ≥ INT3). To insure that interrupts are handled in the proper order, software should follow this convention.

If the INT0 vector field is set to 0, the function of the INT0 pin is changed to IAC, and it is used to signal that an external IAC message has been sent to it. In fact, the INT0 pin must be configured in this manner for external IAC messages to be serviced.

If the INT2 vector field is set to 0, the functions of the INT2 and INT3 pins are changed to INTR and $\overline{\text{INTA}}$, respectively. Here, the INTR pin is used to receive signals from an interrupt controller and the $\overline{\text{INTA}}$ pin is used to send acknowledge signals back to the controller. When the the INTR pin is asserted, an interrupt vector is read from the least-significant 8 bits of the local bus, and an acknowledge signal is sent to the controller through $\overline{\text{INTA}}$. When the INT2 and INT3 pins are configured in this manner, the INT3 vector field is ignored.

The interrupt-control register is memory mapped to physical addresses FF000004$_{16}$ through FF000007$_{16}$. Only the processor can read or write this register using the synchronous load (**synld**) and synchronous move (**synmov**) instructions. External agents on the bus cannot access this register.

The value in the interrupt-control register after the processor is initialized is FF000000$_{16}$.

12.8.2 IAC Interrupts

The processor can also receive an interrupt request by means of the IAC mechanism. (The IAC mechanism is described in detail in Chapter 16.) The interrupt IAC message can be sent to the processor either from an external bus agent, such as an I/O processor or another CPU, or internally as part of the currently running process. The interrupt vector is contained in the interrupt IAC message.

As with any other IAC message, an external interrupt-IAC message is triggered through the INT0 pin, which has been configured as an IAC pin, as described in the previous section. The IAC message is then read to get the interrupt vector.

A program can signal an interrupt through an internal interrupt-IAC message. An internal IAC is sent by means of a synchronous move instruction. When the processor executes a synchronous move to its IAC message space, it signals an IAC message internally. The IAC message is then read as if it was an external IAC.

12.8.3 System-Error Interrupt

Under certain conditions, a system-error interrupt is signaled internally. This interrupt causes a call to interrupt vector 248. The system-error interrupt mechanism, action, and possible handling methods are described in Section 10.6.3.

12.9 Interrupt Handling Actions

The following section describes the actions the taken automatically while handling interrupts. It is not necessary to read this section to use the interrupt mechanism or write an interrupt handler routine. This discussion is provided for those readers who wish to know the details of the interrupt handling mechanism.

12.9.1 Receiving an Interrupt

Whenever the processor receives an interrupt signal, it performs the following action:

1. It temporarily stops work on its current job, whether it is working on a process or another interrupt.
2. It reads the interrupt vector from the interrupt register, the bus, or the IAC message space.
3. It compares the priority of the vector with the priority of the current process (or pseudo-process).
4. If the interrupt priority is higher than that of the process, the processor services the interrupt immediately as described in the next sections.
5. If the interrupt priority is equal to or less than that of the current process, the processor sets the appropriate priority bit and vector bit in pending interrupt record and continues work on the current process.

12.9.2 Servicing an Interrupt

The actions performed to service an interrupt depends on the state the processor is in when it receives the interrupt. The following sections describe the interrupt handling actions for various states of the processor. In all of these cases, it is assumed that the interrupt is a higher priority than the current process and will thus be serviced immediately after the processor receives it. The handling of lower priority interrupts is described later in Section 12.9.8.

12.9.3 Process-Executing State Interrupt

When the processor receives an interrupt while it is in the process-executing state, it performs the following actions to service the interrupt; this procedure is the same regardless of whether the processor is in the user or the supervisor mode when the interrupt occurs:

1. The processor saves the current state of process controls and arithmetic controls in an interrupt record on the stack that the interrupted process is currently using. This stack can be the local-procedure stack or the supervisor stack. (The interrupt record is described in the following section.)
2. If the execution of an instruction was suspended, the processor includes a resumption record for the instruction in the current stack and sets the resume flag in the saved process controls. (Refer to the section in Chapter 16 titled "Instruction Suspension" for a discussion of the criteria for suspending instructions.)

3. The processor switches to the process-interrupted state.
4. In its internally cached process controls, the processor sets the process state to interrupted, the execution mode to supervisor, and the priority to the priority of the interrupt. Changing the mode to supervisor allows the processor to access interrupt handler procedures in protected pages. Setting the process priority to that of the interrupt insures that lower priority interrupts can not interrupt the servicing of the current interrupt.
5. Also in the current process controls, the processor clears the trace-fault-pending, timing, trace-enable, and time-slice flags. Clearing these flags allows the interrupt to be handled without trace faults being raised and without the process timing out.
6. The processor allocates a new frame on the interrupt stack and switches to the interrupt stack.
7. The processor sets the frame return status field (associated with the PFP) to 111_2.
8. The processor performs an implicit call-extended operation (similar to that performed for the **callx** instruction). The address for the procedure that is called is that which is specified in the interrupt table for the specified interrupt vector. This call is to a local procedure.
9. The called procedure may in turn issue a **calld** instruction to an interrupt handler subsystem. The processor handles this call just as it would if the call had been made from within a process. The only difference is that it uses the interrupt environment table instead of the environment table for the current process.

Once the processor has completed the interrupt procedure, it performs the following action on the return:

1. If a subsystem call was made, the processor returns from the subsystem.
2. The processor deallocates the stack frame from the interrupt stack and switches to the local or supervisor stack (whichever one it was using when the process was interrupted).
3. The processor copies the arithmetic controls field from the interrupt record into the arithmetic controls in the processor.
4. The processor copies the process controls field from the interrupt record into the process controls in the processor.
5. If the resume flag of the process controls is set, the processor copies the resumption record from the interrupt record to the resumption record field of the process object for the process being resumed.
6. The processor checks the interrupt table for pending interrupts that are higher then the priority of the process being returned to. If a higher-priority pending interrupt is found, it is handled as if the interrupt occurred at this point.
7. Assuming that there are not pending interrupts to be serviced, the processor switches to the process-executing state and resumes work on the current process.

If the processor is configured to use the high-level process management facilities or multiple processors or both, the processor performs the following additional operations prior to resuming work on the interrupted process.

1. If either the multiprocessor-preempt flag or the check-dispatch-port flag in the processor controls is set, the processor checks the dispatch port and clears the check-dispatch-port flag. Otherwise, it goes to step 4.
2. If the dispatch port contains a process whose priority is higher than that of both the current process and the value in the nonpreempt-limit field in the processor controls, the processor suspends the current process and enqueues it at the front of the queue for its associated

dispatch port. The processor then dispatches the higher priority process, which becomes the current process.

3. If a higher priority process was not found on the dispatch port, the process that was interrupted remains the current process.
4. The processor then begins work on the current process.

The processor executes the interrupt handler procedure from within a pseudo-process. The priority of this pseudo-process is the same as that of the interrupt.

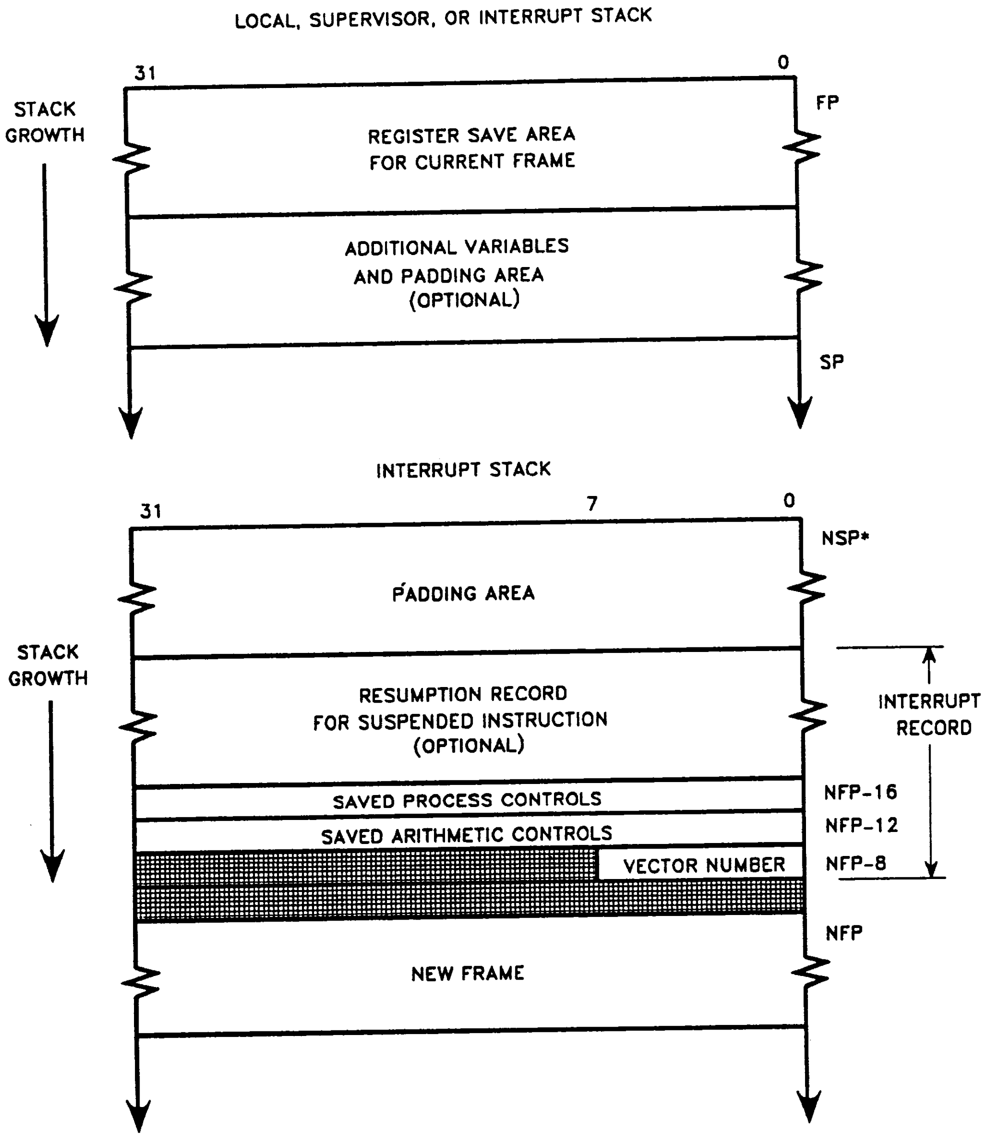
12.9.4 Process-Interrupted State Interrupt

If the processor receives an interrupt while it is servicing an interrupt, and the new interrupt has a higher priority than the current pseudo-process, the pseudo-process is interrupted. Here the processor performs the same action to save the state of the pseudo-process as is described at the beginning of this section. A new pseudo-process is then created in which the interrupt handler for the new interrupt runs.

If the interrupt is received while the processor is executing an interrupt-handler procedure, the interrupt record is saved on the top of the interrupt stack, prior to the new frame that is created for use in servicing the new interrupt.

12.9.5 Interrupt Record

The processor saves the state of the interrupted process in an interrupt record. Figure 12-3 shows the structure of this interrupt record.


LOCAL, SUPERVISOR, OR INTERRUPT STACK
31
0
STACK GROWTH
FP
REGISTER SAVE AREA FOR CURRENT FRAME
ADDITIONAL VARIABLES AND PADDING AREA (OPTIONAL)
SP
INTERRUPT STACK
31
7
0
NSP*
PADDING AREA
STACK GROWTH
RESUMPTION RECORD FOR SUSPENDED INSTRUCTION (OPTIONAL)
INTERRUPT RECORD
SAVED PROCESS CONTROLS
NFP-16
SAVED ARITHMETIC CONTROLS
NFP-12
VECTOR NUMBER
NFP-8
NFP
NEW FRAME

*If the interrupt is serviced while the processor is working on another interrupt procedure, the new stack pointer (NSP) will be the same as the SP.

▨ RESERVED

Figure 12-3. Storing of an Interrupt Record on the Stack

The resumption record within the interrupt record is used to save the state of a suspended instruction. If no instruction is suspended, the resumption record is not created.

12.9.6 Idle or Stopped State Interrupt

The processor can also be interrupted while in either the idle or the stopped state. The processor handles such interrupts in essentially the same way that it handles interrupts that occur while the processor is in the process-executing state, with the following exception. When the processor allocates the new frame on the interrupt stack, it sets the frame return field to 110_2.

This causes the processor to revert to the idle or stopped state when the processor returns from the interrupt-handler procedure.

12.9.7 Idle-Interrupted or Stopped-Interrupted State Interrupt

If the processor receives an interrupt while it is in the idle-interrupted or stopped-interrupted states, it handles the interrupt just as it would if it occurred in the process-interrupted state.

12.9.8 Pending Interrupts

As described earlier, interrupts are evaluated according to their priority. If the interrupt priority is not greater than the current process priority, the interrupt is not serviced, but is instead noted by posting the interrupt in the pending-interrupt section of the interrupt table. Occasionally, these pending-interrupts are checked, and any outstanding higher-priority interrupts are then serviced. This pending interrupt mechanism provides two benefits:

1. The ability to delay the servicing of low priority interrupts (by posting them in the pending interrupt section of the interrupt table) allows a processor to concentrate its processing activity on higher priority tasks.
2. In a system that uses two or more processors, both processors can share the same interrupt table. This interrupt-table sharing allows the processors to share the interrupt handling load.

The following paragraphs describe how pending interrupts are handled.

12.9.8.1 Posting Pending Interrupts

An interrupt can be posted in the pending-interrupt record of the interrupt table in either of the following two ways:

1. The processor receives an interrupt with a priority equal to or lower than that of the process the processor is currently working on. The processor then automatically posts the interrupt in the pending-interrupt record.
2. The kernel can set the desired pending-interrupt and pending-priority bits in the interrupt table.

Using the first method, the processor performs an atomic read/write operation that locks the interrupt table until the posting operation has been completed. Locking the interrupt table prevents other agents on the bus from accessing the interrupt table during this time.

The second method of posting an interrupt is risky, because it does not use this locking technique. (The processor's atomic instructions are not able to perform a locking operation that spans several instructions.) This method will work only if the kernel can insure the following:

- that no external I/O agent will attempt to post a pending interrupt simultaneously with the processor, and
- that an interrupt cannot occur after the pending-priority bit of the pending-interrupt record is set but before the pending-interrupt vector is set.

12.9.8.2 Checking for Pending Interrupts

The interrupt table is automatically checked for pending interrupts at the following times:

- After returning from an interrupt-handler procedure

- While executing a **modpc** instruction, if the instruction causes the process's priority to be lowered.
- After receiving a test-pending-interrupts IAC message.

12.9.8.3 Handling Pending Interrupts

Atomic read/write operations are used for both checking and posting to the interrupt table table. This technique prevents other agents on the bus from accessing the interrupt table until the pending-interrupt check has been completed.

A valid pending interrupt is treated as if the interrupt had just occured.

Should two pending interrupts occur at the same priority, the interrupt with the highest vector number is serviced first.

INTRODUCTION TO PROCESSES, PROCESSORS & SYNCHRONIZATION 13

Chapter 14 describes the interprocess communication facility ("ports").

Chapter 15 describes the management of the process object, and how processes use ports to communicate.

Chapter 16 describes the management of the processor object, and how processors communicate and dispatch processes.

INTERPROCESS COMMUNICATION AND SYNCHRONIZATION 14

The interprocess communication facility provides efficient message passing between processes, and the synchronization, scheduling and dispatching of these processes. Additionally, primitive synchronization mechanisms are provided, like atomic operations and semaphores.

Interprocess communication is facilitated by "ports". This chapter describes the structure and semantics of ports; Chapter 15 describes the role of ports in process scheduling and dispatching.

An operating system (such as the BiiN™ Operating System) may not necessarily provide direct access to the facilities described in this chapter, but instead provide services that use these facilities.

14.1 Interprocess Communication Overview

Interprocess communication is based on ports. A process makes a "request" to a port by sending a message to the port; the message is enqueued until another (or possibly the same) "service" process interrogates the port by attempting to receive a message. When a process receives a message, the message is dequeued from the port and delivered to the process.

Messages are objects which contain their own queuing space. There is no limitation on the number of messages enqueued at a port. Any objects can be used as messages independent of their types.

If a receive is attempted from an empty port, the requesting process is suspended and is said to have "receive-blocked". Receive-blocked process objects form a linked list extending from the port object. There is no limitation on the number of blocked processes that can be blocked and enqueued at a port.

When a message is sent to a port where at least one process (server) is receive-blocked, a process is dequeued and is, in turn, rescheduled (causes a preemption or enqueues at its dispatching port).

There are two enqueuing strategies, depending on the enqueuing mode of the port.

1. **FIFO.** The port has a single linked list, which is strictly FIFO, for both messages or blocked processes. A receive from a FIFO port will obtain the first message in the queue.
2. **Priority.** The port has 32 queues, one for each of 32 priority levels. Messages or blocked processes are linked to the various queues, one for each priority. A receive from a priority port will obtain the first message in the highest priority non-empty queue. The priority ranges from 0 for the lowest priority level to 31 for the highest priority level. Within a priority level, processes are arranged in FIFO order.

14.2 Port Object

A port object has a predefined system type. A dispatch port is required to be "frozen". The structure of a port is shown in Figure 14-1.

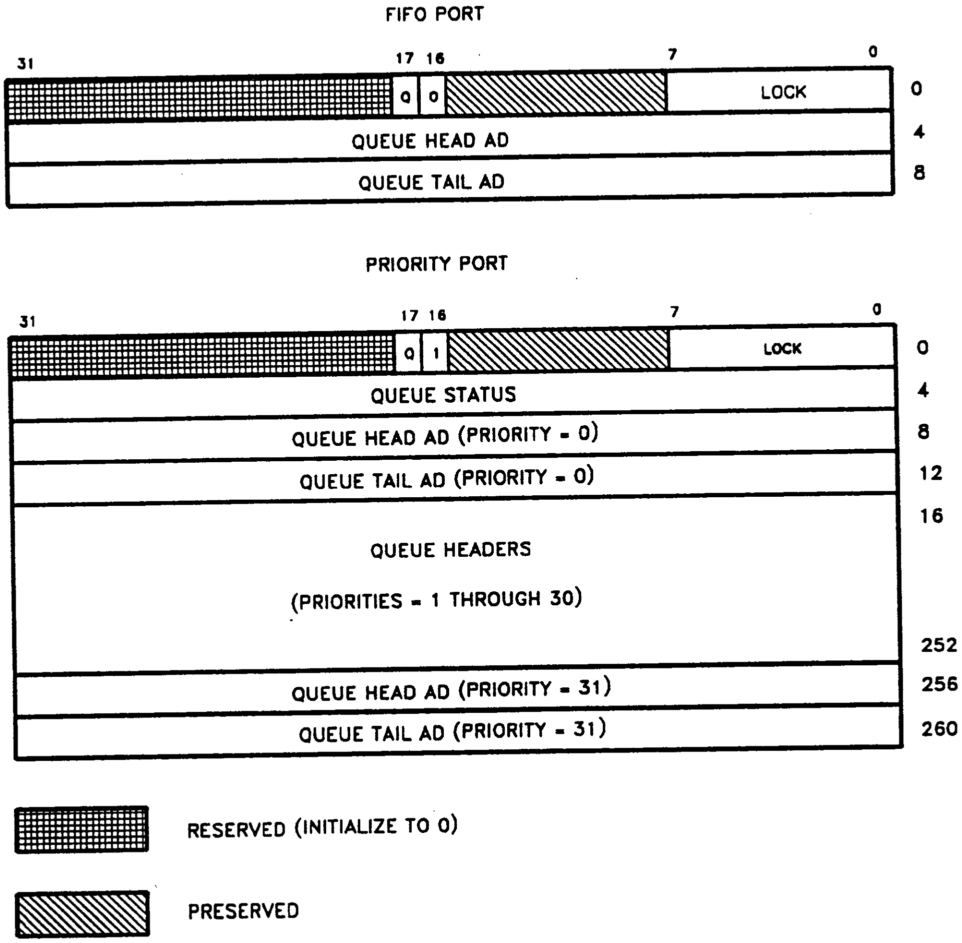

FIFO PORT
31
17 16
7
0
Q
0
LOCK
0
QUEUE HEAD AD
4
QUEUE TAIL AD
8
PRIORITY PORT
31
17 16
7
0
Q
1
LOCK
0
QUEUE STATUS
4
QUEUE HEAD AD (PRIORITY = 0)
8
QUEUE TAIL AD (PRIORITY = 0)
12
16
QUEUE HEADERS
(PRIORITIES = 1 THROUGH 30)
252
QUEUE HEAD AD (PRIORITY = 31)
256
QUEUE TAIL AD (PRIORITY = 31)
260
RESERVED (INITIALIZE TO 0)
PRESERVED

Figure 14-1. Ports

The type rights in an AD for a port object are defined as follows:

Type Rights 1 Uninterpreted.

Type Rights 2 **Send/Receive Rights:** If the bit is 1, a message may be sent to or received from this port.

Type Rights 3 **Service Rights:** If the bit is 1, the current process may be sent to this port using the **sendserv** instruction.

The fields of a FIFO port are defined as follows:

- **Port Lock** (byte 0). This byte is used to synchronize the manipulation of this object. If the least-significant bit of this field is zero, a process or processor can manipulate this object after atomically setting the least-significant bit of this field to one. If the least-significant bit of this field is non-zero, this object is being manipulated.

- **Preserved** (byte 1).
- **Port Status** (bytes 2-3). This field contains the following subfields:

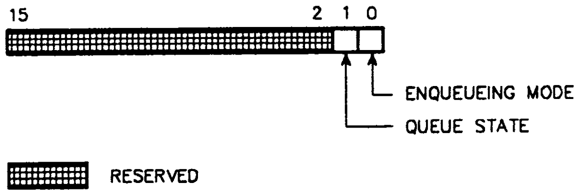


Figure 14-2. Port Status Field

- **Enqueuing Mode** (bit 0).

  0 — **FIFO.** The priority of received blocked processes or messages are ignored and the Queue Head/Tail is always used.

  1 — **Priority.** Received blocked processes or messages are enqueued by priority using the Queue Headers for its priority.

- **Queue State** (bit 1). This bit is 1 when the object(s) in the queue(s) are processes blocked waiting for messages. This bit is 0 when either the port is empty or the object(s) in the queue(s) are messages waiting to be received.

- **Queue Head** (bytes 4-7). This AD references the first process or message enqueued at this port. A data word of value 0 in this field indicates an empty list.
- **Queue Tail** (bytes 8-11). This AD references the last process or message in the queue, if the queue head is not a data word of 0.

The fields of a priority port are defined as follows:

- **Port Lock** (byte 0). The same interpretation as a FIFO port.
- **Preserved** (byte 1).
- **Port Status** (bytes 2-3). The same interpretation as a FIFO port.
- **Queue Status** (bytes 4-7). In this field, each bit position corresponds to a priority level in the port. If the bit corresponding to a particular priority is 1, there are one or more blocked processes or messages at that priority. If the bit is 0, there are no blocked processes or messages at that priority. A queue status of zero indicates an empty port.
- **Queue Headers** (bytes 8-267). For ports operating in Priority Enqueuing mode, there are 32 queue headers. Each Queue Header is structured as follows:
  - **Queue Head** (bytes 0-3). This AD references the first process or message in the queue for this priority. A data word of value 0 indicates an empty list.
  - **Queue Tail** (bytes 4-7). This AD references the last process or message in the queue for this priority, if the queue head is not a data word of value 0.

14.2.1 Port Usage

Ports can either be used as communication ports or as dispatching ports:

- **Communication Port.** Interprocess communication instructions use communication ports. A communication port can be operated in either FIFO or Priority Enqueuing mode.

- **Dispatching port**. Processors use dispatching ports to deposit and obtain schedulable processes. A dispatching port must be a priority port, and the queue state is always zero (to indicates all queues are message queues). Processes are directly enqueued at a dispatching port as messages. When a process is rescheduled at a dispatching port (either as a result of unblocking or via the **schedprcs** instruction), preemption activities may be invoked. A dispatching port is intended to enqueue ready-to-execute processes. See Chapter 15.

The same interprocess communication instructions can be used on both port types.

14.3 Queue Record

A queue record links together processes or messages associated with a port. A queue record can be found in a process control block, and is assumed to be found in all messages specified in an interprocess communication instructions. A queue record is located at offset 0 within an object.

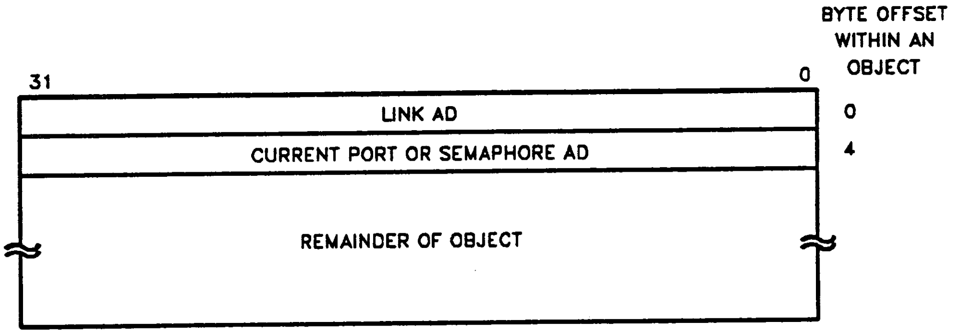


Figure 14-3. Queue Record

The fields of a queue record are defined as follows:

- **Link** (bytes 0-3). This AD links objects in sequence while the object is enqueued at a port or a semaphore; each object's link refers to the next object in the sequence.
- **Current Port or Semaphore** (bytes 4-7). This AD refers to the port or semaphore at which the object is enqueued. The type rights, read and write rights of this AD are cleared. If a data word of value 0, this object is not enqueued at a port or semaphore.

14.4 Message Object

A message object does not have a predefined system type. Any object can be used as a message, and a queue record is assumed to be found in all objects enqueued as messages. Messages of a dispatching port are process objects and must be frozen.

The process executing a **send** instruction must have read/write access to the queue record in the message operand.

A message should be involved in only one port operation at a time.

14.5 Lifetime Checking

Port and message objects are not required to have global lifetime. However, the lifetime of the message must be equal to the lifetime of the port. If they are not equal, a *Lifetime* fault is raised during the send instruction. Lifetime violation is ignored for implicit operations like rescheduling a process.

14.6 Interprocess Communication Instructions

14.6.1 Priorities

All send instructions take a message priority (0..31) as an operand.

The message priority enqueues the message. If a process blocks then it is enqueued at the process priority. When a process is unblocked, it is always sent to the dispatching port at the process priority.

If the port is operating in FIFO mode, the priority of the send operation is ignored.

14.6.2 Send and Receive Instructions

send
receive
condrec

The **send** instruction enqueues a message at the end of the queue of the specified priority at the port, provided that there are no processes waiting at the port. Otherwise, a process is dequeued and the message is bound to the process before the process is rescheduled at its dispatching port or causes a preemption action.

The **receive** instruction attempts to receive a message from a port. If the port has enqueued messages, the AD to the message is stored in the destination of the instruction. If the port has no messages, then the process is suspended and the process is enqueued at the port. When the process is unblocked (a message is sent to the port), the process is sent to its dispatching port.

The **conrecv** instruction also performs a receive from the specified destination port. If the operation is successful, the condition code is set to 010_2, otherwise the condition code is set to 000_2 (instead of suspending and enqueuing the process).

14.6.3 Process Level Port Instructions

sendserv
schedprcs

The **sendserv** instruction suspends the currently executing process and sends itself as a message to the end of the queue at its priority in the specified port. The **schedprcs** instruction enqueues the process the the front of its priority queue at the dispatching port if the process is not preempting. Otherwise, preemption actions are performed. A **send** to a dispatching port will not cause a preemption immediately.

14.7 Atomic Instructions

Atomic instructions provide the ability to read, update, and write a data item in memory atomically (that is, without the possibility of another cooperating agent performing an atomic operation to the same 16-byte block, after the processor has performed the read, but before the processor has performed the write). This capability is essential in any system which supports multiple processors. Note that atomic memory operations are independent of normal memory operations; thus, normal memory operations are allowed between an atomic read and write.

The **atadd** instruction adds the source operand to the target. The read and write of target are done atomically. The initial value of target is stored in the destination (that is, the instruction produces two results).

The **atmod** instruction modifies the target as specified by the source and the mask. The original value of target is stored in the destination. The read and write of target are done atomically. The bitwise-logical-AND of the source and the mask is logically-OR'd with the logical-AND of the value in target and the complement of the mask.

The **atrep** instruction replaces the target word with the source. The initial value of target is stored in the destination.

14.8 Semaphores

An additional type of synchronization is provided by a counting semaphore. This mechanism provides the ability for a process to wait on a semaphore by suspension rather than by spinning on a lock.

A semaphore is the only predefined type of embedded descriptor (Chapter 8). The format of the 3-word data structure that resides in an embedded descriptor is shown in Figure 14-4.

The type rights in an AD referring to a semaphore are defined as follows:

Type Rights 1	Uninterpreted.
Type Rights 2	**Signal/Wait Rights:** If the bit is 1, the signal or wait operation on the semaphore can be performed.
Type Rights 3	Uninterpreted.

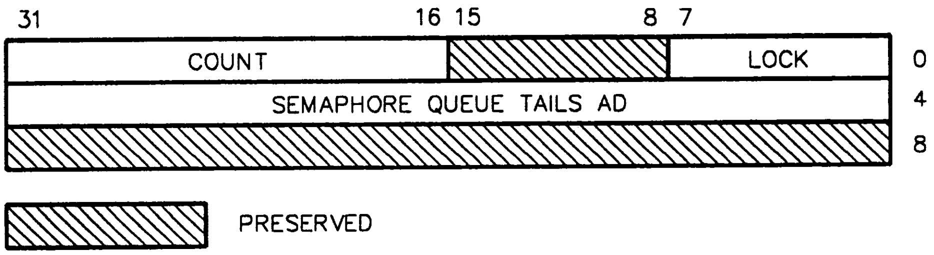


Figure 14-4. Semaphore

The fields of a semaphore are defined as follows:

- **Semaphore Lock** (byte 0). This byte synchronizes accesses to the semaphore count or the semaphore queue. If the least significant bit (LSB) of this byte is zero, the object may be manipulated after atomically setting the bit to one. If the LSB of this byte is 1, another process is manipulating the object.

- **Preserved** (byte 1).
- **Semaphore Count** (bytes 2-3). The semaphore count is a 16-bit short ordinal. If the semaphore queue tail AD is not a data word of value 0, the semaphore count is not interpreted. If the semaphore queue tail AD is a data word of value 0, a non-zero sempahore count indicates the number of waits that can be performed before process blocking will occur, and a zero semaphore count indicates the next **wait** instruction will cause the process to block. This field is initialized to 1 for a binary semaphore.
- **Semaphore Queue Tail** (bytes 4-7). This AD refers to the last process enqueued at this semaphore. A data word of value 0 indicates an empty list. Blocked processes form a circular linked list in decreasing priority order and FIFO within the same priority level.

14.8.1 Semaphore Instructions

wait
condwait
signal

A **wait** instruction decrements the semaphore count by one if the count is non-zero and the queue tail is a data eord of value 0. Otherwise, the process is enqueued on the semaphore queue. The IP of a process blocked on wait points to the **wait** instruction. A **condwait** instruction is similar to the **wait** instruction except that the process will not block. When process blocking would have occurred, the operation is terminated without modifying the semaphore count and sets the conditional code accordingly. A **signal** instruction increments the semaphore count by one if the queue tail is a data word of value 0. Otherwise, a process on the semaphore queue is dequeued and rescheduled.

Processes are enqueued at semaphores in decreasing priority order and FIFO within a priority level. The semaphore references the last process in the queue. The first process in the queue can be located using the link field of the last process in the queue. The link field in the queue record is used to form a linked list. The current port or semaphore field in the queue record refers to the semaphore at which the process is enqueued.

PROCESS MANAGEMENT 15

A process is a single thread of execution that allocates and controls processor resources. This control is achieved via the process control block (PCB). This chapter discusses the process control block and its management.

The process control block is a data structure that is required to be located at the beginning of an object typed as a process object. Thus the terms process control block and process object are often used interchangeably.

15.1 Process Management Overview

Low-level process management is acheived through a set of predefined mechanisms. These mechanisms are responsible for processor resource allocation and operate according to the conventions described here.

There are two major aspects of process management: dispatching and scheduling. "Dispatching" is the activity of assigning a process to a processor. "Scheduling" is the activity of maintaining a list of processes that are awaiting dispatch. Dispatching attempts to deploy processor resources rapidly, while scheduling attempts to allocate those resources to a set of executable processes.

Dispatching and scheduling are based on and supported by the ports described in Chapter 14. Scheduling is equivalent to sending a process control block to a dispatch port, where it is enqueued according to the priority value indicated by the process. Dispatching is equivalent to a processor receiving a process from a dispatch port and resuming execution of that process.

Processors may schedule processes either through predefined mechanisms, or by software-controlled scheduling and dispatching.

15.1.1 Processor Interconnection Architecture

The process-management mechanism is designed to support multiple-processor systems (for example, to have multiple processors share a single dispatch mix).

15.1.2 Process States

A process can be in one of the following states. The term "bound" to a processor means that the state of the process is partly or wholely contained within a processor. The term "suspended" means that the state of a process is not within a processor and is contained within its PCB.

- **Executing.** The process is bound to a processor and is being executed. At most one process per processor can be in this or the interrupted state.
- **Interrupted (but executing).** The process is bound to a processor and is being executed, but is executing a procedure as a result of an interrupt. The processor cannot be allowed to execute another process until this process returns to the executing state (doing so would

leave the processor in an indefinite state). At most one process per processor can be in this or the executing state.

- **Ready.** The process is enqueued on a dispatch port and is suspended.
- **Blocked.** The process is enqueued on a communication port waiting to receive a message and is suspended, or the process is enqueued on a semaphore waiting to receive a signal and is suspended.

A set of transitions among these states without software involvement is predefined. The states and the events that trigger a transition are shown in Figure 15-1.

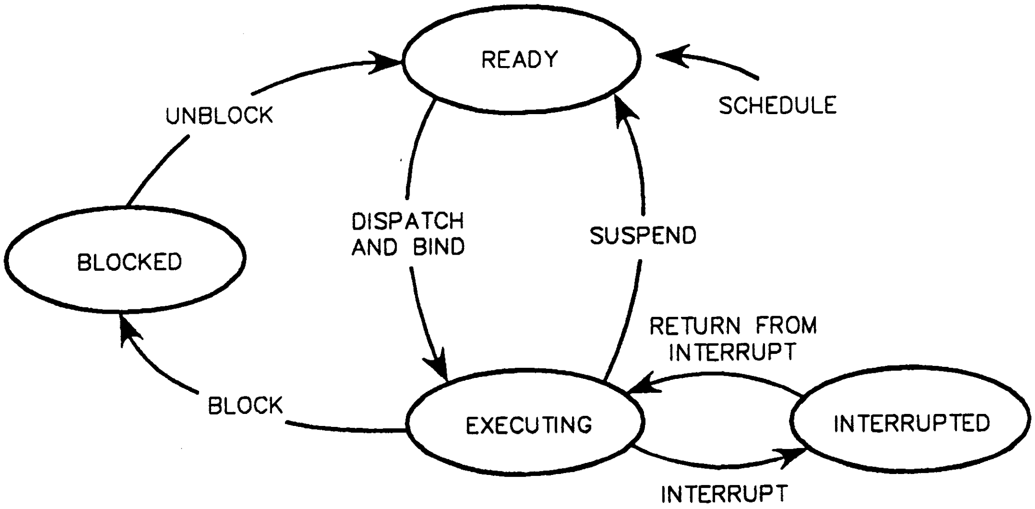


Figure 15-1. Process State Transitions

Events:

1 End of time slice
2 Execution of a **receive** instruction when the port has an empty message queue
3 Execution of a **wait** instruction when the semaphore has no signals
4 Execution of a **sendserv** instruction to a port without a blocked process
5 Execution of a **sendserv** instruction to a port with a blocked process
6 Execution of a **send** instruction to a port with a blocked process
7 Execution of a **signal** instruction to a port with a blocked process
8 Dispatch action of idling processor
9 Reserved
10 Interrupt (see Chapter 16)
11 Execution of a "return-from-interrupt" action (a possible side-effect of executing a return instruction), in some circumstances (see Chapter 16)

Note that some of the events can cause multiple state transitions, and one, event 5, can cause transitions in four processes. To understand this, one needs to think about uniprocessor and multiprocessor systems. Not all of the transitions can occur in uniprocessor systems, where there is at most one executing process. Event 5 can cause four transitions in the following way. Assume process A is executing on processor 1, process B is executing on processor 2, process C is blocked on a port, and process D is ready on the dispatch port. If A executes a **sendserv** to the port, A enters the ready state, C preempts B on processor 2, entering the executing state and moving B to the ready state, and processor 1 grabs process D from the dispatch port, moving it to the executing state.

15.2 Process State Actions

The processor controls transitions of processes from one state to another. The actions performed for these transitions are defined below.

15.2.1 Dispatch Action

A dispatch action is that of a processor examining a dispatch port for a ready process. The steps are:

1. Lock the dispatch port referenced in the processor control block.
2. Find the highest-priority nonempty queue in the dispatch port. If there is none, unlock the dispatch port and examine it at a predefined interval. When it is seen to be non-empty, resume with step 1.
3. Dequeue the first process.
4. Unlock the dispatch port.
5. Lock the process control block.
6. Perform the process-bind action.

15.2.2 Process Bind Action

This action consists of the following steps:

1. Fetch some parts of the PCB. The parts that are fetched and held within the processor during execution are predefined.
2. If the event-fault-request flags are set, clear them and generate an *Event Notice* fault instead of the following steps.
3. Begin instruction execution.

15.2.3 Process Suspension Action

The steps taken for this action are the following:

1. Wait for any uncompleted memory operations and/or instructions to finish.
2. Perform the timing actions defined in Section 15.6.
3. Write any cached local registers back to their associated stack frames. Write the current cached state of the process back to the PCB and, if one exists, the environment table. The state that is not accurate in the PCB prior to this is predefined. The region ADs that are written back have no defined type or rep rights.

15.2.4 End-of-Time-Slice Action

The steps taken for this action are as follows:

1. If the time-slice-reschedule flag is set, do the following:
   a. Perform the process-suspension action.
   b. Unlock the PCB.
   c. Lock the dispatch port referenced by the PCB.
   d. Enqueue the process at the end of the queue (in the dispatch port) whose priority is that of the process.
   e. Unlock the dispatch port.
   f. Perform the process-dispatch action.
2. If the time-slice-reschedule flag is clear, do the following:
   a. Generate a *Time Slice* fault.

15.2.5 Block Action

This action occurs when an executing process enters a blocked state (see Chapter 14). The steps of this action are:

1. Perform the process-suspension action.
2. Perform the action defined for the blocking operation (defined in Chapter 14).
3. Unlock the PCB.
4. Perform the dispatch action.

15.2.6 Unblock Action

This occurs as a result of an operation that causes a blocked process to become unblocked. The steps are:

1. Lock the dispatch port referenced by the PCB.
2. Enqueue the process at the front of the queue (in the dispatch port) for the priority of the process.
3. Unlock the dispatch port.
4. If the process's preempt flag is set, perform the preemption action.

15.2.7 Preemption Action

Processes whose preempt flag is set are considered preempting processes. When such a process goes from the blocked to ready state, it requires an immediate dispatch action.

The action defined here applies only when multiprocessor-preempt flag in the processor controls is clear.

The steps in this action are:

1. If the current process is in the interrupted state, set the check-dispatch-port flag in the processor controls and skip the following steps.

2. If the priority of the preempting process is not greater than that of the current process, skip the following steps.
3. Clear flag check-dispatch-port.
4. Perform the process-suspension action on the current process.
5. Unlock the current PCB.
6. Lock the dispatch port referenced by the current PCB.
7. Enqueue the current process at the front of the queue (in the dispatch port) for the priority of the current process.
8. Unlock the dispatch port.
9. Perform the dispatch action.

One use of the preempt flag is to set the flag in all processes above a predetermined priority level, so that those processes may preempt lower-priority processes immediately.

15.3 Process Object

The process as a unit of scheduleable work is represented by a process control block The process control block (PCB, also known as the "process object") specifies an execution environment, records the execution status of its program, and maintains information about system resources allocated to the process.

The process object has a predefined system type. The PCB must be frozen when in the executing and interrupted states, when enqueued on the dispatch port, and when enqueued on a port or semaphore when a valid addressing path exists to the latter. It must be mapped as a simple object.

When in the executing or interrupted state, some parts of the PCB may be held internally (that is, the memory image may not be accurate, and changing the memory image does not necessarily have any effect on the process). Any part of the PCB may be held internally except for the lock and process-notice fields.

The type rights in an access descriptor that references a PCB are interpreted as follows:

Type Rights 1	undefined.
Type Rights 2	undefined.
Type Rights 3	**Control Right:** If the flag is 1, certain process-related instructions may be executed (see Chapter 18).

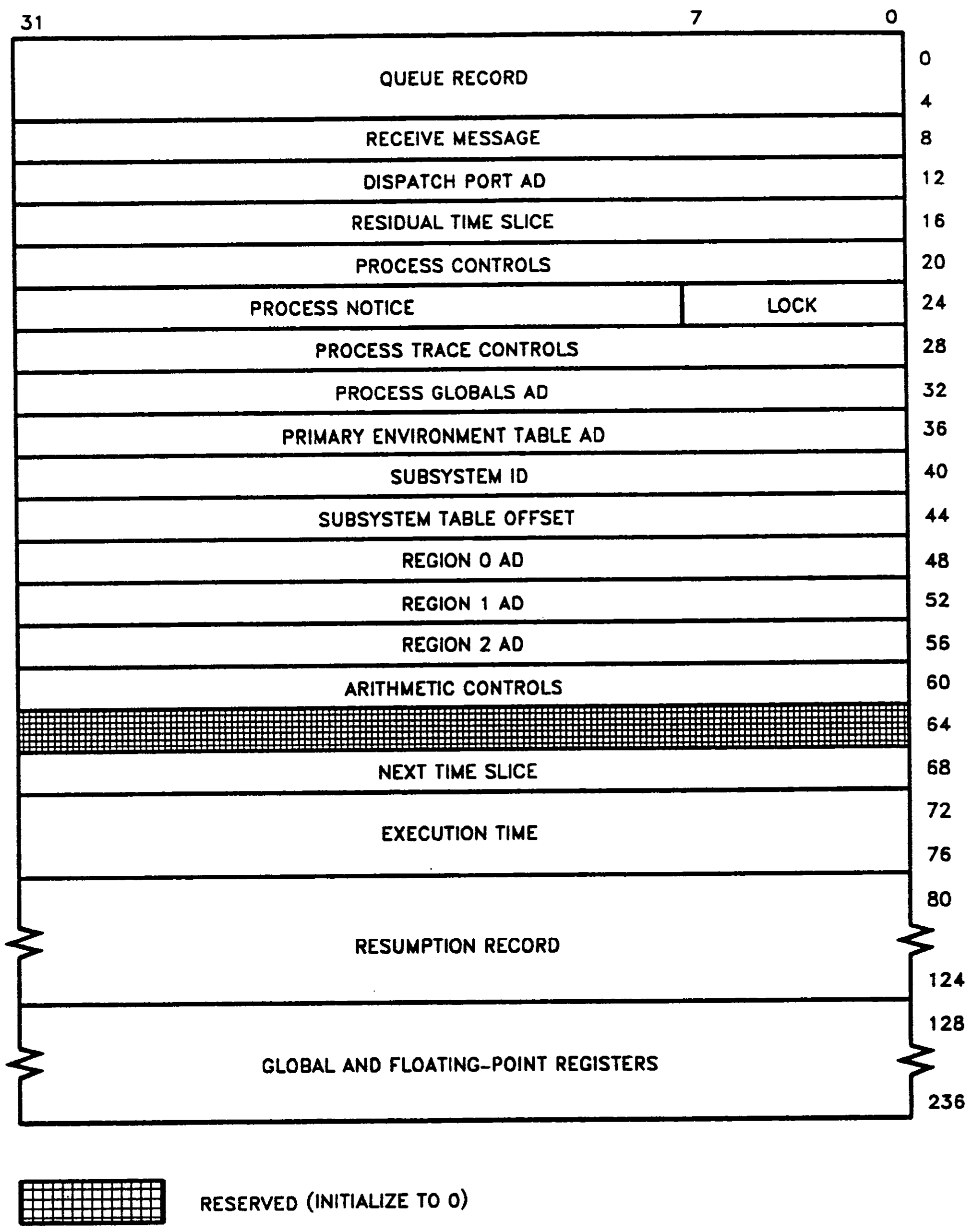


Figure 15-2. Process Control Block

The predefined fields in the PCB are described below.

- **Queue Record.** See Chapter 14. This is used when the process is in the blocked state to link the process on a port or semaphore. It is also used to link the process as a message on a port.
- **Received Message.** When a process leaves the blocked state from a port, the message obtained from the port is placed here (Chapter 14). When the suspended receive instruction resumes, it obtains its result from this field.
- **Dispatch Port AD.** This AD references the dispatch port on which the process is enqueued when it attains the ready state.

- **Residual Time Slice.** An ordinal that specifies the time that the process is allowed to execute before an end-of-time-slice event occurs. (See Section 15.6).
- **Process Controls.**

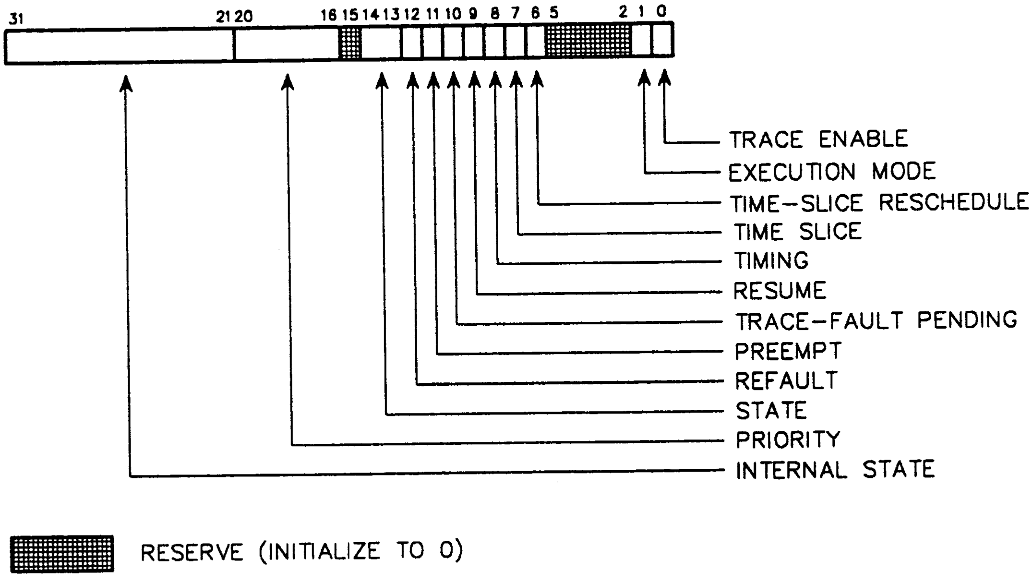


Figure 15-3. Process Controls

- **Internal State.** Reserved. This field should be cleared when the process is created, and not touched after that.
- **Priority.** The priority of the process, a value in the range 0..31 (31 being highest priority). When the process is in the executing state, this becomes the priority of the processor.
- **State.** The state of the process.

00	executing or ready or blocked
01	interrupted (executing)
10	reserved
11	reserved

- **Refault.** This flag is associated with fault override conditions and returns from fault handlers (see Section 10.6.2).
- **Preempt.** If this flag is set, the process is eligible to preempt another process when the process becomes unblocked.
- **Trace Fault Pending.** If this flag is set, a trace fault is generated prior to executing the next instruction, and the flag is cleared. See Chapter 10.
- **Resume.** If this flag is set when the process enters the executing state, or after a return from a fault or interrupt handler, this flag is cleared and the resumption-record field is fetched and used to resume execution of an suspended instruction.
- **Timing.** If set, the execution-timer of the process is operable. If clear, the timer is suspended. (See Section 15.6.)

  - **Time-Slice.** If set, an end-of-time-slice event is permitted (see Section 15.6). If clear, this event will not occur.
  - **Time Slice Reschedule.** If set, the process will be sent to the dispatch port when an end-of-time-slice event occurs. If clear, an end-of-time-slice event will generate a *Time Slice* fault.
  - **Execution Mode.** If set, the process is in supervisor mode. If clear, the process is in user mode.
  - **Trace Enable.** If set, trace modes are enabled, and a detected trace event causes a *Trace* fault (see Chapter 10).

- **Lock.** If bit 0 is 1, the PCB is locked.
- **Process Notice.**

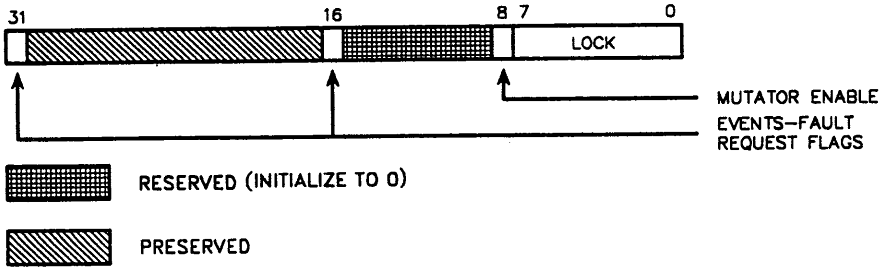


Figure 15-4. Process Notice

  - **Mutator Enable.** If set when the process enters the executing state, gray-bit marking is performed when an access descriptor is copied (see Chapter 8).
  - **Event Fault Request.** This is denoted by two flags. If both are set when the process enters the executing state, the whole field is cleared (bits 16-31), and an *Event Notice* fault is generated.

- **Process Trace Controls.** This word contains trace control and status information. It is described in Chapter 10.
- **Process Globals AD.** This AD references an object that is used by the **ldglobals** instruction.
- **Primary Environment Table AD.** This AD references the environment table used by subsystem call and return operations when the process is not in the interrupted state.
- **Subsystem ID.** This is used by the subsystem call/return operations (see Chapter 7).
- **Subsystem Table Offset.** This is used by the subsystem call/return operations (see Chapter 7).
- **Region 0 AD.** This AD references the object that represents region 0 of the linear address space (see Chapter 6).
- **Region 1 AD.** This AD references the object that represents region 1 of the linear address space (see Chapter 6).
- **Region 2 AD.** This AD references the object that represents region 2 of the linear address space (see Chapter 6).

- **Arithmetic Controls.** This word contains arithmetic control and status information. It is described in Chapter 6.
- **Next Time Slice.** An ordinal that is assigned to field residual-time-slice when the value of the latter is 0.
- **Execution Time.** A long-ordinal. The time this process has spent in the executing state with timing enabled is given by the value of this field minus the value of residual time slice.
- **Resumption Record.** An area used to hold the intermediate state of a suspended instruction when the current instruction is suspended because of an interrupt, preemption or end-of-time-slice, or because of certain types of faults. When the resume flag is set, the resumption information is taken from this field (see Chapter 16).
- **Global Registers.** The first 64 bytes hold the values of global registers G0-G15 (where G0 is in the first word, and so on). The remaining 48 bytes hold the values of the floating-point registers FP0-FP3 (where, in the 48 bytes, FP0 is in bytes 0-9, FP0 is in bytes 12-21, and so on). The two bytes (e.g., bytes 10-11) between each floating-point register are reserved.

15.4 Event Fault

The event-fault-request flags in the process-notice field and the **Event Notice** fault allow a process to induce a fault in a second process without encountering race conditions in a multiprocessor system (that is, independent of the state of the second process, which may be changing while one is trying to induce the fault).

The event-fault-request flags are tested during a process-bind action and possibly as the result of an IAC operation. If the flags are both set, they are both cleared after being tested (to ensure that the fault does not occur twice).

Since the flags are not tested and cleared as an atomic operation, a race condition can exist if software tries to set one when already set. The existence of two flags allows software to use one as an enabled/disabled flag and the other as the signal.

15.5 Changing the Process Controls

There are several circumstances where the process controls of the current process can be changed explicitly by the program. These circumstances are (1) during a **modpc** instruction, (2) a return-from-interrupt action during a **ret** instruction (Chapter 16), and (3) a return-from-fault action during a **ret** instruction (Chapter 10). The first alters the process controls under a mask; the latter two restore the process controls from a word on the stack.

Such changes to the process controls have predictable results, except in situations where the program alters the new value of the following fields of the process controls. This alteration could occur by the **modpc** instruction, or changing the saved process controls in the stack prior to executing the return operation.

- **State.** Changing the state field may or may not change the actual state, unless the change is from "interrupt" to "executing" during a return.
- **Resume.** Changing this flag can cause a subsequently executed instruction to behave unpredictably.

- **Internal State.** Changing this field can leave the process in an undefined state.
- **Trace Enable.** If the trace-enable flag is changed via the **modpc** instruction, the effects may not take place for up to four instructions.

15.6 Process Timing

A set of timing functions is provided for processes. All times are in units of "ticks". A tick is a predefined unit of time; see Appendix B for the proper value.

Timing is as exact as possible, although in certain cases, variable results are possible, and should be regarded as approximate.

To define the timing functions, the following notation will be used.

NTS	Next time slice (in PCB)
RTS	Residual time slice (in PCB)
ET	Execution time (in PCB)
Tflag	Timing flag (in PCB)
TSflag	Time-slice flag (in PCB)
TSRflag	Time-slice-reschedule flag (in PCB)

When timing is enabled (Tflag = 1), the following occurs every tick:

```
if RTS /= 0 then RTS := RTS - 1;
```

The following occurs whenever RTS is 0 and Tflag = 1:

```
RTS := NTS;
ET  := ET + NTS;
if TSflag then
  if TSRflag then
    perform process-suspension action;
    send process to dispatch port;
  else
    raise TimeSlice fault;
  end if;
end if;
```

To prevent endless loops in calling a time-slice fault handler or rescheduling the process, a minimum "safe" value for NTS is predefined. See Appendix B for the value.

The rationale and usage of the timing information in the process is explained below.

NTS (next time slice) is the time a process is allowed to remain in the executing state before an end-of-time-slice event occurs. It is set by software and not changed by the processor.

RTS (residual time slice) represents the actual time counter used by the processor. RTS is used to hold the amount of time remaining in the time slice. It is used by the processor and not normally altered by software. For a new process, software should set RTS equal to NTS.

ET (execution time) is the elapsed execution time of the process. Since it is updated at the end of a time slice, one should subtract the value of RTS from it. For a new process, software should set ET equal to NTS.

Tflag (timing) can be used to stop the execution timer (C).

TSflag (time slice) can be used to disable the generation of an end-of-time-slice event. If disabled, a process will continue to execute beyond the expiration of its time slice. If TSflag is cleared by a **modpc** instruction, an end-of-time-slice event will not occur during the instruction.

TSRflag (time-slice reschedule) can be used to have the processor automatically schedule the process when an end-of-time-slice event occurs, or to allow a fault handler to execute when the event occurs. The latter allows software to change attributes of the process (for example, NTS, priority) before sending it to the dispatch port. Since TSEflag is set when the event occurs, the process will receive a full time slice (NTS) the next time it executes if the fault handler sends it to the dispatch port.

The **ldtime** instruction allows a process to obtain its elapsed time during execution. The instruction stores the value ET-RTS in its destination.

15.7 Other Process-Oriented Instructions

The **modpc** instruction does a modify operation on the process-controls word of the current process. It uses a mask operand to allow one to set or clear bits selectively. The original value of the process controls is stored in the destination. The process controls can be read without alteration by using a mask of all zeros. If the process priority is lowered, the interrupt table is checked for higher-priority interrupts.

The **ldglobals** instruction returns the value of a word at a specified offset in the object referenced by the process-globals field in the PCB.

The **saveprcs** instruction suspends the current process (Section 15.2.3). The current state of the process, including local register sets and the environment table, if being used, is written to memory. The process is not unlocked, and continues to execute.

The **resumprcs** instruction binds the specified process to a processor (Section 15.2.2). This causes the processor to switch processes. The current process is not suspended; thus, its current state disappears.

The **saveprcs** and **resumprcs** instructions are similar to the "save()" and "resume()" functions in most UNIX™ kernels.

PROCESSOR MANAGEMENT AND INTERRUPTS 16

Processors are controlled with the processor control block (PRCB, also called the "processor object"). This chapter discusses the management of PRCBs.

16.1 Processor Object

Each processor is represented by a data structure called a processor control block (PRCB, also called the "processor object"). A PRCB is identified as such as a result of processor initialization.

Some parts of the PRCB may be held internally. Examining or changing the corresponding memory locations with ordinary memory-access instructions may not necessarily produce the expected results. Changing the internal representation of the PRCB can be performed by certain instructions, by IACs, or by reinitializing the processor.

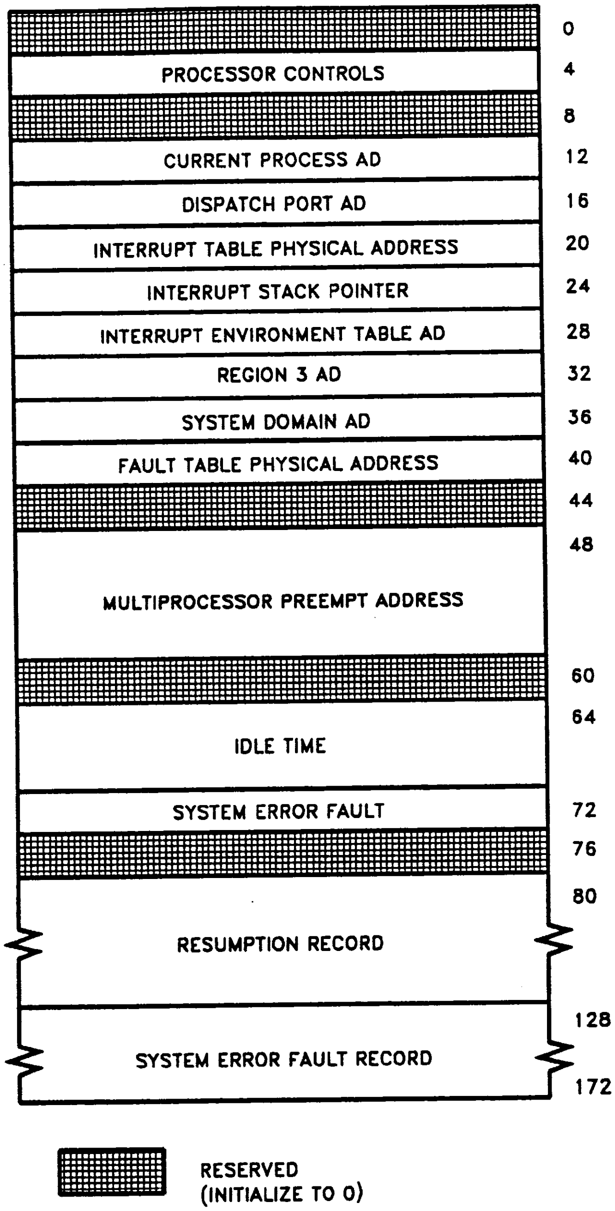


Figure 16-1. Processor Control Block

The fields that constitute a PRCB are defined as follows:

- Processor Controls.

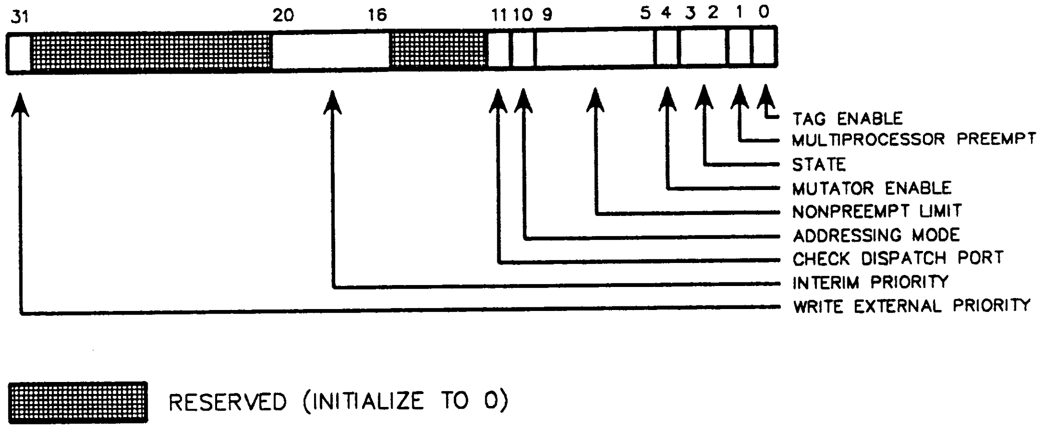


Figure 16-2. Trace Controls

- **Internal State.** This field is reserved, and should be cleared to zero at initialization, and then unmodified later.
- **Check Dispatch Port.** If set, the dispatch port is checked during certain interrupt returns. The flag is set by the processor during certain process-unblock actions.
- **Addressing Mode.** If set, address translation is enabled and addresses are translated as defined in Chapter 8. If clear, the process is in physical-address mode, where linear addresses are interpreted as 32-bit physical addresses (see Section 16.2).
- **Nonpreempt limit.** A priority used during returns from interrupts.
- **Mutator enable.** If set, mutator is enabled (see Chapter 8).
- **State**

  00 The processor is stopped or stopped-interrupted. The processor will not attempt to dispatch processes. In the stopped state, the priority of the processor is zero.

  01 Reserved.

  10 The processor is idle or idle-interrupted. In the idle state, it will examine the dispatch port for an available process at a predefined rate. In the idle state, the priority of the processor is zero.

  11 The processor is process-executing, that is, executing a process whose AD is contained in field current-process-AD.
- **Multiprocessor Preempt.** If set, enables a high-level process preemption function that allows multiple processors to handle preempting processes. This function is only useful in multiprocessor systems, and should be set to 0 in single-processor systems. Refer to Section for details.
- **Tag enable.** If set, tagging is enabled (see Section 16.3).

- **Current Process AD.** This AD refers to the process that is currently bound to the processor. This field is not cached. It has no defined value when the processor is not in the executing state.
- **Dispatch Port AD.** This AD refers to the dispatch port used during the dispatching sequence.

- **Interrupt Table Physical Address.** This physical address points to the base of the interrupt table, which is used to determine the action to perform for an interrupt.
- **Interrupt Stack Pointer.** This linear address points to the base of the interrupt stack.
- **Interrupt Environment Table AD.** This AD refers to the object used for interdomain calls and returns when the process is in the interrupted state.
- **Region 3 AD.** This AD refers to the object used as region 3 of all linear address spaces (see Chapter 6).
- **System Domain AD.** This AD references the system domain, which is used in the **calls** instruction (Chapter 7).
- **Fault Table Physical Address.** This physical address points to the base of the fault table, which specifies how faults are handled (Chapter 10).
- **Idle Time.** A long-ordinal. The time that this processor has spent in the idle state is given by the value of this field. See Section 16.4.
- **System Error Fault.** When a system error occurs, bits 16-23 specify the type and bits 0-7 specify the subtype of the fault directly causing the system error (see Section 10.6.3). This field is not cached.
- **Resumption Record.** This field has the same purpose as the resumption record in the process object. This field is used in place of the resumption record in a process object when the processor is in the idle-interrupted or stop-interrupted state.
- **System Error Fault Record.** This field contains the fault record when a system error occurs. This field is not cached.

16.2 Addressing Modes

When the addressing-mode flag is set, addresses are translated as specified in Chapter 8.

When the addressing-mode flag is clear, linear addresses are interpreted as physical addresses, and such memory references are treated as cacheable (Chapter 9). Implicit and explicit references via ADs behave according to Chapter 8, with the exception of objects representing the current regions of the process, and regions of a subsystem within the control stack; such references may have an undefined effect.

When a virtual-address is produced when in physical mode (for example, by the **cvtadr** instruction), the AD produced is unpredictable and the offset is the physical address of the operand.

16.3 Tag Control

ADs (Access Descriptors) are protected pointers to individual address spaces and have special hardware semantics (that is, they cannot be arbitrarily altered or created). The hardware semantics are controlled by using a tag bit, which is the 33rd bit of a data word and is not directly accessible by user programs. Tag semantics may be disabled for those systems that do not wish to use this feature. This capability is provided by tag-enable flag.

The tag-enable flag (in the processor controls) controls the interpretation of information inside the processor. If tagging is disabled and the execution mode is not supervisor, the behavior is the same as if tagging were enabled. If tagging is disabled and the execution mode is supervisor, the behavior is also the same, except that any value that must be an AD is assumed to be an AD (that is, an implicit tag is supplied).

In a system where tags are not used, processor tag-enable should be clear. This allows processor operations (such as dispatching) to occur, interpreting data as ADs. Any reference to an AD by a process becomes a "privileged" operation, which is prohibited from occurring if the process is not in supervisor mode.

16.4 Idle Timing

The idle-time field in the PRCB is used to count the time spent by the processor in the idle state, and in the idle-interrupted state if timing is enabled. (Idle-interrupted time would only be counted if an interrupt handler enabled timing, which is not advisable.) The count is in terms of ticks (see Section 15.6 for a definition of ticks).

This field, like the others in the PRCB, may be cached. However, in order to make the idle time visible to software, the value is written back to the PRCB in memory at a predefined interval. See Appendix B for details.

Like the process execution time, the processor idle time is only required to be an approximation, and it need not be exactly reproducible. Changing the idle-time field has an unpredictable effect on its future values. The only time the the idle-time field is necessarily fetched from the PRCB in memory is when a PRCB is first associated with the processor.

INSTRUCTION FORMATS AND OPERAND ADDRESSING 17

This chapter defines the formats of the instructions. It also defines the mechanisms for specifying the addresses of operands in memory.

17.1 Instruction Formats

Most instructions are one word long, except for the class of instructions that use an additional word as a 32-bit displacement. All instructions begin on a word boundary.

The formats of the instructions are shown in Figure 17-1. There are four basic formats, named REG, COBR, CTRL, and MEM. The format of each instruction is defined by the opcode field.

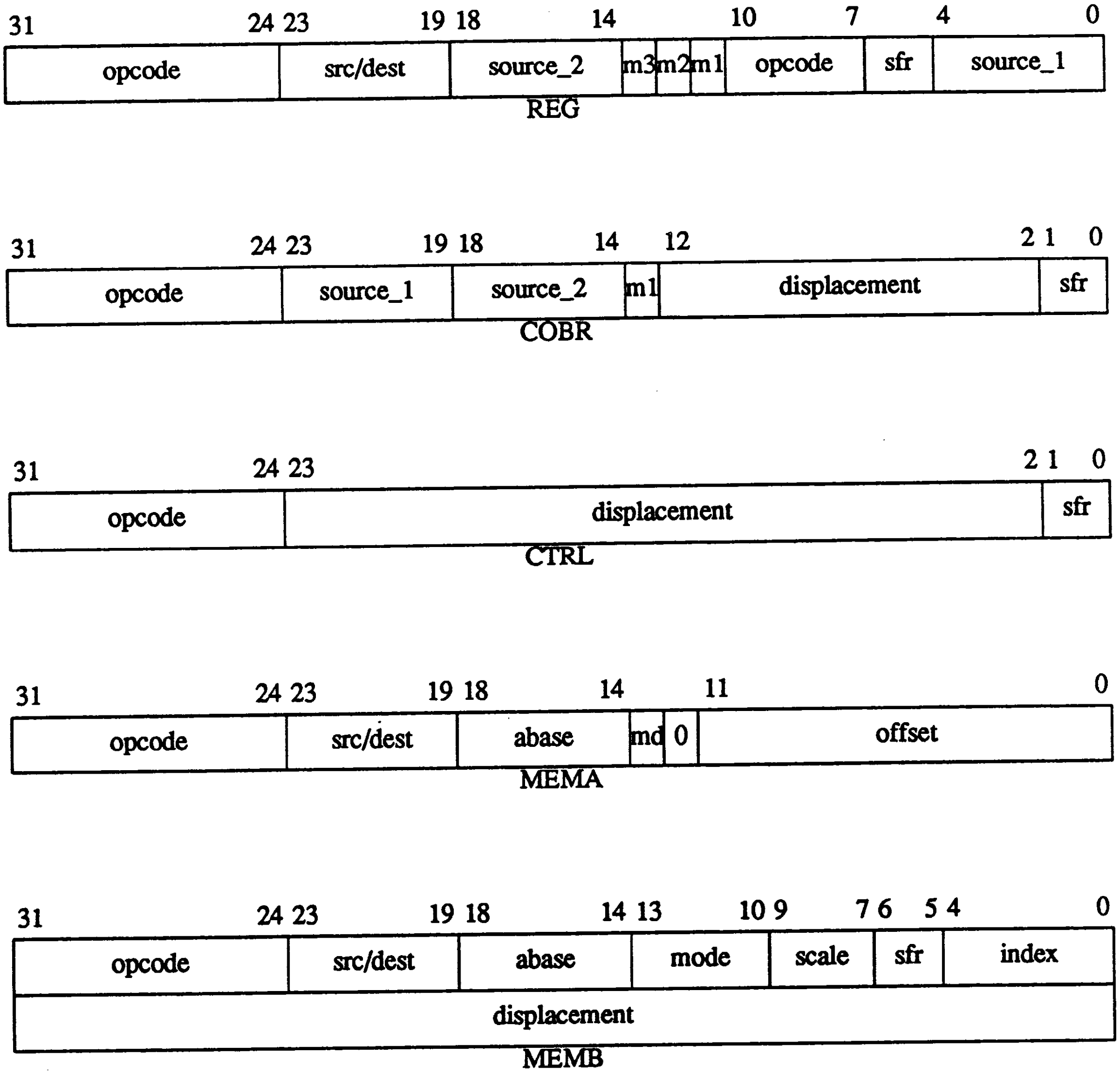


Figure 17-1. Instruction Formats

As shown in Figure 17-1, the MEM format has two "subformats," named MEMA and MEMB. These are distinguished by a bit outside of the opcode; thus an instruction having the MEM format can be encoded with either the MEMA or MEMB format. The MEMB format is unique in that an instruction in this format can be optionally followed by a 32-bit displacement. Thus there are actually three "subformats" of the MEM format.

The following terms appear in the descriptions of the formats:

register A 5-bit register number. Values 00000_2 through 01111_2 denote the local registers (l0 through l15), and values 10000_2 through 11111_2 denote the global registers (g0 through g15). Encodings for floating-point registers are defined in Section 17.2.1.

literal A 5-bit value that is extended to a 32-bit value and used as the operand. When a specific instruction defines an operand as a non-floating-point value, a literal value of the form $rrrrr_2$ defines an integer/ordinal value in the range 0 through 31. When the instruction defines an operand as floating-point, the values specified by literals are defined in Section 17.2.1. When the instruction defines an operand as larger than 32 bits, values specified by literals are zero-extended to the operand size.

value When a specific instruction specifies that an operand is a "value", the operand may be a value in a register or a literal. When the size of an operand is greater than 32 bits and the value is specified as a register, successive registers hold the value as described in Chapter 6 (except for the floating-point registers); when the value is specified as a literal, the literal value is zero-extended to the size of the operand (except for floating-point literals).

address When an instruction specifies that an operand is an "address", the address may be either a linear or virtual address. Linear addresses are contained in the specified register. Virtual addresses are contained in a consecutive pair of registers: the specified register contains an offset, and the next higher register contains the AD of the object into which the offset selects an address.

The fields in Figure 17-1 are defined as follows.

opcode The opcode of the instruction. Opcode encodings are defined in Chapter 18.

source_1 An input to the instruction. Specifies a value or address. In the COBR format, this field specifies a register in which a result is stored.

source_2 An input to the instruction. Specifies a value or address.

src/dest Depending on the specific instruction, this can be either an input value or address, the register where the result is stored, or both.

m1 If the instruction defines source_1 as a value, denotes whether source_1 is a register (0) or literal (1). If source_1 is a floating-point value, see Section 17.2.1. If the instruction defines source_1 as an address, denotes whether source_1 is a register containing a linear address (0) or a register pair containing a virtual address (1).

m2 Same as m1, but applies to source_2.

m3 If the instruction defines src/dest as an input value, m3 has the same meaning as m1, but applying to src/dest. When src/dest is used as a destination, m3 is ignored.

abase A register. The register's value is used to compute a memory address.

index A register. The register's value is used to compute a memory address.

displacement A signed two's-complement number.

offset An unsigned positive number.

md Specifies how a memory address for an operand is computed.

mode Specifies how a memory address for an operand is computed, and specifies whether the instruction contains a second word to be used as a displacement.

scale Specifies how a register's contents are multiplied for certain addressing modes (such as indexing).

sfr An unused field for address-set extensions.

Unused fields are ignored.

17.2 REG-Format Instructions

The majority of the instructions have the REG format. Such instructions can have zero to three operands. The usage of the operands is described in Chapter 18 for each instruction. REG-format opcodes are listed in Chapter 18 as three hexadecimal digits, where the rightmost digit denotes the part of the opcode in bits 7-10.

17.2.1 Floating-Point Registers and Literals

When an operand's type is floating-point and the corresponding m bit (m1/m2/m3) is 0, the meaning is the same as that defined earlier; one (or a pair) of the 32 global or local registers is addressed.

When the corresponding m bit is 1, the operand is defined in the following way. Source_1, source_2, and src/dest field encodings 00000_2 through 00011_2 denote floating-point registers fp0 through fp3, respectively. Encoding 10000_2 denotes the literal +0.0. Encoding 10110_2 denotes the literal +1.0.

All other encodings when the m bit is 1 are reserved. Use of a reserved encoding as a source produces an undefined value or an *Invalid Operation* fault. Use of a reserved encoding or literal as a destination either causes no result to be stored, causes one of the floating-point registers to be altered, or causes the fault.

17.3 COBR-Format Instructions

Instructions having the COBR format are primarily the compare-and-branch instructions. They have two source operands and a displacement. Source_1 can be a literal or register; source_2 is a register. If the branch is taken, the displacement is interpreted as a signed value, multiplied by four, and added to the IP.

In some instructions, source_1 specifies a destination register, in which case m1 is ignored.

17.4 CTRL-Format Instructions

Instructions having the CTRL format are those that have no operands and specify a memory address of an instruction (for example, branches). The displacement is interpreted identically to the COBR format.

17.5 MEM-Format Instructions

Instructions having the MEM format are those that require the computation of a memory address. This category contains load, store, and load-address instructions, as well as some miscellaneous instructions. All address computations produce a 32-bit ordinal result.

For loads, src/dest specifies the destination register (or, for operands bigger than this register, the first of successive registers). For load-address instructions, src/dest specifies the destination register. For stores, src/dest specifies the register (or first of successive registers) whose value is stored in memory. For the miscellaneous instructions, the usage of src/dest depends on the specific instruction.

The abase and index fields specify registers. The md and mode fields define how a memory address is computed. In addition, the opcode defines whether the addresses are computed "linear relative" (L) or "virtual relative" (V). When not explicitly mentioned, an instruction is type L.

When bit 12 of the instruction is 0, the instruction has the MEMA format. Otherwise the instruction has the MEMB format.

17.5.1 MEMA Format

The abase field specifies a register. The usage of the register is specified by the opcode (linear versus virtual) and the md field. If the abase register (and the next higher register) is used as a virtual address, this is denoted as (AD). If the abase register is used as a linear address, this is denoted as (abase).

The memory address is computed as follows:

Instruction Type	md	Computed Address
L	0	offset
L	1	(abase) + offset
V	0	reserved
V	1	(AD) + offset

The load and store instructions having this format can reference operands in the first 4096 bytes of the linear address space, or reference operands up to 4096 bytes beyond a register pointer, or reference operands in the first 4096 bytes of an object.

For a zero offset (such as accessing the value "(abase)"), the MEMB format should be used, as it is faster to compute.

The lea instruction is used with md = 0 to load a 12-bit constant in a register.

17.6 MEMB Format

This category is the same as MEMA, except the options for computing the memory address are different.

The abase and index fields specify registers. Depending on the mode and opcode, the abase register value is used as an offset or a virtual address.

Some addressing modes provide for scaling of the index register. When this occurs, the value in the register is multiplied by a value specified by the scale field. The definition of the scale field is:

Scale	Scale factor (multiplier)
000_2	1
001_2	2
010_2	4
011_2	8
100_2	16

All other values for scale are reserved.

The mode field defines how the memory address is computed, as follows:

Instruction Type	mode	Computed Address
L	0100_2	(abase)
L	0101_2	(IP) + displacement + 8
L	0110_2	reserved
L	0111_2	(abase) + (index)*2**scale
L	1100_2	displacement
L	1101_2	(abase) + displacement
L	1110_2	(index)*2**scale + displacement
L	1111_2	(abase) + (index)*2**scale + displacement
V	0100_2	reserved
V	0101_2	reserved
V	0110_2	(AD) + (index)*2**scale
V	0111_2	reserved
V	1100_2	(AD) + displacement
V	1101_2	reserved
V	1110_2	(AD) + (index)*2**scale + displacement
V	1111_2	reserved

A reserved encoding raises an *Invalid Operation* fault.

For the modes in which a displacement is used, the word following the instruction is the displacement.

(IP) + displacement + 8 denotes that the displacement + 8 is added to the address of the current instruction.

INSTRUCTION REFERENCE 18

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

18.1 Introduction

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

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

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

- Chapter 4 — Summary of the instruction set by group and description of the assembly-language instruction format
- Chapter 17 — Machine-Level Instruction Formats
- Appendix A — Instruction Quick Reference

18.2 Notation

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

18.2.1 Alphabetic Reference

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

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

A box around the alphabetic reference (such as **addr, addrl**) indicates that the instruction or group of instructions are specifc to this processor, and may not necessarily be available in similar processors.

18.2.2 Mnemonic

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

subi Subtract Integer

18.2.3 Format

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

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

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

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

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

reg Global (g0 ... g15) or local (l0 ... l15) register

freg Global (g0 ... g15) or local (l0 ... l15) register, or floating-point (fp0 ... fp3) register, where the registers contain floating-point numbers

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

flit Floating-point literal of value 1.0 or 0.0

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

mem Address defined with the full range of addressing modes

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

addr	Address
efa	Effective address
AD	Access descriptor

18.2.4 Description

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

18.2.5 Action

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

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

In these action statements, the term AC.cc means the condition-code bits in the arithmetic controls. The notation 2#*value*# means that the value enclosed in the "#" signs is in base 2.

Some of the action statements use the following notation:

byte ( )
halfword ( )
word ( )
memory ( )
atomic_read ( )
atomic_write( )

The expression given in parentheses is a memory address (either virtual or linear); however, all arithmetic in the experssion applies only to the linear address, or the offset part of a virtual address. Byte (x) denotes the byte at address x in memory; halfword (x) denotes a 16-bit quanitity at memory address x; and word (x) denotes a 32-bit quantity at memory address x. Memory (x), atomic_read (x), and atomic_write (x) denote a quantity x at a memory address, where the type or size of the quantity is obvious from the context.

On an atomic read operation, it is assumed that external hardware provides a memory lock. The read is not performed until the lock is unlocked and is not completed until the lock is relocked. On an atomic write operation, the read is not performed until the lock is unlocked.

18.2.6 Faults

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

All instructions can signal a group of general faults which are referred to as STANDARD FAULTS. The list of standard faults is as follows:

STANDARD FAULTS
- Trace Instruction
- Virtual Memory Object
- Virtual Memory PTD
- Virtual Memory PTE
- Invalid Opcode
- Unimplemented Operation
- Invalid AD
- Type Mismatch
- Contents
- Process Time Slice
- Invalid Descriptor
- Protection Length
- Protection Page Rights
- Protection Rep Rights
- Protection Type Rights
- Machine Bad Access

Note that the virtual memory and protection faults listed above can occur on instructions that only access registers. Here, they can occur as a result of the memory access to fetch the instruction.

The following additional standard faults can occur on any instruction that accesses memory:

STANDARD FAULTS
- Invalid AD
- Invalid Descriptor
- Protection Rep Rights

The invalid opcode fault is a standard fault for all instrucitons that use the MEM machine-format (load, store, branch extended, call extended, and so on).

Finally, the type mismatch, contents, and protection type rights faults are standard faults for all instructions that perform operations on specific object types.

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

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

OPERATION
- Invalid Opcode
- Invalid Operand

ARITHMETIC
- Integer Overflow
- Arithmetic Zero-Divide

FLOATING-POINT
- Floating Overflow
- Floating Underflow

Floating Invalid-Operation
Floating Zero-Divide
Floating Inexact
Floating Reserved-Encoding

CONSTRAINT
Constraint Range
Invalid AD

VIRTUAL MEMORY
Invalid Object
Invalid Page-Table-Directory-Entry (PTDE)
Invalid Page-Table-Entry (PTE)

PROTECTION
Object Length
Page Rights

MACHINE
Bad Access

STRUCTURAL
Control
Dispatch
IAC

TYPE
Type Mismatch
Contents

PROCESS
Time Slice

DESCRIPTOR
Invalid Descriptor

EVENT
Event Notice

18.2.7 Example

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

18.2.8 Opcode and Instruction Format

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

subi	593	REG

The opcode is given in hexadecimal format.

The machine language format is one of four possible formats: REG, COBR, CTRL, and MEM. Refer to Chapter 17 for more information on the machine-language instruction formats.

18.2.9 See Also

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

18.3 Instructions

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

addc

Mnemonic:

addc	5B0	REG	Add Ordinal With Carry

Format:

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

Description:

Adds the *src2* and *src1*, and bit 1 of the condition code (used here as a carry in), and stores the result in *dst*. If the ordinal addition results in a carry, bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer addition results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Bit 2 of the condition code is always set to 0. Regardless of the result of the addition, the condition code is always updated.

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

This instruction never signals an integer overflow fault.

Action:

```
# Let the condition code be xCx.
dst ← src2 + src1 + C;
AC.cc ← 2#0CV#;
# C is carry from ordinal addition.
# V is 1 if integer addition would have generated an overflow.
```

Faults:

Example:

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

See Also:

subc

addi, addo

Mnemonic:

addi	591	REG	Add Integer
addo	590	REG	Add Ordinal

Format:

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

Description:

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

Action:

dst ← *src2* + *src1*;

Faults:

Integer Overflow

addr, addrl

Mnemonics:

addr	78F	REG	Add Real
addrl	79F	REG	Add Long Real

Format:

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

Description:

Adds *src2* and *src1* and stores the result in *dst.*

For the **addrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	-∞	-∞	-∞	-∞	-∞	INV	NaN
-F	-∞	-F	src2	src2	± F or ± 0	+∞	NaN
-0	-∞	src1	-0	± 0	src1	+∞	NaN
+0	-∞	src1	± 0	+0	src1	+∞	NaN
+F	-∞	± F or ± 0	src2	src2	+F	+∞	NaN
+∞	INV	+∞	+∞	+∞	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

When the sum of two operands with opposite signs is zero, the result is +0, except for the round toward -∞ mode, in which case, the result is -0.

Action:

dst ← *src2* + *src1*;

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation

Floating Inexact

alterbit

Mnemonic:

alterbit	58F	REG	Alter Bit

Format:

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

Description:

Copies the *src* to *dst* with one bit altered. Bit 1 of the condition code is copied into bit *bitpos* of the *dst.*

Action:

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

Faults:

Example:

```
   # mov bit 10 of g3 into bit 24 of g4 and store in g9
chkbit   10, g3      # test bit 10 of g3
alterbit 24, g4, g9  # g9 ← g4, with bit 24 changed
```

amplify

Mnemonic:

amplify	653	REG	Amplify Rights

Format:

amplify	*tdo_ad,*	*rtsmsk,*	*src/dst*
	reg/lit	reg/lit	reg
	AD		AD

Description:

Amplifies selected rights (read, write, and type) of an AD from *src/dst* and stores the amplified AD in *src/dst.* The rights to be amplified are specified by a rights-mask in *rtsmsk,* which has the same format as an AD, except that the local bit, the object index field, and the tag bit are not used. The rights to be amplified are set to 1 in the rights mask. This rights mask is logically ORed with the access rights of the specified AD.

The amplify rights operation is carried out under the control of a type definition object, the AD of which is specified with *tdo_ad.*

The local bit cannot be amplified; the **cread** instruction has to be used.

Action:

```
if src/dst.tag = 0 then
   raise invalid-AD-fault;         # AD to be amplified must be valid
elsif TDO_of(src/dst) = tdo_ad then
   continue;                       # TDO matches
elsif object_type_of(tdo_ad) ≠ TDO then
   raise type-mismatch-fault;  # tdo_ad must be of object type TDO
elsif super TDO bit in tdo_ad object = 1 then
   continue;                       # no check necessary for super TDO
elsif extended bit in tdo_ad object = 1 then
   raise type-mismatch-fault;  # TDO must match
elsif object_type_of(src/dst) ≠ (object type field in tdo_ad object) then
   raise type mismatch fault;  # object type must match
endif

src/dst ← src/dst or (rtsmsk and 2#11111#);
```

Faults:

Invalid AD

Type Mismatch

MEMORY FAULT

See Also:

restrict, cread

and, andnot

Mnemonics:

and	581	REG	And
andnot	582	REG	And Not

Format:

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

Description:

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

Action:

and: *dst* ← *src2* and *src1*;

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

Faults:

See Also:

LOGICAL

atadd

Mnemonic:

atadd	612	REG	Atomic Add

Format:

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

Description:

Adds the *src* to the ordinal in memory specified by the *ataddr*. The initial value from memory is stored in *dst*. The *ataddr* is the linear or virtual address of the the ordinal. The address is automatically aligned to a word boundary.

The read and write of memory are done atomically.

Action:

```
tempa ← ataddr and not 2#11#;   # force word alignment
temp ← atomic_read (tempa);
atomic_write (tempa) ← temp + src;
dst ← temp;
```

Faults:

MEMORY FAULTS

atanr, atanrl

Mnemonics:

atanr	680	REG	Arctangent Real
atanrl	690	REG	Arctangent Long Real

Format:

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

Description:

Computes the arctangent of *src2/src1* and stores the result in *dst*. The result is returned in radians and is in the range of -π to +π, inclusive. If *src1* is the floating-point literal value +1.0, then these instructions return a result in the range of -π/2 to +π/2. The sign of the result is always the sign of *src2*.

For the **atanrl** instruction, if any operand references a general register, two successive registers are used.

These instructions are commonly used as part of an algorithm to convert rectangular coordinates to polar coordinates. They can also be used to implement the FORTRAN intrinsic functions ATAN and ATAN2.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	-3π/4	-π/2	-π/2	-π/2	-π/2	-π/4	NaN
-F	-π	-π to -π/2	-π/2	-π/2	-π/2 to -0	-0	NaN
-0	-π	-π	-π	-0	-0	-0	NaN
+0	+π	+π	+π	+0	+0	+0	NaN
+F	+π	+π to +π/2	+π/2	+π/2	+π/2 to +0	+0	NaN
+∞	+3π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
F Means finite-real number.

Action:

dst ← arctan (*src2/src1*);

Faults:

Floating Reserved Encoding

Floating Underflow

Floating Invalid Operation

Floating Inexact

atmod

Mnemonic:

atmod	610	REG	Atomic Modify

Format:

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

Description:

Replaces the bits of the ordinal in memory specified by *ataddr* by the corresponding bits in *src/dst* under the controls of a bit *mask*. The bits set in the *mask* select the bits to be modified in memory. The initial value from memory is stored in *src/dst*. The *ataddr* is the address of the the ordinal. The address is automatically aligned to a word boundary.

The read and write of memory are done atomically.

Action:

```
tempa ← ataddr and not 2#11#;   # force word alignment
temp ← atomic_read (tempa);
atomic_write (tempa) ← (src/dst and mask)
            or (temp and not(mask));
src/dst ← temp;
```

Faults:

MEMORY FAULTS

atrep

Mnemonic:

atrep	611	REG	Atomic Replace Mixed

Format:

atrep	*ataddr,*	*src,*	*dst*
	reg	reg/lit	reg

Description:

Copies the mixed word (with tag bit) specified by *ataddr* to *dst*. Copies *src* (with tag bit) to the mixed word specified by *ataddr*. The *ataddr* is the address of the the mixed word. The address is automatically aligned to a word boundary.

The read and write of memory are done atomically.

Action:

```
tempa ← src/dst and not 2#11#;  # force word alignment
perform lifetime check of src with memory (tempa);
dst ← atomic_read (tempa);
atomic_write (tempa) ← src;
```

Faults:

Object Lifetime

MEMORY FAULTS

bal, balx

Mnemonic:

bal	0B	CTRL	Branch And Link
balx	85	MEM	Branch And Link Extended

Format:

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

Description:

Stores the address of the next instruction (the instruction following the **bal** or **balx** instruction) and branches to the instruction specified with the *targ* operand.

With the **bal** instruction, the branch target is within -2^{23} and $(2^{23} - 4)$ from the current IP. The address of the next instruction is stored in g14.

With the **balx** instruction, the branch target can be anywhere in the linear address space. The address of the next instruction is stored in *dst*. The *targ* is specified using the full range of addressing modes. (Refer to Section 17.5 for different addressing modes.)

Action:

```
bal:     G14 ← IP + 4;    # destination next IP is always g14
         IP ← IP + targ;  # resume execution at the new IP

balx:    dst ← IP + inst length; # instruction length
                          # is 4 or 8 bytes
         IP ← effective_address(targ);  # resume execution at the new IP
```

Faults:

b, bx

Mnemonic:

b	08	CTRL	Branch
bx	84	MEM	Branch Extended

Format:

b	*targ* disp
bx	*targ* mem

Description:

Branches to the instruction specified by the *targ*.

With the **b** instruction, the branch target is within -2^{23} and $(2^{23} - 4)$ from the current IP.

With the **bx** instruction, the branch target can be anywhere in the linear address space. The *targ* is specified using the full range of addressing modes. (Refer to Section 17.5 for the different addressing modes.)

Action:

```
b:      IP ← IP + targ * 4; # resume execution at the new IP

bx:     IP ← effective_address(targ); # resume execution at the new IP
```

Faults:

bbc, bbs

Mnemonic:

bbc	30	COBR	Check Bit and Branch If Clear
bbs	37	COBR	Check Bit and Branch If Set

Format:

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

Description:

Copies the bit in *src* (designated by *bitpos*) to bit 1 of the condition code. Bits 0 and 2 are set to zeros. If the selected bit is 0, the condition code is set to 000_2, if the selected bit is 1, the condition code is set to 010_2. The processor then performs a conditional branch based on the condition code.

For the **bbc** instruction, if the selected bit in **src** is 0, the processor sets the condition code to 000_2 and branches to *targ*; otherwise, it sets the condition code to 010_2 and goes to the next instruction.

For the **bbs** instruction, if the selected bit is 1, the processor sets the condition code to 010_2 and branches to *targ*; otherwise, it sets the condition code to 000_2 and goes to the next instruction.

The branch target is within -2^{12} and $(2^{12} - 4)$ from the current IP.

The **chkbit** instruction followed by one of the branch-if instructions is equivalent to these check-bit-and-branch instructions. The latter method produces more compact code; however, the former method can be faster because the branch instruction may not have to wait for the result of the comparison.

Action:

bbc:

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

bbs:

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

Faults:

BRANCH IF

Mnemonics:

be	12	CTRL	Branch If Equal
bne	15	CTRL	Branch If Not Equal
bl	14	CTRL	Branch If Less
ble	16	CTRL	Branch If Less Or Equal
bg	11	CTRL	Branch If Greater
bge	13	CTRL	Branch If Greater Or Equal
bo	17	CTRL	Branch If Ordered
bno	10	CTRL	Branch If Unordered

Format:

b* *targ*
disp

Description:

Branches to a new instruction according to the condition code. The branch target is within -2^{23} and $(2^{23} - 4)$ from the current IP.

For all branch-if instructions except the **bno** instruction, the processor branches to the instruction specified by the *targ*, if the logical AND of the condition code and the mask-part of the opcode is not zero. Otherwise, it goes to the next instruction. The mask is in bits 0-2 of the opcode.

For the **bno** instruction, the processor branches to the instruction specified with *targ*, if the condition code is 000_2. Otherwise, it goes to the next instruction.

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

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

Action:

```
if ((mask and AC.cc) ≠ 2#000#) or (mask = AC.cc) then
   IP ← IP + targ * 4;   # resume execution at new IP
end if;
```

Faults:

call

Mnemonic:

call	09	CTRL	Call

Format:

call	*targ* disp

Description:

Calls a new procedure. The processor performs a local call operation as described in Chapter 6 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

The called procedure must be within -2^{23} and $(2^{23} - 4)$ from the current IP.

Action:

```
syncf;    -- wait for all possible faults
temp ← (SP + 63) and not 16#3f#; -- round to next boundary
RIP ← next_IP;    -- IP of the next instruction after the call
if register_set_full then
   save a register_set in memory at its FP;
end if;
allocate as new frame;
-- local register references now refer to new frame
IP ← IP + displacement * 4;
PFP ← FP;
FP ← temp;
SP ← temp + 64;
```

Faults:

MEMORY FAULTS

Call Trace

calld

Mnemonic:

call	661	REG	Call Domain

Format:

calld	*proc_no*,	*dmn_ad*,	---
	reg/lit	reg AD	

Description:

Calls a procedure in a different subsystem (that is, performs a subsystem call). The *dmn_ad* contains the AD of the domain object for the called subsystem. The *proc_no* contains the procedure number of the called procedure.

This operation causes linkage information between the calling and called subsystem and the calling and called procedures to be saved. A new 64-byte stack frame is allocated in the called subsystem. The processor then gets the IP for the called procedure from the domain object of the called subsystem and begins execution of the new procedure.

For more information about this instruction, refer to Chapter 7.

Action:

```
syncf;    -- wait for all possible faults
RIP ← next_IP;    -- IP of the instruction after the call

-- check the input operands
pe_offset ← ((proc_no * 4) + 48) mod 2^32;
if pe_offset < 48 then  -- wraps around 32 bits
   raise invalid-operand fault;
end if;
if object_type_of(dmn_ad) ≠ domain_type then
   raise type-mismatch fault;
end if;

-- read the procedure entry and the first 4 words of the domain object
proc_entry ← read_va(dmn_ad, pe_offset, word);
dmn_base ← read_va(dmn_ad, 0, quad_mixed);
   -- read new_region_0_ad, new_region_1_ad, new_subsystem_id,
   -- supervisor_sp, new_trace_enable

-- handle different procedure type
case proc_entry.entry_type is
   when local =>
      perform a callx operation with
         proc_entry.offset as the target procedure
```

```
when supervisor =>
   if process_controls.supervisor then
      -- use current stack if already in supervisor mode
      perform a callx operation with
         proc_entry.offset as the target procedure, and
         supervisor_sp as the current stack pointer
   else
      -- use the supervisor stack if calling from user mode
      perform a callx operation with
         proc_entry.offset as the target procedure
      if implicit_fault_call then
         null;
      elsif process_controls.trace_enable then
         new_frame_status ← 2#011#
      else
         new_frame_status ← 2#010#
      end if;
   end if;
   -- always change the trace controls
   process_controls.trace_enable ← new_trace_enable;
   process_controls.supervisor ← 1;

   -- check for trace events (the new_trace_enable is in effect)
   if call or supervisor/subsystem trace enabled then
      raise call and/or supervisor/subsystem-trace fault;
   end if;
when subsystem =>
   -- compose the new control stack entry
   new_control_stack_entry.region_0_ad ← current_region_0_ad;
   new_control_stack_entry.region_1_ad ← current_region_1_ad;
   new_control_stack_entry.trace_enable ← process_controls.trace_enable;
   new_control_stack_entry.subsystem_table_offset ←
         current_subsystem_table_offset;
   new_control_stack_entry.calling_domain_ad ← dmn_ad;

   -- separate intra- vs inter-subsystem call
   if subsystem_id = current_subsystem_id or
      subsystem_id = 0 or process_controls.interrupted then

      -- intra-subsystem call
      if implicit_fault_call then
         new_control_stack_entry.return_mode ← 2#100#;
      else
         new_control_stack_entry.return_mode ← 2#000#;
      end if;
      -- push the new control stack entry
      write_va(current_environment_table_ad,
                  current_control_stack_pointer, quad_mixed) ←
                     new_control_stack_entry;

      -- use the current stack
      perform a callx operation with
         proc_entry.offset as the target procedure
      new_frame_status ← 2#100#

   else
```

```
-- inter-subsystem call
if implicit_fault_call then
   new_control_stack_entry.return_mode ← 2#101#;
else
   new_control_stack_entry.return_mode ← 2#001#;
end if;
-- push the new control stack entry
write_va(current_environment_table_ad,
            current_control_stack_pointer, quad_mixed) ←
               new_control_stack_entry;

-- compute hash index
init_subsystem_table_offset ←
      (new_subsystem_id/4) and (subsystem_table_size);
new_subsystem_table_offset ← init_subsystem_table_offset;

-- search subsystem table (backward)
loop
   new_subsystem_entry ← read_va(current_environment_table_ad,
         new_subsystem_table_offset, quad_mixed);
   if new_subsystem_entry.subsystem_id = new_subsystem_id then
      exit;  -- found
   elsif new_subsystem_entry.subsystem_id = 0 then
      -- a zero means the entry is available,
      -- thus the new_subsystem_id cannot be in the other entries
      raise subsystem-not-found fault;
   elsif new_subsystem_table_offset = init_subsystem_table_offset then
      -- search all entries already
      raise subsystem-not-found fault;
   else
      -- performing BACKWARD search
      new_subsystem_table_offset ← new_subsystem_table_offset - 16;
      if new_subsystem_table_offset = 0 then
         new_subsystem_table_offset ← subsystem_table_size;
      end if;
   end if;
end loop;
```

```
      -- new_subsystem_entry contains
      --   topmost_fp
      --   topmost_sp
      --   new_region_2_ad
      --   new_event_fault_mask
      --     (type rights 1 of the subsystem_id in the subsystem table)

      -- save topmost_fp and topmost_sp of current subsystem table entry
      write_va(current_environment_table_ad,
                    current_subsystem_table_offset, long_word) ←
                      (current_fp, current_sp);

      -- update the intersubsystem specific part of the execution enviroment
      current_region_2_ad ← new_region_2_ad;
      current_subsystem_id ← new_subsystem_id;
      current_subsystem_offset ← new_subsystem_offset;
      current_event_fault_mask ← new_event_fault_mask;

      perform a callx operation with
         proc_entry.offset as the target procedure
         topmost_sp as the current stack pointer
         topmost_fp as the current frame pointer
      new_frame_status ← 2#101#
    end if;
    -- update the current environment for all subsystem calls
    current_region_0_ad ← new_region_0_ad;
    current_region_1_ad ← new_region_1_ad;
    process_controls.trace_enable ← new_trace_enable;
    current_control_stack_pointer ← current_control_stack_pointer + 16;

    -- check for control stack overflow
    if current_control_stack_pointer = control_stack_limit then
       raise control-stack-overflow;
    end if;

    -- check for trace events (the new_trace_enable is in effect)
    if call or supervisor/subsystem trace enabled then
       raise call and/or supervisor/subsystem-trace fault;
    end if;
end case;
```

Faults:

The following faults applies to all cases:

	Invalid AD *dmn_ad* is invalid.
Invalid Operand	*proc_no* too large.
Type Mismatch	*dmn_ad* does not point to a domain object.
Call Trace	

The following faults are specific supervisor calls:

	Supervisor/Subsystem Trace

The following faults are specific subsystem calls:

Subsystem Not Found

Control Stack Overflow

Supervisor/Subsystem Trace

See Also:

call, callx, calls

calls

Mnemonic:

calls	660	REG	Call System

Format:

calls	*proc_no*, reg/lit	---,	---

Description:

Calls a procedure in the system domain. The **proc_no** contains the procedure number of the called procedure.

The **calls** performs the same operation as the **calld** except that the system domain is used instead of specifing the domain using *dmn_ad*. The system domain AD can be found in the processor control block.

Action:

See **calld**, replace *dmn_ad* with an AD to the system domain.

Faults:

See **calld**, except that neither Type Mismatch Fault (on the object specified by *dmn_ad*), nor Rep Rights Fault (on the *dmn_ad*) is possible.

callx

Mnemonic:

callx	86	MEM	Call Extended

Format:

callx	*targ* mem

Description:

Calls a new procedure. The processor performs a local call operation as described in Chapter 6 in the section titled "Local Calls." As part of this operation, the processor allocates a new set of local registers and a new stack frame for the called procedure. The processor then goes to the instruction specified with the *targ* argument and begins execution of the new procedure.

This instruction performs the same operation as the **call** instruction except that the target procedure can be anywhere in the linear address space. The *targ* is specified using the full range of addressing modes. (Refer to Section 17.5 for the different addressing modes.)

Action:

```
syncf;    -- wait for all possible faults
new_FP ← SP + 63;
if implicit_fault_call then
   new_FP ← new_FP + size-of-fault-resumption-record;
end if;    -- round to next boundary
new_FP ← new_FP and not 16#3f#;
if implicit_fault_call then
   new_FP ← new_FP + size-of-fault-resumption-record;
   store fault and resumption records relative to new_FP
end if;    -- round to next boundary
RIP  ← next_IP;    -- IP of the instruction after the call
if register_set_full then
   save a register_set in memory at its FP;
end if;
allocate as new frame;
-- local register references now refer to new frame
IP ← effective_address(targ);
PFP ← FP;
if implicit_fault_call then
   PFP ← FP or 2#001#; -- fault procedure
else
   PFP ← FP or 2#000#; -- local procedure
end if;
FP ← new_FP;
SP ← temp + 64;
```

Faults:

MEMORY FAULTS

Call Trace

See Also:

call

chkbit

Mnemonic:

chkbit	5AE	REG	Check Bit

Format:

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

Description:

Copies the bit in *src* designated by *bitpos* to bit 1 of the condition code. Bits 0 and 2 are set to zeros. If the selected bit is 0, the condition code is set to 000_2, if the selected bit is 1, the condition code is set to 010_2.

Action:

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

Faults:

chktag

Mnemonic:

chktag	5A8	REG	Check Tag

Format:

chktag	*src*, reg/lit	---,	---

Description:

Copies the tag bit of the *src* to bit 1 of the condition code. Bits 0 and 2 are set to zeros. If the tag is 1, the condition code is set to 2#010#; if the tag is 0, the condition code is set to 2#100#.

The purpose of this instruction is to check whether or not the word is an AD.

Action:

```
if src.tag = 1 then
   AC.cc ← 2#010#;
else
   AC.cc ← 2#000#;
end if;
```

Faults:

classr, classrl

Mnemonic:

classr	Classify Real
classrl	Classify Long Real

Format:

classr* *src*
freg/flit

Description:

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

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

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

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

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

Refer to Chapter 5 for a discussion of the different real number classifications.

Action:

```
s ← sign_of(src)
if src = 0
    then arithmetic_status ← s000;
elseif src = denormalized
    then arithmetic_status ← s001;
elseif src = normal finite
    then arithmetic_status ← s010;
elseif src = ∞
    then arithmetic_status ← s011;
elseif src = QNaN
    then arithmetic_status ← s100;
elseif src = SNaN
    then arithmetic_status ← s101;
elseif src = reserved operand
    then arithmetic_status ← s110;
end if
```

Faults:

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

None of the floating-point exceptions can be raised.

Example:

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

Opcode:

classr	68F	REG
classrl	69F	REG

clrbit

Mnemonic:

clrbit	58C	REG	Clear Bit

Format:

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

Description:

Copies the *src* to *dst* with one bit cleared. The *bitpos* specifies the bit to be cleared.

Action:

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

Faults:

See Also:

alterbit, chkbit, notbit, setbit

cmpi, cmpo

Mnemonics:

cmpi	5A1	REG	Compare Integer
cmpo	5A0	REG	Compare Ordinal

Format:

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

Description:

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

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

The cmp* instruction followed by one of the branch-if instructions is equivalent to one of the compare-and-branch instructions. The latter method of comparing and branching produces more compact code; however, the former method can be faster because the branch instruction may not have to wait for the result of the comparison.

Action:

```
if src1 < src2 then AC.cc ← 2#100#;
elseif src1 = src2 then AC.cc ← 2#010#;
else AC.cc ← 2#001#;
end if;
```

Faults:

cmpdeci, cmpdeco

Mnemonics:

cmpdeci	5A7	REG	Compare and Decrement Integer
cmpdeco	5A6	REG	Compare and Decrement Ordinal

Format:

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

Description:

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

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

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

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

Action:

```
if src1 < src2 then AC.cc ← 2#100#;
elsif src1 = src2 then AC.cc ← 2#010#;
elsif src1 > src2 then AC.cc ← 2#001#;
end if;
dst ← src2 - 1;  # no overflow, ordinal arithmetic
```

Faults:

cmpinci, cmpinco

Mnemonics:

cmpinci	5A5	REG	Compare and Increment Integer
cmpinco	5A4	REG	Compare and Increment Ordinal

Format:

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

Description:

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

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

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

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

Action:

```
if src1 < src2 then AC.cc ← 2#100#;
elsif src1 = src2 then AC.cc ← 2#010#;
elsif src1 > src2 then AC.cc ← 2#001#;
end if;
dst ← src2 + 1;  # no overflow, ordinal arithmetic
```

Faults:

cmpm

Mnemonic:

icmpm	5AA	REG	Compare Mixed

Format:

cmpm	*src1*, reg	i[src2], reg	---

Description:

Compares *src1* and *src2* for either access equality (point to the same object) or data equality (contain the same ordinal value) depending if both tag bits are both 1s or 0s. The condition code is set accordingly.

If both words have tag bits of 1 and both words have the same object index (bits 6 through 31), they have access equality and the condition code is set to 010_2. If both words have tag bits of 0 and the words are the same, they have data equality and the condition code is also set to 010_2. Otherwise, the condition code is set to 000_2.

Action:

```
if (src1.tag = src2.tag) and
   ((src1.tag = 1 and src1.object_index = src2.object_index)
    or
    (src1.tag = 0 and src1.word = src2.word)) then
   AC.cc ← 2#010#;
else
   AC.cc ← 2#100#;
 end if;
```

Faults:

COMPARE AND BRANCH

Mnemonics:

cmpibe	3A	COBR	Compare Integer And Branch If Equal
cmpibne	3D	COBR	Compare Integer And Branch If Not Equal
cmpibl	3C	COBR	Compare Integer And Branch If Less
cmpible	3E	COBR	Compare Integer And Branch If Less Or Equal
cmpibg	39	COBR	Compare Integer And Branch If Greater
cmpibge	3B	COBR	Compare Integer And Branch If Greater Or Equal
cmpibo	3F	COBR	Compare Integer And Branch If Ordered
cmpibno	38	COBR	Compare Integer And Branch If Unordered
cmpobe	32	COBR	Compare Ordinal And Branch If Equal
cmpobne	35	COBR	Compare Ordinal And Branch If Not Equal
cmpobl	34	COBR	Compare Ordinal And Branch If Less
cmpoble	36	COBR	Compare Ordinal And Branch If Less Or Equal
cmpobg	31	COBR	Compare Ordinal And Branch If Greater
cmpobge	33	COBR	Compare Ordinal And Branch If Greater Or Equal

Format:

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

Description:

Compares the *src2* and *src1* and sets the condition code according to the results of the comparison. If the logical AND of the condition code and the mask-part of the opcode is not zero, the processor branches to the instruction specified with the *targ* operand; otherwise, the processor goes to the next instruction.

The branch target is within -2^{12} and $(2^{12} - 4)$ from the current IP.

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

Instruction	Mask	Branch Condition
cmpibno	000	Never
cmpibg	001	*src1* > *src2*
cmpibe	010	*src1* = *src2*
cmpibge	011	*src1* ≥ *src2*
cmpibl	100	*src1* < *src2*
cmpibne	101	*src1* ≠ *src2*
cmpible	110	*src1* ≤ *src2*
cmpibo	111	Always
cmpobg	001	*src1* > *src2*
cmpobe	010	*src1* = *src2*
cmpobge	011	*src1* ≥ *src2*
cmpobl	100	*src1* < *src2*
cmpobne	101	*src1* ≠ *src2*
cmpoble	110	*src1* ≤ *src2*

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

The **cmp*** instruction followed by one of the branch-if instructions is equivalent to one of the compare-and-branch instructions. The latter method of comparing and branching produces more compact code; however, the former method can be faster because the branch instruction may not have to wait for the result of the comparison.

Action:

```
if src1 < src2 then AC.cc ← 2#100#;
elsif src1 = src2 then AC.cc ← 2#010#;
else AC.cc ← 2#001#;
end if;
if ((mask and AC.cc) ≠ 2#000#) or (mask = AC.cc) then
   # resume execution at the new IP
   IP ← IP + (displacement * 4);
else
   # resume execution at the next IP
   IP ← IP + 4;
end if;
```

Faults:

See Also:

BRANCH IF, cmpi

cmpor, cmporl

Mnemonics:

cmpor	684	REG	Compare Ordered Real
cmporl	694	REG	Compare Ordered Long Real

Format:

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

Description:

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

For the **cmporl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

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

Action:

```
if   src1 < src2 then AC.cc ← 2#100#;
elsif src1 = src2 then AC.cc ← 2#010#;
elsif src1 > src2 then AC.cc ← 2#001#;
else
   AC.cc ← 2#000#; # indicates one number is a NaN
   raise floating invalid operation fault
end if;
```

Faults:

Floating Reserved Encoding

Floating Invalid Operation

cmpr, cmprl

Mnemonics:

cmpr	685	REG	Compare Real
cmprl	695	REG	Compare Long Real

Format:

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

Description:

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

For the **cmprl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

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

Action:

```
if    src1 < src2 then AC.cc ← 2#100#;
elseif src1 = src2 then AC.cc ← 2#010#;
elseif src1 > src2 then AC.cc ← 2#001#;
else
   AC.cc ← 2#000#; # indicates one number is a NaN
end if;
```

Faults:

Floating Reserved Encoding

Floating Invalid Operation

cmpstr

Mnemonic:

cmpstr	603	REG	Compare String

Format:

cmpstr	*src1*, addr	*src2*, addr	*len* reg/lit

Description:

Compares two strings of equal length and sets the condition code according to the result. The *src1* and *src2* specify the addresses of the first byte in each string, and the *len* specifies the string length in bytes, from 0 to $2^{32} - 1$.

The two strings are compared in lexicographical order. The strings are compared byte-by-byte, from low address to high address, according to their ordinal value. If the byte-by-byte comparison shows that the two strings are identical, the condition code is set to 010_2. If the byte from the *src1* string is greater than the byte from the *src2* string, the condition code is set to 001_2. If the byte from the *src1* string is less than the byte from the *src2* string, the condition code is set to 100_2.

Action:

```
AC.cc ← 2#010#;
for i in 0 .. len - 1 loop
   if byte (src1 + i) > byte (src2 + i) then
      AC.cc ← 2#001#;
      Exit;
   elsif byte (src1 + i) < byte (src2 + i) then
      AC.cc ← 2#100#;
      Exit;
   end if;
end loop;
```

Faults:

MEMORY_FAULT

See Also:

fill, movstr, movqstr

concmpi, concmpo

Mnemonics:

concmpi	5A3	REG	Conditional Compare Integer
concmpo	5A2	REG	Conditional Compare Ordinal

Format:

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

Description:

Compares the *src2* and *src1* if bit 2 of the condition code is 0. If the comparison is performed, the condition code is set according to the results of the comparison.

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

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

Action:

```
if (AC.cc and 2#100#) = 0 then
   if src1 ≤ src2 then
      AC.cc ← 2#010;
   else
      AC.cc ← 2#001;
   endif;
endif;
```

Faults:

Example:

```
cmpo g6, g3       # compares g6 and g3 and sets
                  # condition code
concmpo g5, g3    # if condition code is not
                  # 2#1xx#, g5 is compared
                  # with g3
```

condrec

Mnemonic:

condrec 646 REG Conditional Receive

Format:

```
condrec   port_ad,   dst
          reg        reg
          AD         AD
```

Description:

Attempts to receive a message from a port and sets the condition code to indicate whether the message was received successfully or not. The *port_ad* contains the AD of the port.

If the message is received successfully, the AD of the message is stored in the *dst*, the condition code is set to 010_2.

If a message is not available, the *dst* is undefined and the condition code is set to 000_2.

This instruction is similar to the **receive** instruction, except that with the **receive** instruction, the process blocks and is suspended if a message is not available at the port.

Action:

```
if not port_ad.tag then
   raise invalid-AD fault;
elsif not port_ad.type_rights_2 then
   raise type-rights fault;
elsif object_type_of(port_ad) ≠ port_type then
   raise type-mismatch fault;
end if;

-- loop the port
loop
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   if (port_header.lock_byte and 2#1#) = 2#0#) then exit end if;
   -- wait until the semaphore is unlocked
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;
   delay;
end loop;

if port_header.queue_state = process or else
   (port_header.enq_mode = fifo and port_header.queue_head = 0) or
   (port_header.enq_mode = priority and port_header.queue_status = 0) then
   -- port is empty
   -- unlock the port
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   port_header.lock_byte ← port_header.lock_byte and not 2#1#;
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;

   -- set condition code
   AC.cc ← 2#000#;
```

```
else
   -- there is one or more messages
   if port_header.enq_mode = fifo then
      dequeue the first message;
   else
      dequeue the first message from the highest priority non-empty queue;
   end if;

   -- unlock the port
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   port_header.lock_byte ← port_header.lock_byte and not 2#1#;
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;

   dst ← dequeued_messgae;
   AC.cc ← 2#010#
end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

condwait

Mnemonics:

condwait	668	REG	Conditional Wait

Format:

condwait *sem_ad*
reg
AD

Description:

Attempts to wait on the semaphore and sets the condition code to indicate whether the wait was completed successfully or not. The *sem_ad* operand contains the AD of the semaphore.

If the queue tail is non-zero or the count is zero, the condition code is set to 000_2. Otherwise, the count is decremented by one and the condition code is set to 010_2.

This instruction is similar to the **wait** instruction, except that with the **wait** instruction, the process is suspended and enqueued on the semaphore if the semaphore queue tail is non-zero or the semaphore count is zero.

Action:

```
if not sem_ad.tag then
   raise invalid-AD fault;
elsif not sem_ad.type_rights_2 then
   raise type-rights fault;
end if;

sem_desc_offset ← sem_ad.objec_index * 16;
sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, quad_mixed);

if sem_desc.entry_type ≠ invalid_descriptor then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
         semd_desc.word_1;
   raise invalid-descriptor fault;
elsif sem_desc.entry_type ≠ embedded_descriptor or
   sem_desc.object_type ≠ semaphore then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
         semd_desc.word_1;
   raise type-mismatch fault;
end if;

while (sem_desc.lock_byte and 2#1#) = 2#1#) loop
   -- wait until the semaphore is unlocked
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ←
         semd_desc.word_0;
   delay;
   sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, long_mixed);
end loop;
```

```
if sem_desc.sem_tail ≠ 0 or else sem_desc.count = 0 then
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;
   AC.cc ← 2#000#;

else
   sem_desc.sem_count ← - 1;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;
   AC.cc ← 2#010#;
end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

cosr, cosrl

Mnemonics:

cosr	68D	REG	Cosine Real
cosrl	69D	REG	Cosine Long Real

Format:

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

Description:

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

For the **cosrl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

The 80960MC uses a value for π with a 66-bit mantissa. As a result, the cosine of any values never equals to 0. Refer to the section in Chapter 5 titled "Pi" for further details.

Action:

dst ← **cosine** (*src*);

Faults:

Floating Reserved Encoding	
Floating Invalid Operation	when *src* is ∞.
Floating Inexact	when *src* is not 0.

cpyrsre, cpysre

Mnemonics:

cpysre	6E2	REG	Copy Sign Real Extended
cpyrsre	6E3	REG	Copy Reversed Sign Real Extended

Format:

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

Description:

Copies *src1* into *dst* with the exception of the sign bit. For the **cpysre** instruction, the sign of *src2* is copied to *dst*; for the **cpyrsre** instruction, the opposite of the sign of *src2* is copied to *dst*.

If any operand references a general register, three successive registers are used.

These instructions only operate on extended-real formats. The same operations can be performed on real- and long-real values using the **setbit** and **clearbit** instructions, or a combination of the **chkbit** and **alterbit** instructions. The same technique should be used for extended-real operands if they are stored in general registers.

Action:

```
cpysre    dst ← src1;
          dst.sign ← sign(src2);

cpyrsre   dst ← src1;
          dst.sign ← not sign(src2);
```

Faults:

cread

Mnemonic:

cread	648	REG	Create Access Descriptor

Format:

cread	*tdo_ad,*	*ad_image,*	*dst*
	reg	reg	reg

Description:

Copies the AD image in *ad_image* with the tag bit set to the *dst*. The creation of ADs is under the controls of a type definition object specified by *tdo_ad.*

Action:

```
if tdo_ad.tag = 0 then
   raise invalid-AD faults;
elsif tdo_ad.type_rights_1 then
   # needs creation rights
   raise type-rights fault;
elsif object_type_of(tdo_ad) ≠ TDO then
   # must be TDO object type
   raise type-mismatch fault;
elsif super TDO bit in tdo_ad object = 0 then
   # must be super TDO
   raise type-mismatch fault;
end if;

dst ← ad_image;
dst.tag ← 1;
```

Faults:

Invalid AD Faults

Type Mismatch Fault

Type Rights Fault

cvtadr

Mnemonic:

cvtadr	672	REG	Convert Address

Format:

cvtadr	*src*,	---,	*dst*
	reg/lit		reg

Description:

Converts the linear address in *src* into a virtual address (AD plus offset). The region offset is stored in the register specified by *dst*, and the region AD is stored in the next highest register.

If this instruction is executed in physical mode, the value returned is undefined.

Action:

```
# src1 is an address in the processor's current
# linear address space.
dst ← region offset of src1
dst + 1 ← region AD of src1
```

Faults:

cvtilr, cvtir

Mnemonics:

cvtir	674	REG	Convert Long Integer to Real
cvtilr	675	REG	Convert Integer to Real

Format:

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

Description:

Converts the integer in *src* to a real and stores the result in *dst*. For the **cvtilr** instruction, the *src* operand references the first (lowest numbered) of two successive registers.

Converting an integer to long real format requires two instructions. The integer is first converted to extended real, then the extended-real is converted to long real using the **movrl** instruction.

Action:

dst ← **real** (*src*);

Faults:

Floating Inexact

Example:

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

cvtri, cvtril, cvtzri, cvtzril

Mnemonics:

cvtri	6C0	REG	Convert Real To Integer
cvtril	6C1	REG	Convert Real To Integer Long
cvtzri	6C2	REG	Convert Truncated Real To Integer
cvtzril	6C3	REG	Convert Truncated Real To Long Integer

Format:

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

Description:

Converts the *src* to an integer and stores the result in *dst*. The conversion **never** raise floating inexact exception. The nontruncated versions of these instructions round according to the current rounding mode in the Arithmetic Controls. The truncated versions always round toward zero.

For the **cvtril** and **cvtzril** instructions, the *dst* operand references the first (lowest numbered) of two successive registers.

Converting a long real value to an integer requires two instructions. The long real value is first converted to extended real (always exact), then one of the convert real-to-integer instructions can be used. The example section below illustrates this conversion.

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

Action:

dst ← **integer** (*src*1);
src1 is rounded to integer value

Faults:

Floating Reserved Encoding

Integer Overflow

Example:

```
# Conversion of long real value to a long integer
movrl g4, fp2    # convert long-real to extended-real
cvtril fp2, g12  # extended-real to long integer
```

daddc

Mnemonic:

daddc	642	REG	Decimal Add With Carry

Format:

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

Description:

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

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

Action:

```
# Let the value of the condition code be xCx.
temp ← (src2 and 2#1111#) + (src1 and 2#1111#) + C;
if temp > 10 then
   temp ← temp - 10;
   AC.cc ← 2#010#;
else
   AC.cc ← 2#000#;
end if;
dst ← (src2 and not 2#1111#) + temp;
```

Faults:

See Also:

dsubc, dmovt

divi, divo

Mnemonic:

divi	74B	REG	Divide Integer
divo	70B	REG	Divide Ordinal

Format:

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

Description:

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

The **divi** instruction can signal an integer-overflow fault.

Action:

dst ← *src2* / *src1*;

Faults:

Integer Overflow	In **subi**, -2^31 / -1.
Arithmetic Zero Divide	When *src1* is 0.

divr, divrl

Mnemonic:

divr	78B	REG	Divide Real
divrl	79B	REG	Divide Long Real

Format:

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

Description:

Divides *src2* by *src1* and stores the result in *dst*. The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source values is 0 or ∞.

For the **divrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	INV	+∞	+∞	-∞	-∞	INV	NaN
-F	+0	+F	+Z	-Z	-F	-0	NaN
-0	+0	+0	INV	INV	-0	-0	NaN
+0	-0	-0	INV	INV	+0	+0	NaN
+F	-0	-F	-Z	+Z	+F	+0	NaN
+∞	INV	-∞	-∞	+∞	+∞	INV	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:
F Means finite-real number.
INV Indicates floating invalid-operation exception.
Z Indicates floating zero-divide exception, or ∞ when masked.

Action:

dst ← *src2* / *src1*;

Faults:

Floating Zero Divide	when *src1* is 0 and *src2* is finite.
Floating Invalid Operation	when *src1* and *src2* are 0.
Floating Inexact	
Floating Overflow	
Floating Underflow	
Floating Reserved Encoding	

See Also:

scaler

dmovt

Mnemonic:

dmovt	644	REG	Decimal Move And Test

Format:

dmovt	*src*, reg/lit	---,	*dst* reg

Description:

Copies the *src* value into *dst*. If the least-significant eight bits of *src* are between 00110000_2 (ASCII 0) and 00111001_2 (ASCII 9), the condition code is set to 000_2 for valid ASCII decimal; otherwise, it is set to 010_2.

This instruction can be used iteratively to validate decimal strings.

Action:

```
dst ← src;
temp ← bits 0-7 of src;
if (temp ≥ 2#0011000#) and (temp ≤ 2#00111001#) then
          AC.cc ← 2#000#;
else
          AC.cc ← 2#010#;
end if;
```

Faults:

See Also:

daddc, dsubc

dsubc

Mnemonic:

dsubc	643	REG	Decimal Subtract With Carry

Format:

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

Description:

Subtracts (in decimal) the bits 0-3 of *src1* and (1 - carry) from the bits 0-3 of *src1*, and stores the result in bits 0-3 of *dst*. Bit 1 of the condition code is used as the carry bit, which is the complement of the borrow bit. If the (decimal) subtraction results in a carry (or no borrow), bit 1 of the condition code is set. Bits 0 and 2 of the condition code are set to zeros. Bits 4-31 of *src* are copied to bits 4-31 of the *dst*.

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

Action:

```
# Let the value of the condition code be xCx.
temp ← (src2 and 2#1111#) - (src1 and 2#1111#) - (1 - C);
if temp < 0 then
   temp ← temp + 10;
   AC.cc ← 2#000#;
else
   AC.cc ← 2#010#;
end if;
dst ← (src2 and not 2#1111#) + temp;
```

Faults:

See Also:

daddc, dmovt

ediv

Mnemonic:

ediv	671	REG	Extended Divide

Format:

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

Description:

Divides *src2* by *src1* and stores the quotient and remainder in *dst*. This instruction performs ordinal arithmetic. The *src2* is a long ordinal (i.e., 64 bits), which is contained in two adjacent registers. The *src2* operand specifies the lower numbered register, which contains the least significant bits of the operand. The *src2* operand must be an even numbered register. The *src1* is a normal ordinal (i.e., 32 bits).

The remainder is stored in the register designated by *dst* and the quotient is stored in the next highest numbered register. The *dst* operand must be an even numbered register.

If this operation overflows, the quotient does not fit in 32-bits, no fault is raised and the result is undefined.

Action:

```
dst.lower ← (src2 - (src2 / src1) * src1); # remainder
dst.upper ← (src2 / src1); # quotient
```

Faults:

Arithmetic Integer Divide

See Also:

emul

emul

Mnemonic:

emul	670	REG	Extended Multiply

Format:

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

Description:

Multiplies *src2* by *src1* and stores the result in *dst*. This instruction performs ordinal arithmetic. The result is a long ordinal (i.e., 64 bits), which is stored in two adjacent registers. The *dst* operand specifies the lower numbered register, which receives the least significant bits of the result. The *dst* operand must be an even numbered register.

Action:

```
dst.lower ← (src1 * src2) mod 2^32;
dst.upper ← (src * src2)/mod 2^32;
```

Faults:

See Also:

ediv

expr, exprl

Mnemonic:

expr	689	REG	Exponent Real
exprl	699	REG	Exponent Long Real

Format:

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

Description:

Computes 2 to the power of the *src*, then minus 1, and stores the result in *dst*. The *src* value must be within the range of -0.5 to +0.5, inclusive. If the *src* value is outside this range, the result is undefined.

For the **exprl** instruction, if any operand references a general register, two successive registers are used.

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

Src	Dst
-0.5 to -0	-(1/√2)-1 to -0
-0	-0
+0	+0
+0 to +0.5	+0 to √2-1

Notes:
*** Results are unpredictable.

Action:

dst ← (2^*src*) - 1;

Faults:

Floating Reserved Encoding

Floating Underflow

Floating Invalid Operation

Floating Inexact

Example:

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

```
expr    g0,g0
addr    0f1.0,g0,g0
cvtri   fp0,g1
scaler  g1,fp0,g0
```

See Also:

scaler, logr

extract

Mnemonic:

extract	651	REG	Extract

Format:

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

Description:

Shifts *src/dst* right by *bitpos* with zero fills, and copies the least-significant *len* bits to *src/dst*.

Action:

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

Faults:

FAULT IF

Mnemonic:

faulte	1A	CTRL	Fault If Equal
faultne	1D	CTRL	Fault If Not Equal
faultl	1C	CTRL	Fault If Less
faultle	1E	CTRL	Fault If Less Or Equal
faultg	19	CTRL	Fault If Greater
faultge	1B	CTRL	Fault If Greater Or Equal
faulto	1F	CTRL	Fault If Ordered
faultno	18	CTRL	Fault If Unordered

Format:

fault*

Description:

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

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

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

Action:

```
if ((mask and AC.cc) ≠ 2#000#) or (mask = AC.cc) then
   raise constraint-range fault;
end if;
```

Faults:

Constraint Range

fill

Mnemonic:

fill	617	REG	Fill String

Format:

fill	*dst*	*value*	*len*
	addr	reg/lit	reg/lit

Description:

Fills a string in memory with repeated copies of the word given in *value*. The *dst* operand specifies the address of the first byte of the string, and the *len* operand specifies the length of the string in bytes.

Action:

```
for i in 0 .. (len/4) - 1 loop
    word (dst + i) ← value;
end loop;
case len rem 4 is
    when 0 => null;
    when 1 => byte (dst + len - 1) ← value;
    when 2 => halfword(dst + len - 2) ← value;
    when 3 => halfword(dst + len - 3) ← value;
              byte (dst + len - 1) ← value/65536;
end case;
```

Faults:

MEMORY_FAULT

flushreg

Mnemonic:

flushreg	66D	REG	Flush Local Registers

Format:

flushreg

Description:

Copies the contents of all the cached local-register sets, with the exception of the current one, into their associated register-save areas in the procedure stack. All the non-current local-register sets are marked invalid. This allows software to examine the contents of previous frames, and forces a return instruction to load the previous frame from memory. It is possible to perform non-local returns if the current L0 or the RIP of a previous frame is modified.

Action:

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

Faults:

fmark

Mnemonic:

fmark 66C REG Force Mark

Format:

fmark

Description:

Generates a breakpoint trace-event, regardless of the breakpoint trace mode.

The **fmark** instruction differs from **mark** where the breakpoint trace mode must be enabled. For more information on trace-fault generation, refer to Chapter 10.

Action:

```
if process_controls.trace_enable then
   raise breakpoint-trace fault;
end if;
```

Faults:

Breakpoint Trace

inspacc

Mnemonic:

inspacc	613	REG	Inspect Access

Format:

inspacc	*src_addr*,	---,	*dst*
	reg		reg
	addr		

Description:

Loads the effective page representation rights of the byte specified by *src_addr* in *dst*. The *src_addr* is an address contained in register(s).

The page representation rights are contained in a two-bit field (bits 1 and 2) in the page table entry for the page that contains the selected byte. This field is loaded into bits 0 and 1 of the *dst*.

Action:

- If the reference is via an invalid descriptor, raises an invalid-descriptor fault.
- If the offset is greater than the length of the segment, raises a object-length fault.
- If the object is a simple object, returns 2#11#.
- Otherwise, returns the effective page-representation rights of the access path in bits 0 and 1 of the *dst* operand.

Faults:

Invalid Descriptor, Object Length

LOAD

Mnemonic:

ld	90	MEM	Load
ldob	80	MEM	Load Ordinal Byte
ldos	88	MEM	Load Ordinal Short
ldib	C0	MEM	Load Integer Byte
ldis	C8	MEM	Load Integer Short
ldl	98	MEM	Load Long
ldt	A0	MEM	Load Triple
ldq	B0	MEM	Load Quad

Format:

ld*	*src,*	*dst*
	mem	reg

Description:

Copies a byte or string of bytes from memory into a register or group of successive registers. The *src* operand specifies the address of the first byte to be loaded. The full range of linear addressing modes may be used in specifying *src*. (Refer to Section 17.5 for the different addressing modes.)

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

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

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

Action:

dst ← memory (*src*);

Faults:

MEMORY FAULTS

ldm

Mnemonics:

ldm	D0	MEM	Load Mixed
ldml	D8	MEM	Load Mixed Long
ldmq	F0	MEM	Load Mixed Quad

Format:

ldm*	*src,*	*dst*
	mem	reg

Description:

Loads *src*, including the tag bits, into *dst*. The *src* operand is a memory type, which allows the full range of linear addressing modes to be used to specify the word or words in memory to be loaded. (Refer to Section 17.5 for the different addressing modes.)

If multiple words are loaded, they are placed in adjacent registers. Here, the *dst* operand specifies the lowest numbered register, which receives the word from the address spacified with *src*.

Action:

dst ← read (*src*);

Faults:

MEMORY FAULTS

LOAD VIRTUAL

Mnemonic:

ldv	91	MEM	Load Virtual
ldvib	C1	MEM	Load Virtual Integer Byte
ldvis	C9	MEM	Load Virtual Integer Short
ldvob	81	MEM	Load Virtual Ordinal Byte
ldvos	89	MEM	Load Virtual Ordinal Short
ldvl	99	MEM	Load Virtual Long
ldvt	A1	MEM	Load Virtual Triple
ldvq	B1	MEM	Load Virtual Quad

Format:

ldv*	*src,*	*dst*
	mem	reg

Description:

Copies a byte or string of bytes from memory into a register or group of successive registers, with tag bits ignored. The *src* operand specifies the virtual address of the first byte to be loaded. The full range of virtual addressing modes may be used in specifying *src*. (Refer to Section 17.5 for the different addressing modes.)

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

The **ldvob** and **ldvib**, and **ldvos** and **ldvis** instructions load a byte and half word, respectively, and convert it to a full 32-bit word. The **ldv**, **ldvl**, **ldvt**, and **ldvq** instructions copy 4, 8, 16, and 32 bytes, respectively, from memory into to successive registers.

Action:

dst ← memory (*src*);

Faults:

MEMORY FAULTS

ldvm

Mnemonics:

ldvm	D1	MEM	Load Virtual Mixed
ldvml	D9	MEM	Load Virtual Mixed Long
ldvmq	F1	MEM	Load Virtual Mixed Quad

Format:

ldvm*	*src*,	*dst*
	mem	reg

Description:

Loads *src*, including the tag bits, into *dst*. The *src* operand is a memory type, which allows the full range of virtual addressing modes to be used to specify the word or words in memory to be loaded. (Refer to Section 17.5 for the different addressing modes.)

If multiple words are loaded, they are placed in adjacent registers. Here, the *dst* operand specifies the lowest numbered register, which receives the word from the address spacified with *src*.

Action:

dst ← read (*src*);

Faults:

MEMORY FAULTS

lda

Mnemonic:

lda	8C	MEM	Load Address

Format:

lda	*src*	*dst*
	mem efa	reg

Description:

Computes the effective address specified with *src* and stores it in *dst*. (Refer to Section 17.5 for the different addressing modes.) The *src* address is not checked for validity.

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

Action:

dst ← effective_address(*src*);

Faults:

Example:

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

lda 0x749, r8     # loads the constant 16#749#
                  # in r8
```

ldscp

Mnemonic:

ldcsp	657	REG	Load Control Stack Pointer

Format:

ldcsp	---,	---,	*dst*
			reg

Description:

Loads the control stack pointer of the current process into *dst*. The control stack pointer in the environment is **not** kept up to date when the process is running.

Action:

dst ← current control stack pointer

Faults:

ldglobals

Mnemonic:

ldglobals	64A	REG	Load From Process Globals

Format:

ldglobals	*offset,* reg/lit	---,	*dst* reg

Description:

Loads a word from the process-globals object into *dst*. The *offset* specifies the offset of the word in the process-globals object. The AD of the process-globals object is specified in the process object for the current process.

Action:

dst ← **read** (*src*)

Faults:

ldphy

Mnemonic:

ldphy	614	REG	Load Physical Address

Format:

ldphy	*src*, reg addr	---,	*dst* reg

Description:

Translates the linear or virtual address in *src* into the corresponding physical address and stores the result in *dst*. This instruction is provided to convert linear or virtual addresses into physical addresses.

Action:

dst ← physical address (*src*)

Faults:

MEMORY FAULT

ldtime

Mnemonic:

ldtime	673	REG	Load Process Time

Format:

ldtime	---,	---,	*dst*
			reg

Description:

Stores the elapsed execution time (in units of ticks) of the current process up until the time of execution of this instruction in *dst*. The elapsed time is computed by subtracting the execution time (ET) from the residual time slice (RTS). Both of these values are cached in the processor.

Action:

dst ← execution_time - residual_time_slice;

Faults:

ldtypdef

Mnemonic:

ldtypdef	649	REG	Load Type Definition

Format:

ldtypdef	*src,* reg	---,	*dst* reg

Description:

Stores the type definition object AD associated with the *src* object in *dst.*

Action:

```
if src.tag = 0 then
   raise invalid-AD fault;
elsif OTE_of(src).entry_type is embedded or invalid type and
     OTE_of(src).use_default_tdo then
   # use default TDO
   dst ← AD of default TDO
else
   # use the TDO ad in the OTE
   dst ← OTE_of(src).TDO;
endif
```

Faults:

Invalid AD Fault

MEMORY FAULTS

logbnr, logbnrl

Mnemonic:

logbnr	68A	REG	Log Binary Real
logbnrl	69A	REG	Log Binary Long Real

Format:

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

Description:

Computes the $\log_2$ *src* and stores the integral part of this value as a real number in *dst.* The result of this operation is the unbiased exponent of the number. When the *src* is a denormalized number, the number is first normalized before the unbiased exponent is stored in *dst.*

This instruction implements the IEEE recomended function *logb.* It is useful for calculating the order of magnitude of a number. If the fractional part of $\log_2$ *src* is needed, use the **logr** or **logrl** instruction.

For the **logbnrl** instruction, if any operand references a general register, two successive registers are used.

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

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

Notes:
F Means finite-real number.
Z Indicates floating zero-divide exception, or ∞ when masked.

Action:

```
dst ← truncate_to_integral (log$_2$ src);
# return the integral part of log$_2$ src)
```

Faults:

Floating Reserved Encoding	
Floating Invalid Operation	
Floating Zero Divide	when *src* is 0.

logepr, logeprl

Mnemonic:

logepr	681	REG	Log Epsilon Real
logeprl	691	REG	Log Epsilon Long Real

Format:

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

Description:

Computes (*src2* * $\log_2$ (*src1* + 1)), and stores the result in *dst*. The range of *src1* is restricted to the following:

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

For the **logeprl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

This instruction offers optimal accuracy when *src1* + 1 is close to 1, or *src1* is close to 0. This is commonly found in compound interest and annuity calculations. When the *src1* operand is outside the specified range, the **logr** or **logrl** instruction can be used with very insignificant loss of accuracy.

This instruction can be used to implement logarithm of a different base with the following identity:

$$\log_n m = \log_n 2 * \log_2 m$$

Action:

dst ← *src2* * $\log_2$ (*src1* + 1);

Faults:

Floating Reserved Encoding	
Floating Overflow	
Floating Underflow	
Floating Invalid Operation	when *src1* is 0 and *src2* is ∞.
	when *src1* falls outside the range defined above.
Floating Inexact	

logr, logrl

Mnemonic:

logr	682	REG	Log Real
logrl	692	REG	Log Long Real

Format:

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

Description:

Computes (*src2* * $\log_2$ (*src1*)), and stores the result in *dst*.

For the **logrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	*	*	**	**	± ∞	-∞	NaN
-F	*	*	**	**	± F	-∞	NaN
-0	*	*	*	*	± 0	*	NaN
+0	*	*	*	*	± 0	*	NaN
+F	*	*	**	**	± F	± ∞	NaN
+∞	*	*	**	**	± F	± ∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

If *src1* is very close to 1, the result may suffer significant loss of precisoin. In such case, use the **logepr** or **logeprl** instruction.

These instructions can be used to implement logarithm of a different base with the following identity:

$$\log_n m = \log_n 2 * \log_2 m$$

Action:

dst ← *src2* * $\log_2$ (*src1*);

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Zero Divide	when *src1* is 0 and *src2* is finite.
Floating Invalid Operation	when *src1* is less than 0.
	when *src1* and *src2* are 0.
	when *src1* is 1 and *src2* is ∞.
	when *src1* is ∞ and *src2* is 0.
Floating Inexact	always

mark

Mnemonic:

mark	66B	REG	Mark

Format:

mark

Description:

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

The **mark** instruction differs from **fmark** where the breakpoint trace mode is ignored. For more information on trace-fault generation, refer to Chapter 11.

Action:

```
if process_controls.trace_enable and trace_controls.breakpoint_trace_mode then
   raise breakpoint-trace fault;
end if;
```

Faults:

Breakpoint Trace

modac

Mnemonic:

modac	645	REG	Modify AC

Format:

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

Description:

Reads and modifies the arithmetic controls for the current process. The *src* contains new value for the arithmetic controls and the *mask* specifies the bits that may be changed. Only the bits set in *mask* are modified in the arithmetic controls. The initial arithmetic controls is copied into *dst.*

Action:

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

Faults:

modi

Mnemonic:

modi	749	REG	Modulo Integer

Format:

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

Description:

Divides *src2* by *src1*, where both are integers, and stores the modulo remainder of the result in *dst*. The sign of the result (if nonzero) is the same as the sign of *src1*. The result is between 0 (inclusive) and *src1* (exclusive).

Action:

```
tmp ← src2 - ((src2/src1) * src1);
if (src2 * src1 < 0) and (tmp /= 0) then
  then tmp ← tmp + src1;
end if;
dst ← tmp;
```

Faults:

Arithmetic Zero Divide

modify

Mnemonic:

modify	650	REG	Modify

Format:

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

Description:

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

Action:

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

Faults:

modpc

Mnemonic:

modpc	655	REG	Modify Process Controls

Format:

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

Description:

Reads and modifies the process controls for the current process. The processor changes its internally cached process controls as specified with *mask* and *src/dst*. The *src/dst* contains the new process controls and the *mask* operand specifies the bits that may be changed. Only the bits set in the mask are modified in the process controls. Once the process controls have been changed, their initial value is copied into *src/dst*. The *prcs_ad* points to the current process AD.

The *prcs_ad* must match the current process AD, or the processor must be in the supervisor mode to modify the process controls. If the *mask* operand is set to 0, the **modpc** returns the process controls, without modifying the process controls.

If the priority of the current process being lowered, pending interrupts are checked.

Changing the state, resume, internal state, and trace enable fields of the process controls can lead to unpredictable behavior, as described in Section 15.5.

Action:

```
if mask ≠ 0 or process_controls.execution_mode ≠ supervisor then
   if prcs_ad ≠ current_process_ad then
      raise type-mismatch fault;
   end if;
   if not prcs_ad.type_rights_3 then
      raise type-rights fault;
   end if;
end if

temp ← process.process_controls;
process_controls ← (src/dst and mask) or (process_controls and not (mask));
src/dst ← temp;

if temp.priority > process_controls.priority then
   check_pending_interrupts;
end if;
```

Faults:

Type Mismatch

modtc

Mnemonic:

modtc	654	REG	Modify Trace Controls

Format:

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

Description:

Reads and modifies the trace controls for the current process. The processor changes trace controls as specified with *mask* and *new_tc*. The *new_tc* contains the new trace controls and the *mask* specifies the bits that may be changed. Only the bits set in the mask are modified in the trace controls. Once the trace controls have has been changed, their initial trace controls is copied into *dst.*

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

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

For more information on the trace controls, refer to Chapters 11.

Action:

```
temp ← trace_controls;
temp1 ← 16#00FF00FF# and mask;
trace_controls ← (new_tc and temp1) or (trace_controls and not temp1);
dst ← temp;
```

Faults:

MOVE

Mnemonic:

mov	5CC	REG	Move
movl	5DC	REG	Move Long
movt	5EC	REG	Move Triple
movq	5FC	REG	Move Quad

Format:

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

Description:

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

For the **movl, movt,** and **movq** instructions, the *src* and *dst* operands specify the first (lowest numbered) register of several successive registers.

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

Action:

dst ← *src*;

Faults:

MOVE MIXED

Mnemonic:

movm	5CD	REG	Move Mixed
movml	5DD	REG	Move Mixed Quad
movmq	5FD	REG	Move Mixed Long

Format:

movm*	*src,*	*dst*
	reg	reg

Description:

Copies the content of *src* (including the tag bits) to *dst*. For the **movml** and **movmq** instructions, the *src* and *dst* operands specify the first (lowest numbered) register of several successive registers.

These instructions are intended for moving words among registers that contain ADs or combinations of data and ADs.

For the **movml** and **movmq** instructions, the *src* and *dst* operands specify the first (lowest numbered) register of several successive registers.

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

Action:

dst ← *src*

Faults:

movqstr

Mnemonic:

movqstr	604	REG	Move Quick String

Format:

movqstr	*dst,*	*src,*	*len*
	addr	addr	reg/lit

Description:

Copies a string of bytes from one location in memory to another, where the source and destination strings are assumed not to overlap. The *src* specifies the address of the first byte of the source string and the *dst* specifies the address of the first byte of the destination string. The *len* specifies the length of the string in bytes, from 0 to 2^{32}-1.

If the strings overlap, the value copied is not predictable. (Use the **movstr** instruction instead.)

Action:

```
for i in 0 .. len - 1 loop
  byte (dst + i) ← byte (src + i);
end loop;
```

Faults:

MEMORY FAULT

movr, movre, movrl

Mnemonic:

movr	6C9	REG	Move Real
movrl	6D9	REG	Move Long Real
movre	6E9	REG	Move Extended Real

Format:

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

Description:

Copies a real value from one or more source registers (specified with the *src* operand) to one or more destination registers (specified with the *dst* operand). The **movr** and **movrl** instructions can raise a floating-point exception or faults. The **movre** instruction can never raise an exception and thus never faults.

For the **movrl** instruction, if any operand references a general register, two successive registers are used. For the **movre** instruction, if any operand references a general register, three successive registers are used.

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

When the **movre** instruction moves an operand from general registers to a floating-point register, the most-significant 16 bits of the third register is ignored (refer to Figure 5-5). Likewise, when this instruction is used to move an operand from a floating-point register to general registers, the most-significant 16 bits of the third register are filled with zero.

These instruction are mainly used to copy numbers between general registers and floating-point registers, and within floating-point registers. When a floating-point value is copied without format conversion, the source and destination formats are the same, the **mov** and **movl** instructions are faster and never raise a floating-point exception. When the extended real value is in general registers, the **movt** instruction should be used instead.

Action:

dst ← *src*;

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation

Floating Inexact

Example:

```
# Conversion of real value to long real
movr   g3, fp2
movrl  fp2, g4
```

movstr

Mnemonic:

movstr	605	REG	Move String

Format:

movstr	*dst,*	*src,*	*len*
	addr	addr	reg/lit

Description:

Copies a string of bytes from one location in memory to another. The *src* specifies the address of the first byte of the source string and the *dst* specifies the address of the first byte of the destination string. The *len* specifies the length of the string in bytes, from 0 to 2^{32}-1.

If the strings overlap, the **movstr** guarantees that no byte of the source string is overwritten before it is copied into the destination string. If it is guaranteed that there are no overlaps, the **movqstr** instruction should be used instead. The overlap check is based on virtual addresses, thus virtual address aliases will not be detected.

Action:

```
if src ≤ dst then
   for i in 1 .. len loop
      byte (dst + len - i) ← byte (src + len - i);
   end loop;
else
   for i in 0 .. len - 1 loop
      byte (dst + i) ← byte (src + i);
   end loop;
end if;
```

Faults:

MEMORY FAULT

muli, mulo

Mnemonic:

muli	741	REG	Multiply Integer
mulo	701	REG	Multiply Ordinal

Format:

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

Description:

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

Action:

dst ← *src2* * *src1*;

Faults:

Integer Overflow (in **muli**)

mulr, mulrl

Mnemonic:

mulr	78C	REG	Multiply Real
mulrl	79C	REG	Multiply Long Real

Format:

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

Description:

Multiplies *src2* by *src1* and stores the result in *dst*. The sign of the result is always the exclusive-OR of the source signs, even if one or more of the source is 0 or ∞.

For the **mulrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → **Src2 ↓**	**-∞**	**-F**	**-0**	**+0**	**+F**	**+∞**	**NaN**
-∞	+∞	+∞	INV	INV	-∞	-∞	NaN
-F	+∞	+F	+0	-0	-F	-∞	NaN
-0	INV	+0	+0	-0	-0	INV	NaN
+0	INV	-0	-0	+0	+0	INV	NaN
+F	-∞	-F	-0	+0	+F	+∞	NaN
+∞	-∞	-∞	INV	INV	+∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

The **scaler** and **scalerl** instructions can be used to multiply by the power of 2.

Action:

dst ← *src2* * *src1*;

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation when 0 * ∞

Floating Inexact destination format.

See Also:

scaler

nand

Mnemonic:

nand	58E	REG	Nand

Format:

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

Description:

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

Action:

dst ← not (*src2* and *src1*);

Faults:

See Also:

LOGICAL

nor

Mnemonic:

nor	588	REG	Nor

Format:

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

Description:

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

Action:

```
dst ← not (src2 or src1);
```

Faults:

See Also:

LOGICAL

not, notand

Mnemonic:

not	58A	REG	Not
notand	584	REG	Not And

Format:

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

Description:

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

Action:

not: *dst* ← not *src1*;

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

Faults:

See Also:

LOGICAL

notbit

Mnemonic:

notbit	580	REG	Not Bit

Format:

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

Description:

Copies the *src* to *dst* with one bit inverted. The *bitpos* operand specifies the bit to be inverted.

Action:

dst ← *src* xor 2^(*bitpos* mod 32);

Faults:

See Also:

alterbit, chkbit, clrbit, setbit

notor

Mnemonic:

notor	58D	REG	Not Or

Format:

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

Description:

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

Action:

dst ← (not *src2*) or *src1*;

Faults:

See Also:

LOGICAL

or, ornot

Mnemonic:

or	587	REG	Or
ornot	58B	REG	Or Not

Format:

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

Description:

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

Action:

or: *dst* ← *src2* or *src1*;

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

Faults:

See Also:

LOGICAL

receive

Mnemonic:

receive	656	REG	Receive

Format:

receive	*port_ad,* reg AD	---,	*dst* reg AD

Description:

Receive a message from a port. The *port_ad* contains the AD of the port. If the port has enqueued messages, the AD of the message at the head of the message queue is stored in *dst* and execution continues.

If the port is empty (i.e., has no messages queued), the process is suspended, with its IP left pointing to the current instruction. The process is then enqueued at the port at the tail of the blocked-processes queue.

The receive-blocked process remains blocked until a message is bound to it by another *send* or *sendserv* instruction. This message is then stored in the process object of the blocked process, and the process is dequeued from the port and enqueued at its dispatching port.

When the process is again dispatched, the processor resumes the **receive** instruction, but this time it reads the message stored in its process object, rather than going to the communication port again.

Action:

```
if not port_ad.tag then
   raise invalid-AD fault;
elsif not port_ad.type_rights_2 then
   raise type-rights fault;
elsif object_type_of(port_ad) ≠ port_type then
   raise type-mismatch fault;
end if;

-- loop the port
loop
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   if (port_header.lock_byte and 2#1#) = 2#0#) then exit end if;
   -- wait until the semaphore is unlocked
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;
   delay;
end loop;
```

```
if port_header.queue_state = process or else
  (port_header.enq_mode = fifo and port_header.queue_head = 0) or
  (port_header.enq_mode = priority and port_header.queue_status = 0) then
  -- port is empty, suspend process, and enqueue process
  suspend current process;
  if port_header.enq_mode = fifo then
    enqueue current process;
  else
    enqueue current process at the current process priority;
  end if;

  -- unlock the port
  port_header ← atomic_read_ad(port_ad, 0, long_word);
  port_header.lock_byte ← port_header.lock_byte and not 2#1#;
  atomic_write_va(port_ad, 0, word) ← port_header.word_0;

  -- the processor is free
  perform dispatch action;

else
  -- there is one or more messages
  if port_header.enq_mode = fifo then
    dequeue the first message;
  else
    dequeue the first message from the highest priority non-empty queue;
  end if;

  -- unlock the port
  port_header ← atomic_read_ad(port_ad, 0, long_word);
  port_header.lock_byte ← port_header.lock_byte and not 2#1#;
  atomic_write_va(port_ad, 0, word) ← port_header.word_0;

  dst ← dequeued_messgae;
end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

remi, remo

Mnemonic:

remi	748	REG	Remainder Integer
remo	708	REG	Remainder Ordinal

Format:

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

Description:

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

Action:

```
dst ← src2 - ((src2 / src1) * src1);
```

Faults:

remr, remrl

Mnemonic:

remr	683	REG	Remainder Real
remrl	693	REG	Remainder Long Real

Format:

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

Description:

Divides *src2* by *src1* and stores the remainder in *dst.* The sign of the result is the same as the sign of *src2*.

For the **remrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	INV	INV	INV	INV	INV	INV	NaN
-F	src2	-F or -0	INV	INV	-F or -0	src2	NaN
-0	-0	-0	INV	INV	-0	-0	NaN
+0	+0	+0	INV	INV	+0	+0	NaN
+F	src2	+F or +0	INV	INV	+F or +0	src2	NaN
+∞	INV	INV	INV	INV	INV	INV	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

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

The **remr** and **remrl** instructions are different from the remainder described in the IEEE floating-point standard. The difference is the integral quotient (N) of the expression (*src2/src1*) is truncated toward zero, while the standard calls for rounding toward the nearest even integeral quotient.

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

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

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

Action:

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

Faults:

Floating Reserved Encoding	
Floating Overflow	
Floating Underflow	
Floating Invalid Operation	when *src2* is ∞ or *src1* is 0.
Floating Inexact	

Example:

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

restrict

Mnemonic:

restrict	652	REG	Restrict Rights

Format:

restrict	---,	*rtsmsk*,	*src/dst*
		reg/lit	reg

Description:

Restricts the rights in the AD in *src/dst* using a rights mask in *rtsmsk*. The AD with restricted rights is stored in *src/dst*. The rights to be restricted are set to 0 in the rights mask.

Action:

```
if src/dst.tag = 1 then
   src/dst ← src/dst and not (rtsmsk and 2#11111#);
endif
```

Faults:

See Also:

amplify

resumprcs

Mnemonic:

resumprcs 664 REG Resume Process

Format:

```
resumprcs prcs_ad,    ---,      ---
          reg
          AD
```

Description:

Bind the processor to a new PCB specified by *prcs_ad*.

Any state information for the current process that has been cached on the processor chip, such as the PCB, the environment table, and the stack frames, is discarded (i.e., not updated in memory, not unlocked). Thus, to save the state of the current process, the **resumprcs** instruction should be preceded by a **saveprcs** instruction.

The **saveprcs** and **resumprcs** instructions are similar to the **save** and **resume** functions in most UNIX kernels. These instructions allow task (or process) switching without using the processor's automatic dispatching mechanism.

Action:

```
if prcs_ad.tag = 0 then
   raise invalid-ad fault;
elsif prcs_ad.type_right_3 = 0 then
   raise type-rights fault;
elsif object_type_of(prcs_ad) ≠ process_type then
   raise type-mismatch fault;
end if;

perform process-bind action
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

ret

Mnemonic:

ret	0A	CTRL	Return

Format:

ret

Description:

Returns to the calling procedure. Deallocates the current stack frame, changes the FP to point to the stack frame of the calling procedure. Instruction execution is continued at the instruction pointed to by the RIP in the calling procedure's stack frame, which is the instruction immediately following the call instruction.

As shown in the action statement below, the action that the processor takes on the return is determined by the return status and prereturn trace bits. These bits are contained in bits 0-3 of register L0 of the current set of local registers.

Refer to Chapters 6 and 7 for further discussion of the **ret** instruction.

Action:

```
syncf;     -- wait for all possible faults
if pre_return and process_controls.trace_enable then
   pre_return ← 0;
   raise pre-return trace fault;
end if;

-- the frame status determines different return actions
case frame_status is
   when 2#000# =>    -- local return
      FP ← PFP;
      deallocate current register_set;
      if register_set (FP) not allocated then
         retrieve from memory(FP);
      end if;
      IP ← RIP;

   when 2#001# =>    -- fault return
      x ← memory(FP-16);
      y ← memory(FP-12);
      do case 000 action;
      arithmetic_controls ← y;
      if execution_mode = supervisor then
         process_controls ← x;
      end if;
```

```
when 2#010# | 2#011# =>   -- supervisor return
   if execution_mode ≠ supervisor then
      goto case 000;
   else
      process_controls.trace_enable ← frame_status and 2#1#;
      execution_mode ← user;
      do case 000;
      if return or supervisor/subsystem-trace enabled then
         raise return and/or supervisor/subsystem-trace fault;
      end if;
   end if;

when 2#100# | 2#101# =>   -- any subsystem return
   -- pop the top control stack entry
   current_control_stack_pointer ← current_control_stack_pointer - 16;
   control_stack_entry ← read_va(current_environment_table,
         current_control_stack_pointer, quad_mixed);

   -- the topmost_sp is just an approximate value
   if control_stack_entry.return_mode = fault_return then
      -- the size is fault record and resumption record (implementation dependent)
      topmost_sp ← current_fp - 64;
   else
      topmost_sp ← current_fp;
   end if;

   -- cannot trust the return status to determine inter- vs intra-
   -- subsystem return
   if control_stack_entry.return_mode = inter_subsystem or
      control_stack_entry.return_mode = inter_subsystem_fault then
      -- intersubsystem specific actions

      -- save the topmost_fp and topmost_sp
      write_va(current_environment_table, current_subsystem_table_offset,
            qaud_mixed) ← (previous_fp, topmost_sp);

      -- get current subsystem table entry
      current_subsystem_table_offset ← control_stack_entry.subsystem_table_offset;
      subsystem_table_entry ← read_va(current_environment_table,
            current_subsystem_table_offset, quad_mixed);
      previous_fp ← subsystem_table_entry.topmost_fp;
      current_region_2_ad ← subsystem_table_entry.region_2_ad;
      current_event_fault_mask ← subsystem_table_entry.subsystem_id.type_rights_1;

   end if;
```

```
    process_controls.trace_enable ← control_stack_entry.trace_enable;
    current_region_0_ad ← control_stack_entry.region_0_ad;
    current_region_1_ad ← control_stack_entry.region_1_ad;

    do case 000 action;

    -- check trace events
    if return or supervisor/subsystem-trace enabled then
      raise return and/or supervisor/subsystem-trace fault;
    end if;

  when 2#110# =>    -- idle interrupt
    if execution_mode = supervisor then
      free current register set;
      check_pending_interrupts;
      -- if continue here, no interrupt to do
      if processor_controls.state = 2#10# then
        enter idle state;
      end if;
    else
      do case 000 action;
    end if;

  when 2#110# | 2#111# =>    -- interrupt return
    x ← memory(FP-16);
    y ← memory(FP-12);
    do case 000 action;
    arithmetic_controls ← y;
    if execution_mode = supervisor then
      process_controls ← x;
      check_pending_interrupts;
    end if;
end case;
```

Faults:

The following faults are specific to any types of returns:

	MEMORY FAULTS
Pre-return Trace	when the pre-return trace bit is set.
Return Trace	

The following faults are specific to supervisor or subsystem returns:

	Control-stack underflow the return mode is invalid.
Event Fault	when event fault is postponed.
Supervisor/Subsystem Trace	in supervisor/subsystem return.

The following faults are specific to fault returns or subsystem fault returns:

	TRACE FAULTS Trace fault pending in the saved process controls.

See Also:

call, calls, callx

rotate

Mnemonic:

rotate	59D	REG	Rotate

Format:

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

Description:

Rotates the *src* left (toward the more significant bits) by *len* number of bits and stores the result in *dst.* Bits shifted out of the most significant bit position are wrapped around and shift back in from the least significant bit position.

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

Action:

dst ← *src* **rotate** (*len* mod 32)

Faults:

roundr, roundrl

Mnemonic:

roundr	68B	REG	Round Real
roundrl	69B	REG	Round Long Real

Format:

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

Description:

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

For the **roundrl** instruction, if any operand references a general register, two successive registers are used.

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

Action:

dst ← round_to_integral_value (*src1*);

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation

Floating Inexact

saveprcs

Mnemonic:

saveprcs	666	REG	Save Process

Format:

saveprcs

Description:

Updates the state of the current process in memory by saving that part of the process state that is cached on the processor chip during the execution of the process. he part of the process state that is cached includes part of the PCB and any cached local-register frames. The process is not unlocked and continues to execute with its cached state.

The **saveprcs** and **resumprcs** instructions are similar to the **save** and **resume** functions in most UNIX kernels. These instructions allow task (or process) switching without using the processor's automatic dispatching mechanism.

The primary function of the **saveprcs** instruction is to save the state of a process prior to switching processes using the **resumprcs** instruction.

Action:

```
if PRCB.processor_controls.state = process_executing then
   perform process-suspension action
else
   flush any local register sets;
endif;
```

Faults:

MEMORY FAULTS

scaler, scalerl

Mnemonic:

scaler	677	REG	Scale Real
scalerl	676	REG	Scale Long Real

Format:

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

Description:

Multiples *src2* by 2 to the power of *src1* and stores the result in *dst*. If *src1* is negative, the operation becomes a division. The *src1* operand is an integer; whereas, *src2* and *dst* are reals.

For the **scalerl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-N	0	+N
-∞	-∞	-∞	-∞
-F	-F	-F	-F
-0	-0	-0	-0
+0	+0	+0	+0
+F	+F	+F	+F
+∞	+∞	+∞	+∞
NaN	NaN	NaN	NaN

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

In most cases, only the exponent is changed and the mantissa (fraction) remains unchanged. However, when the *src1* is a denormalized value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

Unlike multiply and divide, the scale operation can cause massive underflow and overflow. Refer to Section 5.10.5 and Section 5.10.6 for further details.

Action:

dst ← *src2* * (2^*src1*)

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Zero Divide

Floating Invalid Operation

Floating Inexact

scanbit

Mnemonic:

scanbit	641	REG	Scan For Bit

Format:

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

Description:

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

Action:

```
dst ← 16#FFFFFFFF#;
AC.cc ← 2#000#;
for i in 31..0 reverse loop
   if (src and 2^i) ≠ 0 then
      dst ← i;
      AC.cc ← 2#010#;
      exit;
   end if;
end loop;
```

Faults:

See Also:

spanbit

scanbyte

Mnemonic:

scanbyte	5AC	REG	Scan Byte Equal

Format:

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

Description:

Performs a byte-by-byte comparison of *src1* and *src2* and sets the condition code to 2#010# if any two corresponding bytes are equal. Otherwise, the condition code is set to 000_2.

Action:

```
if (src1 and 16#000000FF#) = (src2 and 16#000000FF#) or
   (src1 and 16#0000FF00#) = (src2 and 16#0000FF00#) or
   (src1 and 16#00FF0000#) = (src2 and 16#00FF0000#) or
   (src1 and 16#FF000000#) = (src2 and 16#FF000000#) then
    AC.cc ← 2#010#;
else
    AC.cc ← 2#000#;
end if;
```

Faults:

Example:

```
# assume g8 = 0x11AB1100
scanbyte 0, g8
# AC.cc ← 2#010#
```

See Also:

cmpstr

schedprcs

Mnemonic:

schedprcs	665	REG	Schedule Process

Format:

schedprcs	*prcs_ad*, reg AD	---,	---

Description:

Sends a process to its dispatching port. The *prcs_ad* specifies the AD of the PCB for the process to be scheduled. The process is enqueued at the head of its priority queue at its dispatching port. If the preempt bit in PCB of the process is set, a preemption action is initiated.

The AD of the dispatching port and the priority of the process are determined from the process's PCB.

Action:

```
if not prcs_ad.tag then
   raise invalid-AD fault;
elsif not prcs_ad.type_rights_3 then
   raise type-rights fault;
elsif object_type_of(prcs_ad) ≠ process_type then
   raise type-mismatch fault;
end if;

perform unblock action on process specified with prcs_ad;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

send

Mnemonic:

send	662	REG	Send

Format:

send	*port_ad*,	*msg_priority*,		*msg_ad*
	reg	reg/lit	reg	
	AD		AD	

Description:

Sends a message to a port. The *msg_ad* contains the AD of the message being sent and the *port_ad* contains the AD of the port the message is to be sent to.

If the port is a priority-type port, the message is handled as follows. If there are processes enqueued at the port, the message is bound to the process at the head of the highest priority queue that has queued processes. The process is then rescheduled at its dispatching port. If there are no processes enqueued at the port, the message is enqueued at the end of the queue of the priority specified in the *msg_priority*. The *msg_priority* can range from 0 to 31.

If the port is a FIFO port, the message is handled in the same way, except that the *msg_priority* is ignored.

When the dequeued process is rescheduled, a preemption action is initiated if the preempt bit in the process object is set and if the process has a higher priority than the currently running process.

Action:

```
if not port_ad.tag or msg_ad.tag then
   raise invalid-AD fault;
elsif not port_ad.type_rights_2 then
   raise type-rights fault;
elsif the queue link record of msg_ad cannot be accessed then
   raise rep-rights or page-rights fault;
elsif msg_ad.local ≠ port_ad.local then
   raise lifetime fault;
elsif object_type_of(port_ad) ≠ port_type then
   raise type-mismatch fault;
end if;

-- loop the port
loop
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   if (port_header.lock_byte and 2#1#) = 2#0#) then exit end if;
   -- wait until the semaphore is unlocked
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;
   delay;
end loop;
```

```
if port_header.queue_state ≠ process then
   -- enqueue message
   if port_header.enq_mode = fifo then
      enqueue msg_ad;
   else
      enqueüe msg_ad at (msg_priority mod 32) priority;
   end if;

   -- unlock the port
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   port_header.lock_byte ← port_header.lock_byte and not 2#1#;
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;

else
   if port_header.enq_mode = fifo then
      dequeue the first process;
   else
      dequeue the first process from the highest priority non-empty queue;
   end if;

   -- unlock the port
   port_header ← atomic_read_ad(port_ad, 0, long_word);
   port_header.lock_byte ← port_header.lock_byte and not 2#1#;
   atomic_write_va(port_ad, 0, word) ← port_header.word_0;

   dequeued_process.received_message ← msg_ad;
   perform unblock action on dequeue process

end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

sendserv

Mnemonic:

sendserv	663	REG	Send Service

Format:

sendserv	*port_ad*, reg AD	---,	---

Description:

Suspends the current process and sends the AD of its process object as a message to the port specified in *port_ad*. The processor selects another process from its dispatching port afterward.

If there are blocked processes at the port, bind the message AD to highest priority process waiting at the port. If the port is a FIFO port, the process AD is queued at the end of the queue. If the port is a priority port, the process AD is queued at the end of the queue corresponding to the current process priority.

Action:

```
perform process suspension action;
perform send operation with
   port_ad as port_ad,
   the current process AD as msg_ad,
   the current process priority as msg_priority,
   and port_ad.type_rights_3 is checked instead of type_rights_2

-- the processor is free
perform dispatch action;
```

Faults:

see *send* instruction

setbit

Mnemonic:

setbit	583	REG	Set Bit

Format:

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

Description:

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

Action:

dst ← *src* or 2^(*bitpos* mod 32);

Faults:

See Also:

alterbit, chkbit, clrbit, notbit,

SHIFT

Mnemonic:

shlo	59C	REG	Shift Left Ordinal
shro	598	REG	Shift Right Ordinal
shli	59E	REG	Shift Left Integer
shri	59B	REG	Shift Right Integer
shrdi	59A	REG	Shift Right Dividing Integer

Format:

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

Description:

Shifts *src* left or right by the number of digits indicated with the *len* operand and stores the result in *dst*. This operation (with the exception of the **shri** instruction) is equivalent to multiplying (shift left) or dividing (shift right) the *src* value by 2^{len}.

The **shri** instruction performs a conventional arithmetic right shift, which rounds the result towards the more negative number. The **shrdi** instruction rounds the result towards zero, as in the **divi**.

Action:

```
shlo:     dst ← src* 2^len;    -- ordinal arithmetic

shro:     dst ←- src/2^len     -- ordinal arithmetic

shli:     dst ← src* 2^len     -- integer arithmetic

shri:
   sign_extension ← (src and 16#80000000#) / 2^len; -- integer arithmetic
   dst ← sign_extension or (src / 2^len);  -- ordinal arithmetic

shrdi:    dst ← src/2^len
```

Faults:

Integer Overflow

signal

Mnemonic:

signal	66A	REG	Signal

Format:

signal	*sem_ad*, reg AD	---,	---

Description:

Unblocks (dequeues) a process from the semaphore queue if there are processes enqueued. If there is no process queued at the semaphore, the semaphore count is incremented by one.

The *sem_ad* contains the AD of the semaphore being signaled.

If a process is dequeued, it is rescheduled at its dispatching port.

Action:

```
if not sem_ad.tag then
   raise invalid-AD fault;
elsif not sem_ad.type_rights_2 then
   raise type-rights fault;
end if;

sem_desc_offset ← sem_ad.objec_index * 16;
sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, quad_mixed);

if sem_desc.entry_type ≠ invalid_descriptor then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
        semd_desc.word_1;
   raise invalid-descriptor fault;
elsif sem_desc.entry_type ≠ embedded_descriptor or
   sem_desc.object_type ≠ semaphore then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
        semd_desc.word_1;
   raise type-mismatch fault;
end if;

while (sem_desc.lock_byte and 2#1#) = 2#1#) loop
   -- wait until the semaphore is unlocked
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ←
        semd_desc.word_0;
   delay;
   sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, long_mixed);
end loop;

if sem_desc.sem_tail = 0 then
   sem_desc.sem_count ← + 1;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;
```

```
else
   sem_desc.lock_byte ← 1;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;

   dequeue first process from the semaphore queue;

   -- unlock the semaphore
   sema_desc ← sematomic_write_va(current_object_table_ad, sem_desc_offset, word);
   -- clear the lock bit
   sema_desc.lock_byte ← sema_desc.lock_byte and not 2#1#;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;

   perform unblock action on dequeued process;
end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

sinr, sinrl

Mnemonics:

sinr	68C	REG	Sine Real
sinrl	69C	REG	Sine Long Real

Format:

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

Description:

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

For the **sinrl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

In the trigonmetic instructions, the processor uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. Section 5.8.6 gives this π value, along with some suggestions for representing this value in a program.

Action:

dst ← sin (*src*);

Faults:

Floating Reserved Encoding	
Floating Underflow	
Floating Invalid Operation	when *src* is ∞.
Floating Inexact	when *src* is not 0.

spanbit

Mnemonic:

spanbit	640	REG	Span Over Bit

Format:

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

Description:

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

Action:

```
dst ← 16#FFFFFFFF#;
AC.cc ← 2#000#;
for i in 31..0 reverse loop
   if (src and 2^i) = 0 then
      dst ← i;
      AC.cc ← 2#010#;
      exit;
   end if;
end loop;
```

Faults:

See Also:

scanbit

sqrtr, sqrtrl

Mnemonic:

sqrtr	688	REG	Square Root Real
sqrtrl	698	REG	Square Root Long Real

Format:

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

Description:

Takes the square root of *src* and stores it in *dst.*

For the **sqrtrl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

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

Action:

dst ← sqrt (*src*);

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation when *src* < 0.

Floating Inexact

STORE

Mnemonic:

st	92	MEM	Store
stob	82	MEM	Store Ordinal Byte
stos	8A	MEM	Store Ordinal Short
stib	C2	MEM	Store Integer Byte
stis	CA	MEM	Store Integer Short
stl	9A	MEM	Store Long
stt	A2	MEM	Store Triple
stq	B2	MEM	Store Quad

Format:

st*	*src,*	*dst*
	reg	mem

Description:

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

The *dst* operand specifies the address of the memory location where the first byte is to be stored. The full range of linear addressing modes may be used in specifying *dst*. (Refer to Section 17.5 for the different addressing modes.)

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

Action:

memory (*dst*) ← *src*;

Faults:

MEMORY FAULTS	
Integer Overflow Fault	for **stib** and **stis** instructions

stm

Mnemonic:

stm	D2	MEM	Store Mixed
stml	DA	MEM	Store Mixed Long
stmq	F2	MEM	Store Mixed Quad

Format:

stm**	*src,*	*dst*
	reg	mem

Description:

Copies the contents, including the tag bits, of a register or group of registers to memory. The *src* operand specifies a register or the first (lowest numbered) register of successive registers.

The *dst* operand specifies the address of the memory location where the first byte of the word (or words) is to be stored. The full range of linear addressing modes may be used in specifying *dst*. (Refer to Section 17.5 for the different addressing modes.)

The **stm** instruction copies 1 words from a register to memory; **stml** copies two words from successive registers to memory; and **stmq** copies four words from successive registers to memroy.

Action:

memory (*dst*) ← *src*;

Faults:

MEMORY FAULTS

Lifetime faults

STORE VIRTUAL

Mnemonic:

stv	93	MEM	Store Virtual
stvob	83	MEM	Store Virtual Ordinal Byte
stvos	8B	MEM	Store Virtual Ordinal Short
stvib	C3	MEM	Store Virtual Integer Byte
stvis	CB	MEM	Store Virtual Integer Short
stvl	9B	MEM	Store Virtual Long
stvt	A3	MEM	Store Virtual Triple
stvq	B3	MEM	Store Virtual Quad

Format:

stv*	*src,*	*dst*
	reg	mem

Description:

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

The *dst* operand specifies the virtual address of the memory location where the first byte is to be stored. The full range of virtual addressing modes may be used in specifying *dst*. (Refer to Section 17.5 for the different addressing modes.)

The **stvob** and **stvib**, and **stvos** and **stvis** instructions store a byte and half word, respectively, from the low order bytes of the *src* register. The **stv**, **stvl**, **stvt**, and **stvq** instructions copy 4, 8, 16, and 32 bytes, respectively, from successive registers to memory.

Action:

memory (*dst*) ← read (*src*);

Faults:

MEMORY FAULTS	
Integer Overflow Fault	for **stvib** and **stvis** instructions

stvm

Mnemonic:

stvm	D3	MEM	Store Virtual Mixed
stvml	DB	MEM	Store Virtual Mixed Long
stvmq	F3	MEM	Store Virtual Mixed Quad

Format:

stvm*	*src,*	*dst*
	reg	mem

Description:

Copies the contents, including the tag bits, of a register or group of registers to memory. The *src* operand specifies a register or the first (lowest numbered) register of successive registers.

The *dst* operand specifies the address of the memory location where the first byte of the word (or words) is to be stored. The full range of virtual addressing modes may be used in specifying *dst.* (Refer to Section 17.5 for different addressing modes.)

The **stvm** instruction copies 1 words from a register to memory; **stvml** copies two words from successive registers to memory; and **stvmq** copies four words from successive registers to memroy.

Action:

memory (*dst*) ← *src*;

Faults:

MEMORY FAULTS

Lifetime faults

subc

Mnemonic:

subc	5B2	REG	Subtract Ordinal With Carry

Format:

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

Description:

Subtracts *src1* and (1 - carry) from the *src2*, and stores the result in *dst*. Bit 1 of the condition code is used as the carry bit, which is the complement of the borrow bit. If the ordinal subtraction results in a carry (or no borrow), bit 1 of the condition code is set; otherwise, bit 1 is cleared. If integer subraction results in an overflow, bit 0 of the condition code is set; otherwise, bit 0 is cleared. Bit 2 of the condition code is always set to zero. Regardless of the result of the subtraction, the condition code is always updated.

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

This instruction never signals an integer overflow fault.

Action:

```
# Let the value of the condition code be xCx.
dst ← src2 - src1 - (1 - C);
AC.cc ← 2#0CV#;
# C is carry (or no borrow) from ordinal subtraction.
# V is 1 if integer subtraction would have generated an overflow.
```

Faults:

Example:

```
  # Example of double-precision arithmetic
  # Assume 64-bit source operands # in g0,g1 and g2,g3
cmpo 0, 0          # set carry bit in AC.cc
subc g0, g2, g0    # add low-order 32 bits;
subc g1, g3, g1    # add high-order 32 bits;
  # 64-bit result is in g0, g1
```

See Also:

addc

subi, subo

Mnemonic:

subi	593	REG	Subtract Integer
subo	592	REG	Subtract Ordinal

Format:

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

Description:

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

Action:

dst ← *src2* - *src1*;

Faults:

Integer Overflow	in **subi**

subr, subrl

Mnemonic:

subr	78D	REG	Subtract Real
subrl	79D	REG	Subtract Long Real

Format:

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

Description:

Subtracts *src1* from *src2* and stores the result in *dst.* For the **subrl** instruction, if any operand references a general register, two successive registers are used.

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

Src1 → Src2 ↓	-∞	-F	-0	+0	+F	+∞	NaN
-∞	INV	-∞	-∞	-∞	-∞	-∞	NaN
-F	+∞	± F or ± 0	src2	src2	-F	-∞	NaN
-0	+∞	src1	± 0	-0	src1	-∞	NaN
+0	+∞	src1	+0	± 0	src1	-∞	NaN
+F	+∞	+F	src2	src2	± F or ± 0	-∞	NaN
+∞	+∞	+∞	+∞	+∞	+∞	INV	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

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

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

Action:

dst ← *src2* - *src1*;

Faults:

Floating Reserved Encoding

Floating Overflow

Floating Underflow

Floating Invalid Operation

Floating Inexact

syncf

Mnemonic:

syncf	66F	REG	Synchronize Faults

Format:

syncf

Description:

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

Action:

```
if not arithmetic_controls.nif then;
   wait until no imprecise faults can occur
   associated with any uncompleted instructions;
end if;
```

Faults:

synld

Mnemonic:

synld	615	REG	Synchronous Load

Format:

synld	*src_addr*, reg addr	---,	*dst* reg

Description:

Copies a word from the memory location specified with *src_addr* into *dst*. The reading of the memory location at *src_addr* is synchronized, where all prior memory operations have completed before the read operation is started. When the load has been successfully completed, the condition code is set to 010_2. Otherwise the condition code is set to 000_2.

The primary function of this instruction is to allow software to avoid bad-access fault on reading storage location which may signal bad-access. Another function of this instruction is to access the internal Interrupt Control Register.

The setting of the condition code indicates whether or not the load was completed successfully. If the load operation results in a bad access condition (e.g., reading an AP-bus interconnect register), the condition code is set to 000_2 and the *dst* is undefined, but the bad-access fault is not raised.

Action:

```
if PRCB.addressing_mode = physical or
   (src_addr.tag and src_adr.ad = 0) then
   syn_addr ← src_addr.offset;
else
   syn_addr ← physical_address (src_addr);
end if;
syn_addr ← syn_addr and not 2#11#;      # force word alignment
if syn_addr = 16#FF000004# then
   dst ← interrupt_control_reg;
   AC.cc ← 2#010#;
else
   wait for all prior memory accesses to finish
   dst ← memory (syn_addr);    -- bad_access fault is disable
   wait for completion;
   if bad_access then
      AC.cc ← 2#000#;
   else
      AC.cc ← 2#010#;
   end if;
end if;
```

Faults:

MEMORY FAULTS

synmov, synmovl, synmovq

Mnemonic:

synmov	600	REG	Synchronous Move
synmovl	601	REG	Synchronous Move Long
synmovq	602	REG	Synchronous Move Quad

Format:

synmov*	*dst_addr*,	---,	*src_addr*
	reg		reg
	addr		addr

Description:

Copies 1 (**synmov**), 2 (**synmovl**), or 4 (**synmovq**) words from the memory location specified with *src_addr* to the memory location specified with *dst_addr*. The writing of the memory location at *src_addr* is synchronized, where all prior memory operations have completed before the write operation is started. When the write operation has been successfully completed, the condition code is set to 010_2. Otherwise, the condition code is set to 000_2.

The *src_addr* and *dst_addr* specify the address of the first byte. These addresses should be for word boundaries (**synmov**), double-word boundaries (**synmovl**), or quad-word boundaries (**synmovq**). Otherwise, the processor forces alignment to these boundaries.

The primary function of these instructions is for sending IAC messages. Another function is to allow software to avoid bad-access fault on writing into storage location which may signal bad-access.

The setting of the condition code indicates whether or not the write operation was completed successfully. If the write operation results in a bad access condition (e.g., sending an IAC message to a busy BXU), the condition code is set to 000_2, but the Bad Access Fault is not raised.

Address $FF000004_{16}$ is used in **synmov** to modify the internal Interrupt Control Register. Address $FF000010_{16}$ is used in **synmovq** to send an IAC message to the processor executing the instruction.

Action:

```
synmov:

case instruction is
   -- length of the memory operand in bytes
   when synmov  => length ← 4;
   when synmovl => length ← 8;
   when synmovq => length ← 16;
end case;
src_value ← memory(src_addr, length)
if PRCB.addressing_mode = physical or
   (src_addr.ad.tag = 1 and src_addr.ad = 0 then
   syn_addr ← dst_addr;
else
   syn_addr ← physical_address (dst_addr);
end if;
← tempa and not (length-1);     -- force alignment
if syn_addr = 16#FF000004# and instruction = synmov then
   interrupt_control_reg ← memory (src_addr)
   AC.cc ← 2#010#;
elsif syn_addr = 16#FF000010# and instruction = synmovq then
   AC.cc ← 2#010#;
   use src_value as a received iac message;
else
   wait for all prior memory accesses to finish
   -- the following write is marked non-cacheable
   memory (syn_addr) ← src_value;  -- bad_access fault is disable
   wait for completion;
   if bad_access then
      AC.cc ← 2#000#;
   else
      AC.cc ← 2#010#;
   end if;
end if;
```

Faults:

MEMORY FAULTS

tanr, tanrl

Mnemonics:

tanr	68E	REG	Tangent Real
tanrl	69E	REG	Tangent Long Real

Format:

tanr*	*src,* freg/flit	---,	*dst* freg

Description:

Computes the tangent of *src* and stores the result in *dst*. The *src* is an angle given in radians. The result is in the range of -∞ to +∞, inclusive.

For the **tanrl** instruction, if any operand references a general register, two successive registers are used.

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

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

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

In the trigonmetic instructions, the processor uses a value for π with a 66-bit mantissa which is 2 bits more than are available in the extended-real format. Section 5.8.6 gives this π value, along with some suggestions for representing this value in a program.

Action:

dst ← tangent (*src*);

Faults:

Floating Reserved Encoding	
Floating Underflow	
Floating Invalid Operation	when *src* is ∞.
Floating Inexact	when *src* is not 0.

TEST

Mnemonic:

teste	22	COBR	Test For Equal
testne	25	COBR	Test For Not Equal
testl	24	COBR	Test For Less
testle	26	COBR	Test For Less or Equal
testg	21	COBR	Test For Greater
testge	23	COBR	Test For Greater or Equal
testo	27	COBR	Test For Ordered
testno	20	COBR	Test For Unordered

Format:

test*	---,	---,	*dst* reg

Description:

Stores a 1 (for true) in *dst* if the logical AND of the condition code and the mask-part of the opcode is not zero, or if the condition code equals the mask-part of the opcode. Otherwise, the instruction stores a 0 (for false) in *dst.*

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

Instruction	Mask	Condition
testno	000	Unordered
testg	001	Greater
teste	010	Equal
testge	011	Greater or equal
testl	100	Less
testne	101	Not equal
testle	110	Less or equal
testo	111	Ordered

Action:

```
if ((mask and AC.cc) ≠ 2#000#) or (mask #=# AC.cc) then
   dst ← 1; # dst set for true
else
   dst ← 0; # dst set for false
end if;
```

Faults:

wait

Mnemonic:

wait	669	REG	Wait

Format:

wait	*sem_ad,* reg AD	---,	---

Description:

Waits on the semaphore. The *sem_ad* contains the AD of the semaphore.

If the queue tail is non-zero or the count is zero, the process is suspended and enqueued on the semaphore. Otherwise, the count is decremented by one and execution of the process continues.

The process remains queued on the semaphore until it is dequeued by a **signal** instruction. The process is then dequeued and rescheduled at its dispatching port.

Action:

```
if not sem_ad.tag then
   raise invalid-AD fault;
elsif not sem_ad.type_rights_2 then
   raise type-rights fault;
end if;

sem_desc_offset ← sem_ad.objec_index * 16;
sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, quad_mixed);

if sem_desc.entry_type ≠ invalid_descriptor then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
         semd_desc.word_1;
   raise invalid-descriptor fault;
elsif sem_desc.entry_type ≠ embedded_descriptor or
   sem_desc.object_type ≠ semaphore then
   atomic_write_va(current_object_table_ad, sem_desc_offset + 4, mixed) ←
         semd_desc.word_1;
   raise type-mismatch fault;
end if;

while (sem_desc.lock_byte and 2#1#) = 2#1#) loop
   -- wait until the semaphore is unlocked
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ←
         semd_desc.word_0;
   delay;
   sem_desc ← atomic_read_ad(current_object_table_ad, sem_desc_offset, long_mixed);
end loop;
```

```
if sem_desc.sem_tail ≠ 0 or else sem_desc.count = 0 then
   -- lock the semaphore
   sem_desc.lock_byte ← 1;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;

    suspend current process
    enqueue current process on the semaphore queue;

   -- unlock the semaphore
   sema_desc ← sematomic_write_va(current_object_table_ad, sem_desc_offset, word);
   -- clear the lock bit

   -- processor is free now
   perform dispatch action;

else
   sem_desc.sem_count ← - 1;
   atomic_write_va(current_object_table_ad, sem_desc_offset, word) ← semd_desc.word_0;
end if;
```

Faults:

Invalid AD

Type Rights

Type Mismatch

MEMORY FAULTS

xnor, xor

Mnemonic:

xnor	589	REG	Exclusive Nor
xor	586	REG	Exclusive Or

Format:

xnor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg
xor	*src1*,	*src2*,	*dst*
	reg/lit	reg/lit	reg

Description:

Performs a bitwise XNOR (**xnor** instruction) or XOR (**xor** instruction) operation on the *src2* and *src1* and stores the result in *dst.*

Action:

xnor: *dst* ← **not** (*src2* **or** *src1*) **or** (*src2* **and** *src1*);

xor: *dst* ← (*src2* **or** *src1*) **and** (**not** (*src2* **and** *src1*));

Faults:

See Also:

LOGCIAL

INSTRUCTION AND DATA STRUCTURE QUICK REFERENCE A

This appendix provides quick reference for the instructions and data structures.

A.1 Instruction Quick Reference

This section provides two lists of instructions: one sorted by assembly-language mnemonic and another sorted by machine-level opcode. In these lists, each entry includes the assembly-language mnemonic for an instruction; the operands (given in the required order); the machine-level opcode and instruction type (that is, REG, MEM, COBR, or CTRL); and the page number in Chapter 18 where the detailed description of the instruction is given.

A.1.1 Instruction List by Assembler Mnemonic

Mnemonic	Operands			Opcode	Inst. Type	Page
addc	*src1,*	*src2,*	*dst*	5B0	REG	18-7
addi	*src1,*	*src2,*	*dst*	591	REG	18-8
addo	*src1,*	*src2,*	*dst*	590	REG	18-8
addr	*src1,*	*src2,*	*dst*	78F	REG	18-9
addrl	*src1,*	*src2,*	*dst*	79F	REG	18-9
alterbit	*bitpos,*	*src,*	*dst*	58F	REG	18-10
amplify	*src1,*	*src2,*	*src/dst*	653	REG	18-11
and	*src1,*	*src2,*	*dst*	581	REG	18-12
andnot	*src1,*	*src2,*	*dst*	582	REG	18-12
atadd	*src/dst,*	*src,*	*dst*	612	REG	18-13
atanr	*src1,*	*src2,*	*dst*	680	REG	18-14
atanrl	*src1,*	*src2,*	*dst*	690	REG	18-14
atmod	*src,*	*mask,*	*src/dst*	610	REG	18-15
atrep	*src/dst,*	*src,*	*dst*	611	REG	18-16
b	*targ*			08	CTRL	18-18
bal	*targ*			0B	CTRL	18-17
balx	*targ,*	*dst*		85	MEM	18-17
bbc	*bitpos,*	*src,*	*targ*	30	COBR	18-19
bbs	*bitpos,*	*src,*	*targ*	37	COBR	18-19
be	*targ*			12	CTRL	18-21
bg	*targ*			11	CTRL	18-21
bge	*targ*			13	CTRL	18-21
bl	*targ*			14	CTRL	18-21
ble	*targ*			16	CTRL	18-21
bne	*targ*			15	CTRL	18-21
bno	*targ*			10	CTRL	18-21
bo	*targ*			11	CTRL	18-21
bx	*targ*			84	MEM	18-18
call	*targ*			09	CTRL	18-22
calld	*src1,*	*src2*		661	REG	18-23
calls	*targ*			660	REG	18-28
callx	*targ*			86	MEM	18-29
chkbit	*bitpos,*	*src*		5AE	REG	18-31
chktag	*src*	5A8			REG	18-32
classr	*src*			68F	REG	18-33
classrl	*src*			69F	REG	18-33
clrbit	*bitpos,*	*src,*	*dst*	58C	REG	18-35
cmpdeci	*src1,*	*src2,*	*dst*	5A7	REG	18-37
cmpdeco	*src1,*	*src2,*	*dst*	5A6	REG	18-37
cmpi	*src1,*	*src2*		5A1	REG	18-36
cmpibe	*src1,*	*src2,*	*targ*	3A	COBR	18-40

Mnemonic	Operands			Opcode	Inst. Type	Page
cmpibg	*src1,*	*src2,*	*targ*	39	COBR	18-40
cmpibge	*src1,*	*src2,*	*targ*	3B	COBR	18-40
cmpibl	*src1,*	*src2,*	*targ*	3C	COBR	18-40
cmpible	*src1,*	*src2,*	*targ*	3E	COBR	18-40
cmpibne	*src1,*	*src2,*	*targ*	3D	COBR	18-40
cmpibno	*src1,*	*src2,*	*targ*	38	COBR	18-40
cmpibo	*src1,*	*src2,*	*targ*	3F	COBR	18-40
cmpinci	*src1,*	*src2,*	*dst*	5A5	REG	18-38
cmpinco	*src1,*	*src2,*	*dst*	5A4	REG	18-38
cmpm	*src1,*	*src2*		5AA	REG	18-39
cmpo	*src1,*	*src2*		5A0	REG	18-36
cmpobe	*src1,*	*src2,*	*targ*	32	COBR	18-40
cmpobg	*src1,*	*src2,*	*targ*	31	COBR	18-40
cmpobge	*src1,*	*src2,*	*targ*	33	COBR	18-40
cmpobl	*src1,*	*src2,*	*targ*	34	COBR	18-40
cmpoble	*src1,*	*src2,*	*targ*	36	COBR	18-40
cmpobne	*src1,*	*src2,*	*targ*	35	COBR	18-40
cmpor	*src1,*	*src2*		684	REG	18-42
cmporl	*src1,*	*src2*		694	REG	18-42
cmpr	*src1,*	*src2*		685	REG	18-43
cmprl	*src1,*	*src2*		695	REG	18-43
cmpstr	*src1,*	*src2,*	*len*	603	REG	18-44
concmpi	*src1,*	*src2*		5A3	REG	18-45
concmpo	*src1,*	*src2*		5A2	REG	18-45
condrec	*src,*	*dst*		646	REG	18-46
condwait	*src*			668	REG	18-48
cosr	*src,*	*dst*		68D	REG	18-50
cosrl	*src,*	*dst*		69D	REG	18-50
cpyrsre	*src1,*	*src2,*	*dst*	6E3	REG	18-51
cpysre	*src1,*	*src2,*	*dst*	6E2	REG	18-51
cread	*src1,*	*src2,*	*dst*	648	REG	18-52
cvtadr	*src,*	*dst*		672	REG	18-53
cvtilr	*src,*	*dst*		675	REG	18-54
cvtir	*src,*	*dst*		674	REG	18-54
cvtri	*src,*	*dst*		6C0	REG	18-55
cvtril	*src,*	*dst*		6C1	REG	18-55
cvtzri	*src,*	*dst*		6C2	REG	18-55
cvtzril	*src,*	*dst*		6C3	REG	18-55
daddc	*src1,*	*src2,*	*dst*	642	REG	18-56
divi	*src1,*	*src2,*	*dst*	74B	REG	18-57
divo	*src1,*	*src2,*	*dst*	70B	REG	18-57
divr	*src1,*	*src2,*	*dst*	78B	REG	18-58

Mnemonic	Operands			Opcode	Inst. Type	Page
divrl	*src1,*	*src2,*	*dst*	79B	REG	18-58
dmovt	*src,*	*dst*		644	REG	18-59
dsubc	*src1,*	*src2,*	*dst*	643	REG	18-60
ediv	*src1,*	*src2,*	*dst*	671	REG	18-61
emul	*src1,*	*src2,*	*dst*	670	REG	18-62
expr	*src,*	*dst*		689	REG	18-63
exprl	*src,*	*dst*		699	REG	18-63
extract	*bitpos,*	*len,*	*src/dst*	651	REG	18-65
faulte				1A	CTRL	18-66
faultg				19	CTRL	18-66
faultge				1B	CTRL	18-66
faultl				1C	CTRL	18-66
faultle				1E	CTRL	18-66
faultne				1D	CTRL	18-66
faultno				18	CTRL	18-66
faulto				1F	CTRL	18-66
fill	*dst*	*value*	*len*	617	REG	18-67
flushreg				66D	REG	18-68
fmark				66C	REG	18-69
inspacc	*src*		*dst*	613	REG	18-70
ld	*src,*	*dst*		90	MEM	18-71
lda	*src*	*dst*		8C	MEM	18-75
ldcsp	*dst*			657	REG	18-76
ldglobals	*src,*	*dst*		64A	REG	18-77
ldib	*src,*	*dst*		C0	MEM	18-71
ldis	*src,*	*dst*		C8	MEM	18-71
ldl	*src,*	*dst*		98	MEM	18-71
ldm	*src,*	*dst*		D0	MEM	18-72
ldml	*src,*	*dst*		D8	MEM	18-72
ldmq	*src,*	*dst*		F0	MEM	18-72
ldob	*src,*	*dst*		80	MEM	18-71
ldos	*src,*	*dst*		88	MEM	18-71
ldphy	*src,*	*dst*		614	REG	18-78
ldq	*src,*	*dst*		B0	MEM	18-71
ldt	*src,*	*dst*		A0	MEM	18-71
ldtime	*dst*			673	REG	18-79
ldtypdef	*src,*	*dst*		649	REG	18-80
ldv	*src,*	*dst*		91	MEM	18-73
ldvib	*src,*	*dst*		C1	MEM	18-73
ldvis	*src,*	*dst*		C9	MEM	18-73
ldvl	*src,*	*dst*		99	MEM	18-73
ldvm	*src,*	*dst*		D1	MEM	18-72

Mnemonic	Operands			Opcode	Inst. Type	Page
ldvml	*src,*	*dst*		D9	MEM	18-72
ldvmq	*src,*	*dst*		F1	MEM	18-72
ldvob	*src,*	*dst*		81	MEM	18-73
ldvos	*src,*	*dst*		89	MEM	18-73
ldvq	*src,*	*dst*		B1	MEM	18-73
ldvt	*src,*	*dst*		A1	MEM	18-73
logbnr	*src,*	*dst*		68A	REG	18-81
logbnrl	*src,*	*dst*		69A	REG	18-81
logepr	*src1,*	*src2,*	*dst*	681	REG	18-82
logeprl	*src1,*	*src2,*	*dst*	691	REG	18-82
logr	*src1,*	*src2,*	*dst*	682	REG	18-84
logrl	*src1,*	*src2,*	*dst*	692	REG	18-84
mark				66B	REG	18-86
modac	*mask,*	*src,*	*dst*	645	REG	18-87
modi	*src1,*	*src2,*	*dst*	749	REG	18-88
modify	*mask,*	*src,*	*src/dst*	650	REG	18-89
modpc	*src*	*mask,*	*src/dst*	655	REG	18-90
modtc	*mask,*	*src,*	*dst*	654	REG	18-91
mov	*src,*	*dst*		5CC	REG	18-92
movl	*src,*	*dst*		5DC	REG	18-92
movm	*src,*	*dst*		5CD	REG	18-93
movml	*src,*	*dst*		5DD	REG	18-93
movmq	*src,*	*dst*		5FD	REG	18-93
movq	*src,*	*dst*		5FC	REG	18-92
movqstr	*dst,*	*src,*	*len*	604	REG	18-94
movr	*src,*	*dst*		6C9	REG	18-95
movre	*src,*	*dst*		6E9	REG	18-95
movrl	*src,*	*dst*		6D9	REG	18-95
movstr	*dst,*	*src,*	*len*	605	REG	18-97
movt	*src,*	*dst*		5EC	REG	18-92
muli	*src1,*	*src2,*	*dst*	741	REG	18-98
mulo	*src1,*	*src2,*	*dst*	701	REG	18-98
mulr	*src1,*	*src2,*	*dst*	78C	REG	18-99
mulrl	*src1,*	*src2,*	*dst*	79C	REG	18-99
nand	*src1,*	*src2,*	*dst*	58E	REG	18-100
nor	*src1,*	*src2,*	*dst*	588	REG	18-101
not	*src,*	*dst*		58A	REG	18-102
notand	*src,*	*dst*		584	REG	18-102
notbit	*bitpos,*	*src,*	*dst*	580	REG	18-103
notor	*src1,*	*src2,*	*dst*	58D	REG	18-104
or	*src1,*	*src2,*	*dst*	587	REG	18-105
ornot	*src1,*	*src2,*	*dst*	58B	REG	18-105

Mnemonic	Operands			Opcode	Inst. Type	Page
receive	*src,*	*dst*		656	REG	18-106
remi	*src1,*	*src2,*	*dst*	748	REG	18-108
remo	*src1,*	*src2,*	*dst*	708	REG	18-108
remr	*src1,*	*src2,*	*dst*	683	REG	18-109
remrl	*src1,*	*src2,*	*dst*	693	REG	18-109
restrict	*src,*	*src/dst*		652	REG	18-111
resumprcs	*src*			664	REG	18-112
ret				0A	CTRL	18-113
rotate	*len,*	*src,*	*dst*	59D	REG	18-117
roundr	*src,*	*dst*		68B	REG	18-118
roundrl	*src,*	*dst*		69B	REG	18-118
saveprcs				666	REG	18-119
scaler	*src1,*	*src2,*	*dst*	677	REG	18-120
scalerl	*src1,*	*src2,*	*dst*	676	REG	18-120
scanbit	*src,*	*dst*		641	REG	18-122
scanbyte	*src1,*	*src2*		5AC	REG	18-123
schedprcs	*src*			665	REG	18-124
send	*dst,*	*src1,*	*src2*	662	REG	18-125
sendserv	*src*			663	REG	18-127
setbit	*bitpos,*	*src,*	*dst*	583	REG	18-128
shli	*len,*	*src,*	*dst*	59E	REG	18-129
shlo	*len,*	*src,*	*dst*	59C	REG	18-129
shrdi	*len,*	*src,*	*dst*	59A	REG	18-129
shri	*len,*	*src,*	*dst*	59B	REG	18-129
shro	*len,*	*src,*	*dst*	598	REG	18-129
signal	*dst*			66A	REG	18-130
sinr	*src,*	*dst*		68C	REG	18-132
sinrl	*src,*	*dst*		69C	REG	18-132
spanbit	*src,*	*dst*		640	REG	18-133
sqrtr	*src,*	*dst*		688	REG	18-134
sqrtrl	*src,*	*dst*		698	REG	18-134
st	*src,*	*dst*		92	MEM	18-135
stib	*src,*	*dst*		C2	MEM	18-135
stis	*src,*	*dst*		CA	MEM	18-135
stl	*src,*	*dst*		9A	MEM	18-135
stm	*src,*	*dst*		D2	MEM	18-136
stml	*src,*	*dst*		DA	MEM	18-136
stmq	*src,*	*dst*		F2	MEM	18-136
stob	*src,*	*dst*		82	MEM	18-135
stos	*src,*	*dst*		8A	MEM	18-135
stq	*src,*	*dst*		B2	MEM	18-135
stt	*src,*	*dst*		A2	MEM	18-135

Mnemonic	Operands			Opcode	Inst. Type	Page
stv	*src,*	*dst*		93	MEM	18-137
stvib	*src,*	*dst*		C3	MEM	18-137
stvis	*src,*	*dst*		CB	MEM	18-137
stvl	*src,*	*dst*		9B	MEM	18-137
stvm	*src,*	*dst*		D3	MEM	18-136
stvml	*src,*	*dst*		DB	MEM	18-136
stvmq	*src,*	*dst*		F3	MEM	18-136
stvob	*src,*	*dst*		83	MEM	18-137
stvos	*src,*	*dst*		8B	MEM	18-137
stvq	*src,*	*dst*		B3	MEM	18-137
stvt	*src,*	*dst*		A3	MEM	18-137
subc	*src1,*	*src2,*	*dst*	5B2	REG	18-139
subi	*src1,*	*src2,*	*dst*	593	REG	18-140
subo	*src1,*	*src2,*	*dst*	592	REG	18-140
subr	*src1,*	*src2,*	*dst*	78D	REG	18-141
subrl	*src1,*	*src2,*	*dst*	79D	REG	18-141
syncf				66F	REG	18-142
synld	*src,*	*dst*		615	REG	18-143
synmov	*dst,*	*src*		600	REG	18-144
synmovl	*dst,*	*src*		601	REG	18-144
synmovq	*dst,*	*src*		602	REG	18-144
tanr	*src,*	*dst*		68E	REG	18-146
tanrl	*src,*	*dst*		69E	REG	18-146
teste	*dst*			22	COBR	18-147
testg	*dst*			21	COBR	18-147
testge	*dst*			23	COBR	18-147
testl	*dst*			24	COBR	18-147
testle	*dst*			26	COBR	18-147
testne	*dst*			25	COBR	18-147
testno	*dst*			20	COBR	18-147
testo	*dst*			27	COBR	18-147
wait	*src*			669	REG	18-148
xnor	*src1,*	*src2,*	*dst*	589	REG	18-150
xor	*src1,*	*src2,*	*dst*	586	REG	18-150

A.1.2 Instruction List by Opcode

Opcode	Inst. Type	Mnemonic	Operands			Page
8	CTRL	**b**	*targ*			18-18
0B	CTRL	**bal**	*targ*			18-17
10	CTRL	**bno**	*targ*			18-21
11	CTRL	**bg**	*targ*			18-21
12	CTRL	**be**	*targ*			18-21
13	CTRL	**bge**	*targ*			18-21
15	CTRL	**bne**	*targ*			18-21
16	CTRL	**ble**	*targ*			18-21
17	CTRL	**bo**	*targ*			18-21
18	CTRL	**faultno**				18-66
19	CTRL	**faultg**				18-66
1A	CTRL	**faulte**				18-66
1B	CTRL	**faultge**				18-66
1C	CTRL	**faultl**				18-66
1D	CTRL	**faultne**				18-66
1E	CTRL	**faultle**				18-66
1F	CTRL	**faulto**				18-66
20	COBR	**testno**	*dst*			18-147
21	COBR	**testg**	*dst*			18-147
22	COBR	**teste**	*dst*			18-147
23	COBR	**testge**	*dst*			18-147
24	COBR	**testl**	*dst*			18-147
25	COBR	**testne**	*dst*			18-147
26	COBR	**testle**	*dst*			18-147
27	COBR	**testo**	*dst*			18-147
30	COBR	**bbc**	*bitpos,*	*src,*	*targ*	18-19
31	COBR	**cmpobg**	*src1,*	*src2,*	*targ*	18-18
32	COBR	**cmpobe**	*src1,*	*src2,*	*targ*	18-40
33	COBR	**cmpobge**	*src1,*	*src2,*	*targ*	18-40
34	COBR	**cmpobl**	*src1,*	*src2,*	*targ*	18-40
35	COBR	**cmpobne**	*src1,*	*src2,*	*targ*	18-40
36	COBR	**cmpoble**	*src1,*	*src2,*	*targ*	18-40
37	COBR	**bbs**	*bitpos,*	*src,*	*targ*	18-19
38	COBR	**cmpibno**	*src1,*	*src2,*	*targ*	18-40
39	COBR	**cmpibg**	*src1,*	*src2,*	*targ*	18-40
3A	COBR	**cmpibe**	*src1,*	*src2,*	*targ*	18-40
3B	COBR	**cmpibge**	*src1,*	*src2,*	*targ*	18-40
3C	COBR	**cmpibl**	*src1,*	*src2,*	*targ*	18-40
3D	COBR	**cmpibne**	*src1,*	*src2,*	*targ*	18-40
3E	COBR	**cmpible**	*src1,*	*src2,*	*targ*	18-40
3F	COBR	**cmpibo**	*src1,*	*src2,*	*targ*	18-40

Opcode	Inst. Type	Mnemonic	Operands		Page
80	MEM	ldob	*src,*	*dst*	18-71
81	MEM	ldvob	*src,*	*dst*	18-73
83	MEM	stvob	*src,*	*dst*	18-137
85	MEM	balx	*targ,*	*dst*	18-17
86	MEM	callx	*targ*		18-29
88	MEM	ldos	*src,*	*dst*	18-71
89	MEM	ldvos	*src,*	*dst*	18-73
8A	MEM	stos	*src,*	*dst*	18-135
8B	MEM	stvos	*src,*	*dst*	18-137
8C	MEM	lda	*src*	*dst*	18-75
90	MEM	ld	*src,*	*dst*	18-71
91	MEM	ldv	*src,*	*dst*	18-73
92	MEM	st	*src,*	*dst*	18-135
93	MEM	stv	*src,*	*dst*	18-137
98	MEM	ldl	*src,*	*dst*	18-71
99	MEM	ldvl	*src,*	*dst*	18-73
9A	MEM	stl	*src,*	*dst*	18-135
9B	MEM	stvl	*src,*	*dst*	18-137
A0	MEM	ldt	*src,*	*dst*	18-71
A1	MEM	ldvt	*src,*	*dst*	18-73
A2	MEM	stt	*src,*	*dst*	18-135
A3	MEM	stvt	*src,*	*dst*	18-137
B0	MEM	ldq	*src,*	*dst*	18-71
B1	MEM	ldvq	*src,*	*dst*	18-73
B2	MEM	stq	*src,*	*dst*	18-135
B3	MEM	stvq	*src,*	*dst*	18-137
C0	MEM	ldib	*src,*	*dst*	18-71
C1	MEM	ldvib	*src,*	*dst*	18-73
C2	MEM	stib	*src,*	*dst*	18-135
C3	MEM	stvib	*src,*	*dst*	18-137
C8	MEM	ldis	*src,*	*dst*	18-71
C9	MEM	ldvis	*src,*	*dst*	18-73
CA	MEM	stis	*src,*	*dst*	18-135
CB	MEM	stvis	*src,*	*dst*	18-137
D0	MEM	ldm	*src,*	*dst*	18-72
D1	MEM	ldvm	*src,*	*dst*	18-72
D2	MEM	stm	*src,*	*dst*	18-136
D3	MEM	stvm	*src,*	*dst*	18-136
D8	MEM	ldml	*src,*	*dst*	18-72
D9	MEM	ldvml	*src,*	*dst*	18-72
DA	MEM	stml	*src,*	*dst*	18-136
DB	MEM	stvml	*src,*	*dst*	18-136

Opcode	Inst. Type	Mnemonic	Operands			Page
F0	MEM	**ldmq**	*src,*	*dst*		18-72
F1	MEM	**ldvmq**	*src,*	*dst*		18-72
F2	MEM	**stmq**	*src,*	*dst*		18-136
F3	MEM	**stvmq**	*src,*	*dst*		18-136
580	REG	**notbit**	*bitpos,*	*src,*	*dst*	18-103
581	REG	**and**	*src1,*	*src2,*	*dst*	18-12
582	REG	**andnot**	*src1,*	*src2,*	*dst*	18-12
583	REG	**setbit**	*bitpos,*	*src,*	*dst*	18-128
584	REG	**notand**	*src,*	*dst*		18-102
586	REG	**xor**	*src1,*	*src2,*	*dst*	18-150
587	REG	**or**	*src1,*	*src2,*	*dst*	18-105
588	REG	**nor**	*src1,*	*src2,*	*dst*	18-101
589	REG	**xnor**	*src1,*	*src2,*	*dst*	18-150
58A	REG	**not**	*src,*	*dst*		18-102
58B	REG	**ornot**	*src1,*	*src2,*	*dst*	18-105
58C	REG	**clrbit**	*bitpos,*	*src,*	*dst*	18-35
58D	REG	**notor**	*src1,*	*src2,*	*dst*	18-104
58E	REG	**nand**	*src1,*	*src2,*	*dst*	18-100
58F	REG	**alterbit**	*bitpos,*	*src,*	*dst*	18-10
590	REG	**addo**	*src1,*	*src2,*	*dst*	18-8
591	REG	**addi**	*src1,*	*src2,*	*dst*	18-8
592	REG	**subo**	*src1,*	*src2,*	*dst*	18-140
593	REG	**subi**	*src1,*	*src2,*	*dst*	18-140
598	REG	**shro**	*len,*	*src,*	*dst*	18-129
59A	REG	**shrdi**	*len,*	*src,*	*dst*	18-129
59B	REG	**shri**	*len,*	*src,*	*dst*	18-129
59C	REG	**shlo**	*len,*	*src,*	*dst*	18-129
59D	REG	**rotate**	*len,*	*src,*	*dst*	18-117
59E	REG	**shli**	*len,*	*src,*	*dst*	18-129
5A0	REG	**cmpo**	*src1,*	*src2*		18-36
5A1	REG	**cmpi**	*src1,*	*src2*		18-36
5A2	REG	**concmpo**	*src1,*	*src2*		18-45
5A3	REG	**concmpi**	*src1,*	*src2*		18-45
5A4	REG	**cmpinco**	*src1,*	*src2,*	*dst*	18-38
5A5	REG	**cmpinci**	*src1,*	*src2,*	*dst*	18-38
5A6	REG	**cmpdeco**	*src1,*	*src2,*	*dst*	18-37
5A7	REG	**cmpdeci**	*src1,*	*src2,*	*dst*	18-37
5A8	REG	**chktag**	*src*			18-32
5AA	REG	**cmpm**	*src1,*	*src2*		18-39
5AC	REG	**scanbyte**	*src1,*	*src2*		18-123
5AE	REG	**chkbit**	*bitpos,*	*src*		18-31
5B0	REG	**addc**	*src1,*	*src2,*	*dst*	18-7

Opcode	Inst. Type	Mnemonic	Operands			Page
5B2	REG	**subc**	*src1,*	*src2,*	*dst*	18-139
5CC	REG	**mov**	*src,*	*dst*		18-92
5CD	REG	**movm**	*src,*	*dst*		18-93
5DC	REG	**movl**	*src,*	*dst*		18-92
5DD	REG	**movml**	*src,*	*dst*		18-93
5EC	REG	**movt**	*src,*	*dst*		18-92
5FC	REG	**movq**	*src,*	*dst*		18-92
5FD	REG	**movmq**	*src,*	*dst*		18-93
600	REG	**synmov**	*dst,*	*src*		18-144
601	REG	**synmovl**	*dst,*	*src*		18-144
602	REG	**synmovq**	*dst,*	*src*		18-144
603	REG	**cmpstr**	*src1,*	*src2,*	*len*	18-44
604	REG	**movqstr**	*dst,*	*src,*	*len*	18-94
605	REG	**movstr**	*dst,*	*src,*	*len*	18-97
610	REG	**atmod**	*src,*	*mask,*	*src/dst*	18-15
611	REG	**atrep**	*src/dst,*	*src,*	*dst*	18-16
612	REG	**atadd**	*src/dst,*	*src,*	*dst*	18-13
613	REG	**inspacc**	*src*		*dst*	18-70
614	REG	**ldphy**	*src,*	*dst*		18-78
615	REG	**synld**	*src,*	*dst*		18-143
617	REG	**fill**	*dst*	*value*	*len*	18-67
640	REG	**spanbit**	*src,*	*dst*		18-133
641	REG	**scanbit**	*src,*	*dst*		18-122
642	REG	**daddc**	*src1,*	*src2,*	*dst*	18-56
643	REG	**dsubc**	*src1,*	*src2,*	*dst*	18-60
644	REG	**dmovt**	*src,*	*dst*		18-59
645	REG	**modac**	*mask,*	*src,*	*dst*	18-87
646	REG	**condrec**	*src,*	*dst*		18-46
648	REG	**cread**	*src1,*	*src2,*	*dst*	18-52
649	REG	**ldtypdef**	*src,*	*dst*		18-80
64A	REG	**ldglobals**	*src,*	*dst*		18-77
650	REG	**modify**	*mask,*	*src,*	*src/dst*	18-89
651	REG	**extract**	*bitpos,*	*len,*	*src/dst*	18-65
652	REG	**restrict**	*src,*	*src/dst*		18-111
653	REG	**amplify**	*src1,*	*src2,*	*src/dst*	18-11
654	REG	**modtc**	*mask,*	*src,*	*dst*	18-91
655	REG	**modpc**	*mask,*	*src/dst*		18-90
656	REG	**receive**	*src,*	*dst*		18-106
657	REG	**ldcsp**	*dst*			18-76
660	REG	**calls**	*targ*			18-28
661	REG	**calld**	*src1,*	*src2*		18-23
662	REG	**send**	*dst,*	*src1,*	*src2*	18-125

Opcode	Inst. Type	Mnemonic	Operands			Page
663	REG	**sendserv**	*src*			18-127
664	REG	**resumprcs**	*src*			18-112
665	REG	**schedprcs**	*src*			18-124
666	REG	**saveprcs**				18-119
668	REG	**condwait**	*src*			18-48
669	REG	**wait**	*src*			18-148
66A	REG	**signal**	*dst*			18-130
66B	REG	**mark**				18-86
66C	REG	**fmark**				18-69
66D	REG	**flushreg**				18-68
66F	REG	**syncf**				18-142
670	REG	**emul**	*src1,*	*src2,*	*dst*	18-62
671	REG	**ediv**	*src1,*	*src2,*	*dst*	18-61
672	REG	**cvtadr**	*src,*	*dst*		18-53
673	REG	**ldtime**	*dst*			18-79
674	REG	**cvtir**	*src,*	*dst*		18-54
675	REG	**cvtilr**	*src,*	*dst*		18-54
676	REG	**scalerl**	*src1,*	*src2,*	*dst*	18-120
677	REG	**scaler**	*src1,*	*src2,*	*dst*	18-120
680	REG	**atanr**	*src1,*	*src2,*	*dst*	18-14
681	REG	**logepr**	*src1,*	*src2,*	*dst*	18-82
682	REG	**logr**	*src1,*	*src2,*	*dst*	18-84
683	REG	**remr**	*src1,*	*src2,*	*dst*	18-109
684	REG	**cmpor**	*src1,*	*src2*		18-42
685	REG	**cmpr**	*src1,*	*src2*		18-43
688	REG	**sqrtr**	*src,*	*dst*		18-134
689	REG	**expr**	*src,*	*dst*		18-63
68A	REG	**logbnr**	*src,*	*dst*		18-81
68B	REG	**roundr**	*src,*	*dst*		18-118
68C	REG	**sinr**	*src,*	*dst*		18-132
68D	REG	**cosr**	*src,*	*dst*		18-50
68E	REG	**tanr**	*src,*	*dst*		18-146
68F	REG	**classr**	*src*			18-33
690	REG	**atanrl**	*src1,*	*src2,*	*dst*	18-14
691	REG	**logeprl**	*src1,*	*src2,*	*dst*	18-82
692	REG	**logrl**	*src1,*	*src2,*	*dst*	18-84
693	REG	**remrl**	*src1,*	*src2,*	*dst*	18-109
694	REG	**cmporl**	*src1,*	*src2*		18-42
695	REG	**cmprl**	*src1,*	*src2*		18-43
698	REG	**sqrtrl**	*src,*	*dst*		18-134
699	REG	**exprl**	*src,*	*dst*		18-63
69A	REG	**logbnrl**	*src,*	*dst*		18-81

Opcode	Inst. Type	Mnemonic	Operands			Page
69B	REG	**roundrl**	*src,*	*dst*		18-118
69C	REG	**sinrl**	*src,*	*dst*		18-132
69D	REG	**cosrl**	*src,*	*dst*		18-50
69E	REG	**tanrl**	*src,*	*dst*		18-146
69F	REG	**classrl**	*src*			18-33
6C0	REG	**cvtri**	*src,*	*dst*		18-55
6C1	REG	**cvtril**	*src,*	*dst*		18-55
6C2	REG	**cvtzri**	*src,*	*dst*		18-55
6C3	REG	**cvtzril**	*src,*	*dst*		18-55
6C9	REG	**movr**	*src,*	*dst*		18-95
6D9	REG	**movrl**	*src,*	*dst*		18-95
6E2	REG	**cpysre**	*src1,*	*src2,*	*dst*	18-51
6E3	REG	**cpyrsre**	*src1,*	*src2,*	*dst*	18-51
6E9	REG	**movre**	*src,*	*dst*		18-95
701	REG	**mulo**	*src1,*	*src2,*	*dst*	18-98
708	REG	**remo**	*src1,*	*src2,*	*dst*	18-108
70B	REG	**divo**	*src1,*	*src2,*	*dst*	18-57
741	REG	**muli**	*src1,*	*src2,*	*dst*	18-98
748	REG	**remi**	*src1,*	*src2,*	*dst*	18-108
74B	REG	**divi**	*src1,*	*src2,*	*dst*	18-57
78B	REG	**divr**	*src1,*	*src2,*	*dst*	18-58
78C	REG	**mulr**	*src1,*	*src2,*	*dst*	18-99
78D	REG	**subr**	*src1,*	*src2,*	*dst*	18-141
78F	REG	**addr**	*src1,*	*src2,*	*dst*	18-9
79B	REG	**divrl**	*src1,*	*src2,*	*dst*	18-58
79C	REG	**mulrl**	*src1,*	*src2,*	*dst*	18-99
79D	REG	**subrl**	*src1,*	*src2,*	*dst*	18-141
79F	REG	**addrl**	*src1,*	*src2,*	*dst*	18-9

A.2 Summary of System Data Structures

The following pages provide a collection of the system data structures presented in this manual. They are are grouped by function. The chapter reference below each data structure shows where in this manual this data structure is described.

A.2.1 Execution Environment

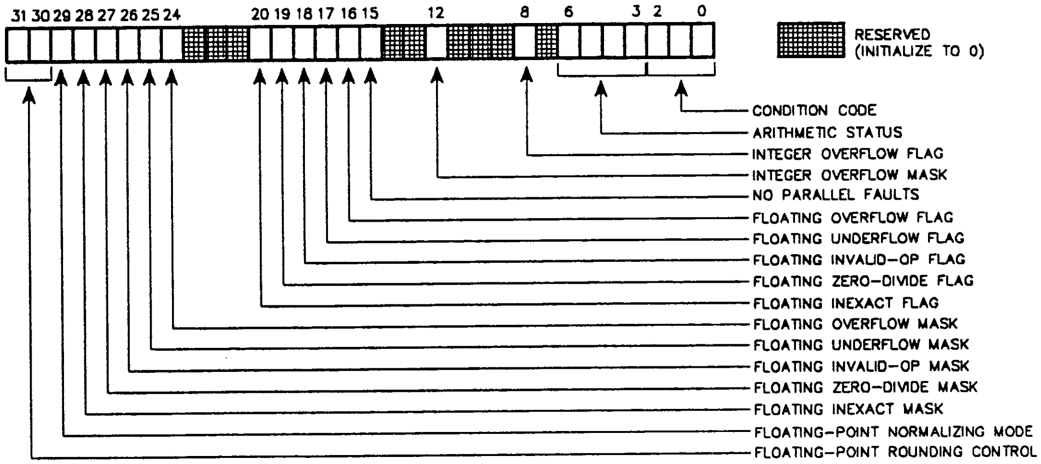


Figure A-1. Arithmetic Controls (Chapter 6)

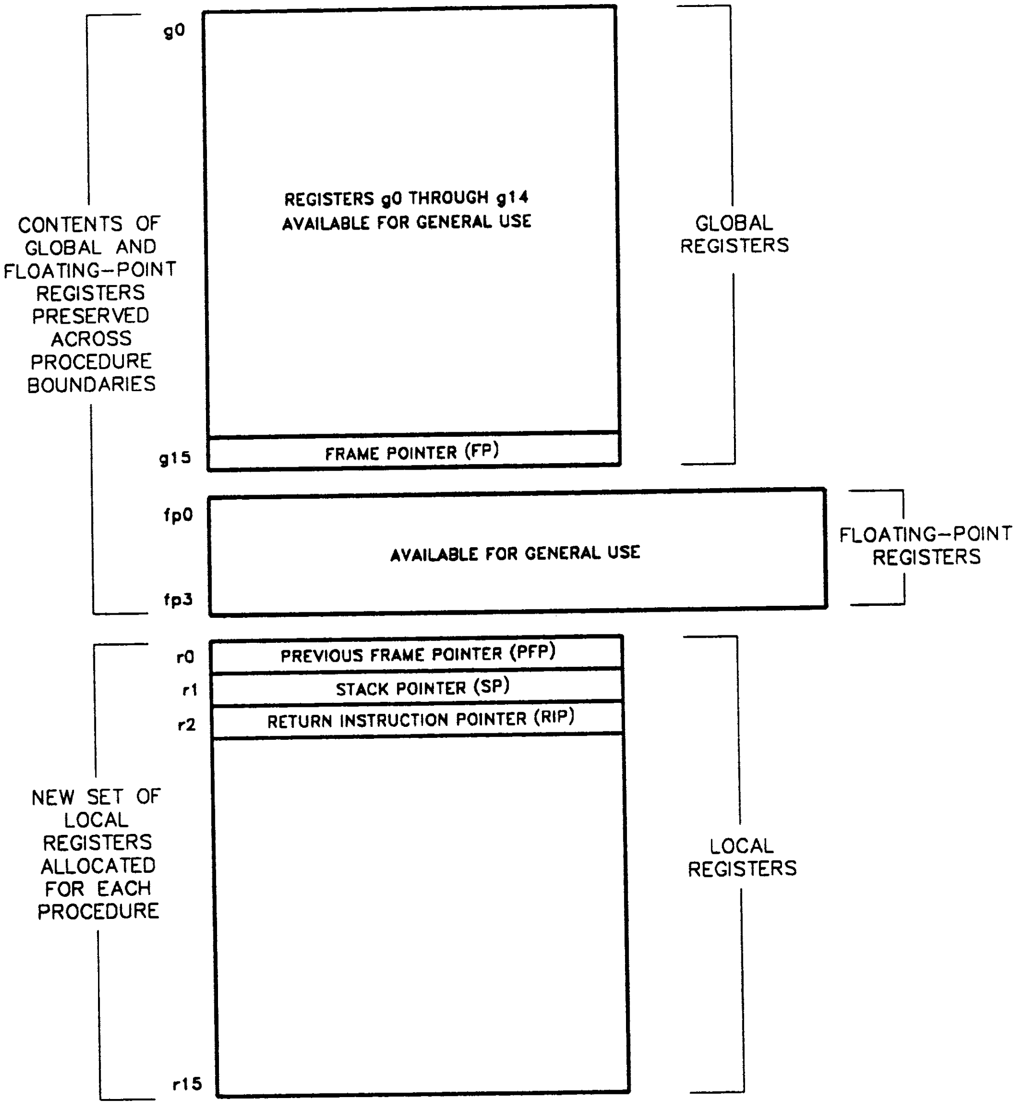


Figure A-2. Registers Available to a Single Procedure (Chapter 6)

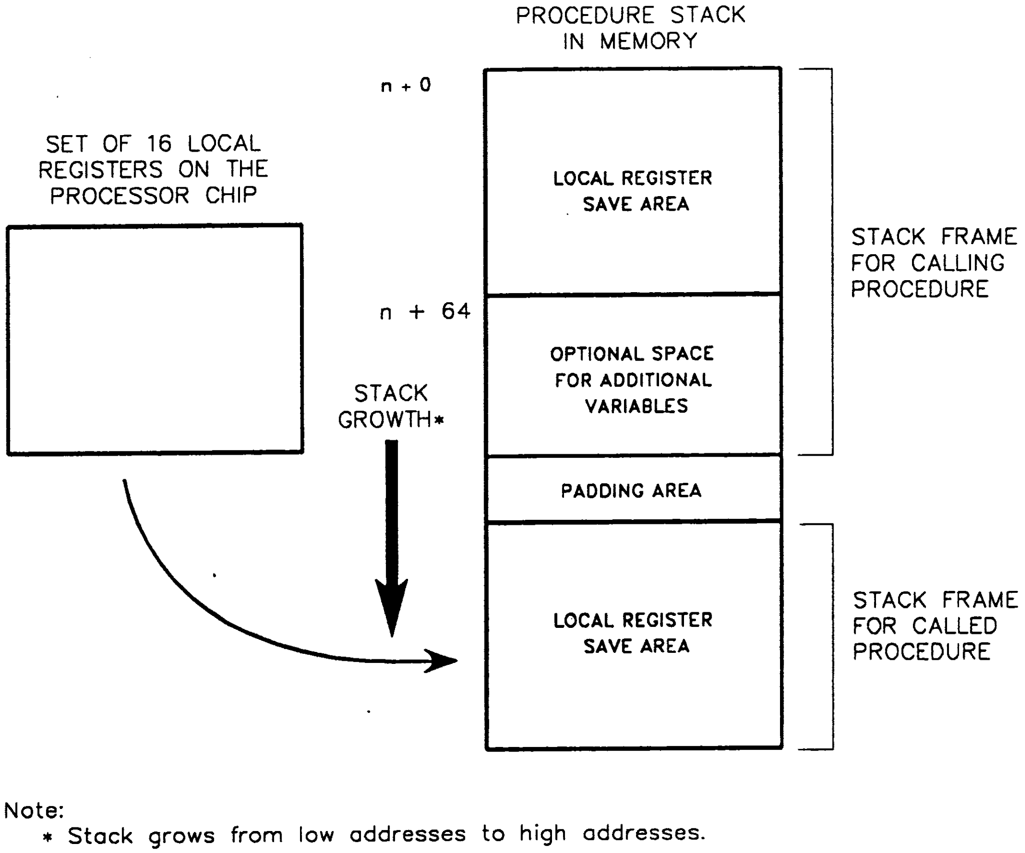


Figure A-3. Procedure Stack Structure (Chapter 18.3)

A.2.2 Memory Management

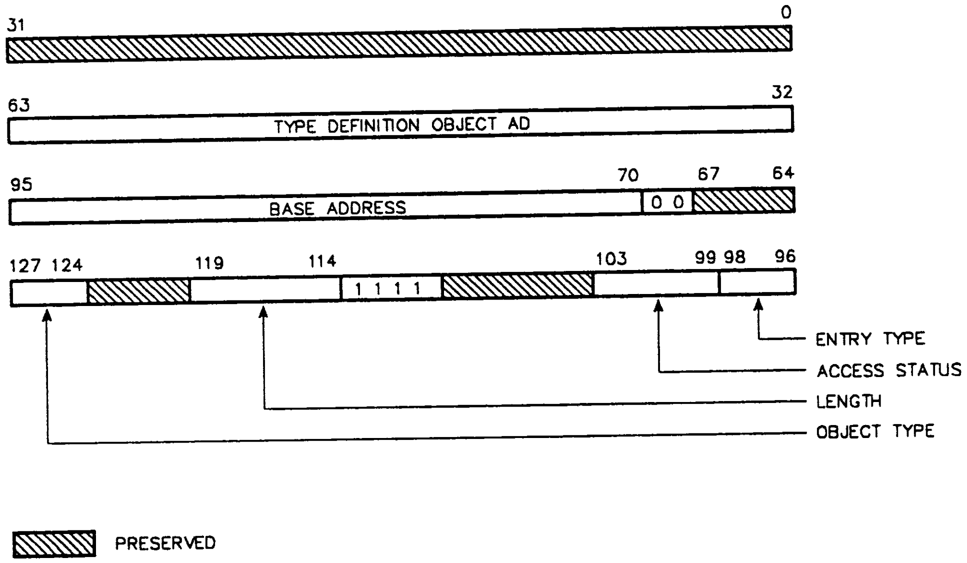
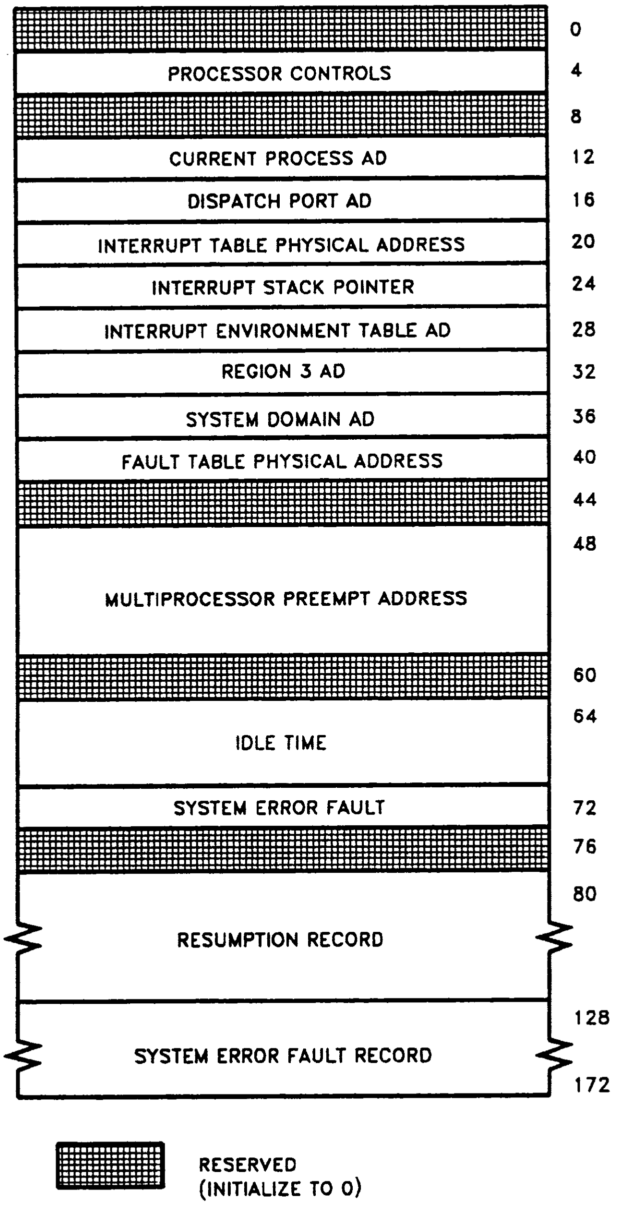


Figure A-4. Object Descriptor Structure (Chapter 8)

A.2.3 Processor Management



▨	0
PROCESSOR CONTROLS	4
▨	8
CURRENT PROCESS AD	12
DISPATCH PORT AD	16
INTERRUPT TABLE PHYSICAL ADDRESS	20
INTERRUPT STACK POINTER	24
INTERRUPT ENVIRONMENT TABLE AD	28
REGION 3 AD	32
SYSTEM DOMAIN AD	36
FAULT TABLE PHYSICAL ADDRESS	40
▨	44
MULTIPROCESSOR PREEMPT ADDRESS	48
▨	60
IDLE TIME	64
SYSTEM ERROR FAULT	72
▨	76
RESUMPTION RECORD	80
SYSTEM ERROR FAULT RECORD	128
	172

▨ RESERVED (INITIALIZE TO 0)

Figure A-5. PRCB (Chapter 16)

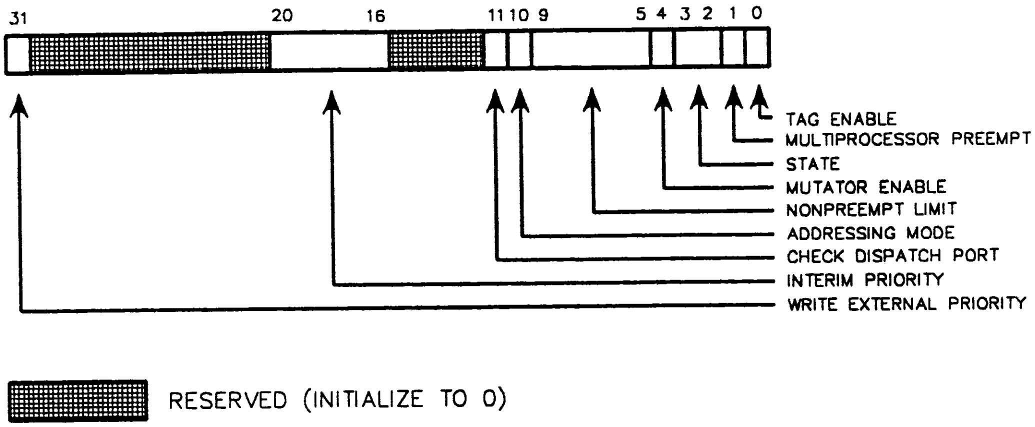


Figure A-6. Processor Controls (Chapter 16)

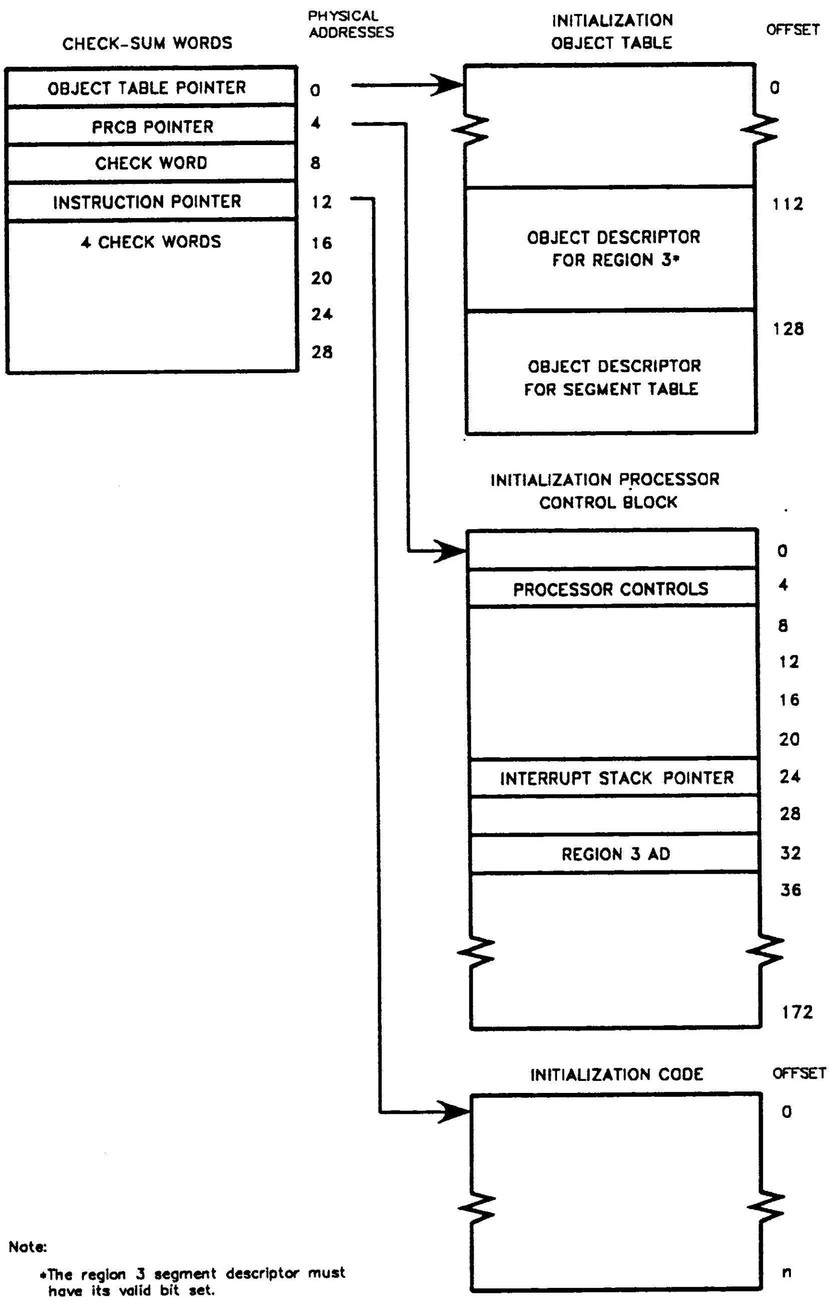


Figure A-7. Initial Memory Image (Chapter 16)

A.2.4 Interrupt Handling

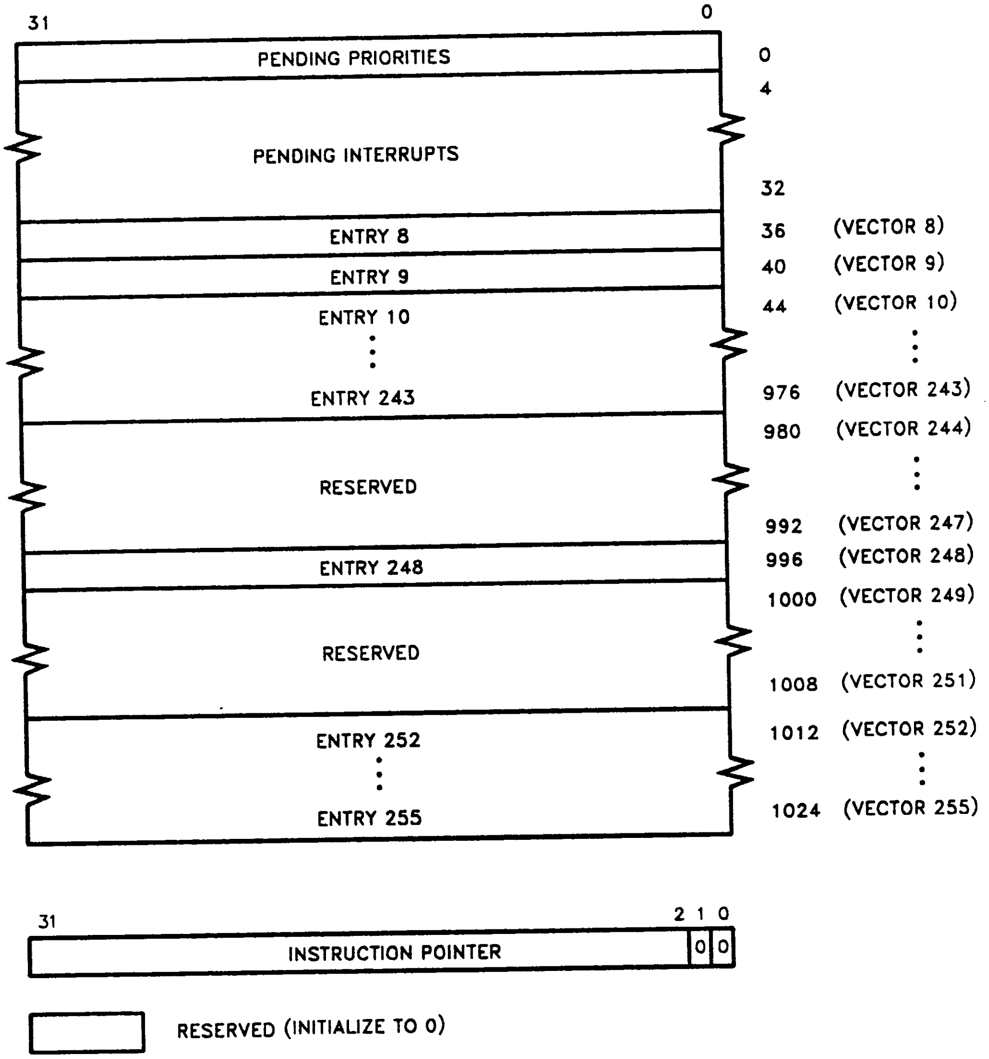
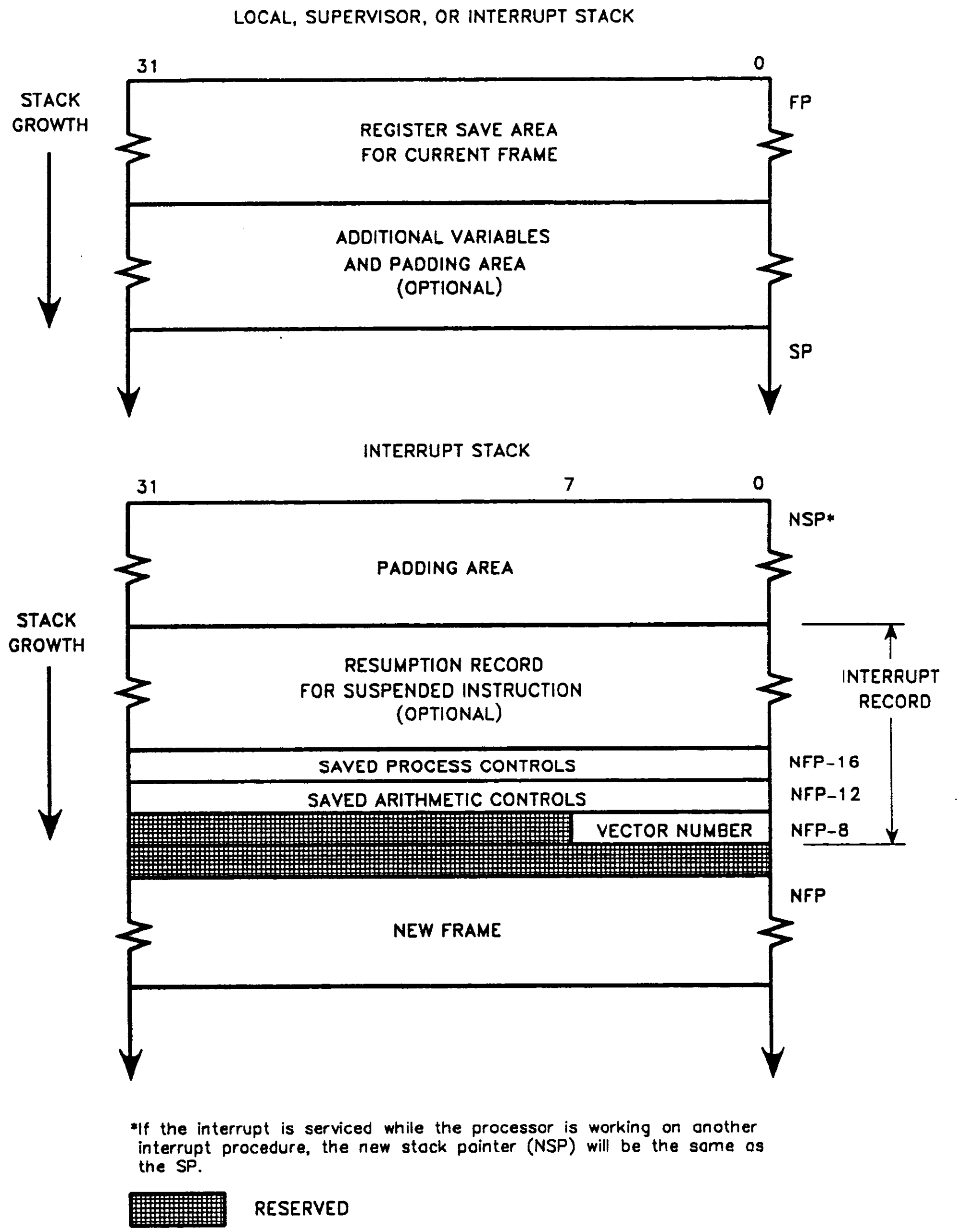


Figure A-8. Interrupt Table (Chapter 12)


LOCAL, SUPERVISOR, OR INTERRUPT STACK
31
0
STACK GROWTH
FP
REGISTER SAVE AREA FOR CURRENT FRAME
ADDITIONAL VARIABLES AND PADDING AREA (OPTIONAL)
SP
INTERRUPT STACK
31
7
0
NSP*
PADDING AREA
STACK GROWTH
RESUMPTION RECORD FOR SUSPENDED INSTRUCTION (OPTIONAL)
INTERRUPT RECORD
SAVED PROCESS CONTROLS
NFP-16
SAVED ARITHMETIC CONTROLS
NFP-12
VECTOR NUMBER
NFP-8
NFP
NEW FRAME
RESERVED

*If the interrupt is serviced while the processor is working on another interrupt procedure, the new stack pointer (NSP) will be the same as the SP.

Figure A-9. Interrupt Record on Stack (Chapter 12)

A.2.5 IACs

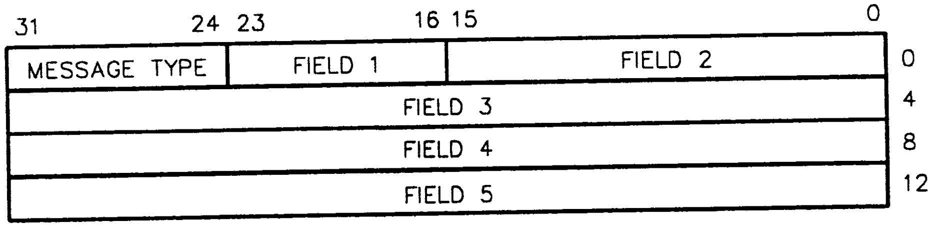


Figure A-10. IAC Message Format (Chapter 10)

A.2.6 Fault Handling

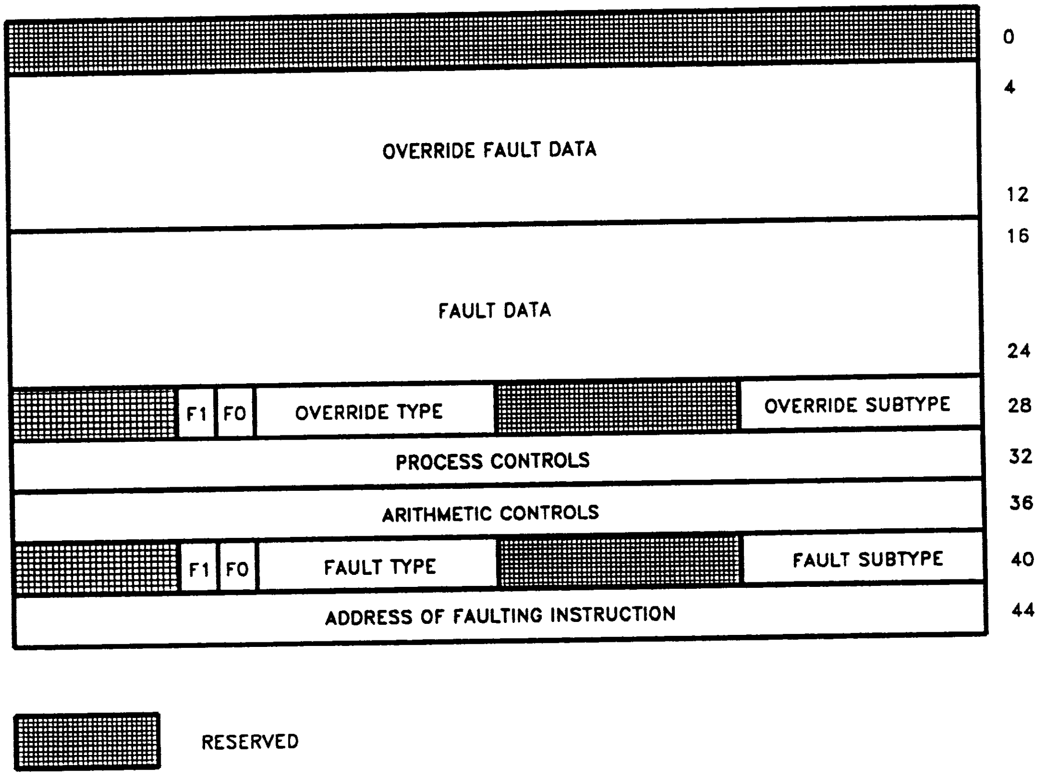


Figure A-11. Fault Record (Chapter 10)

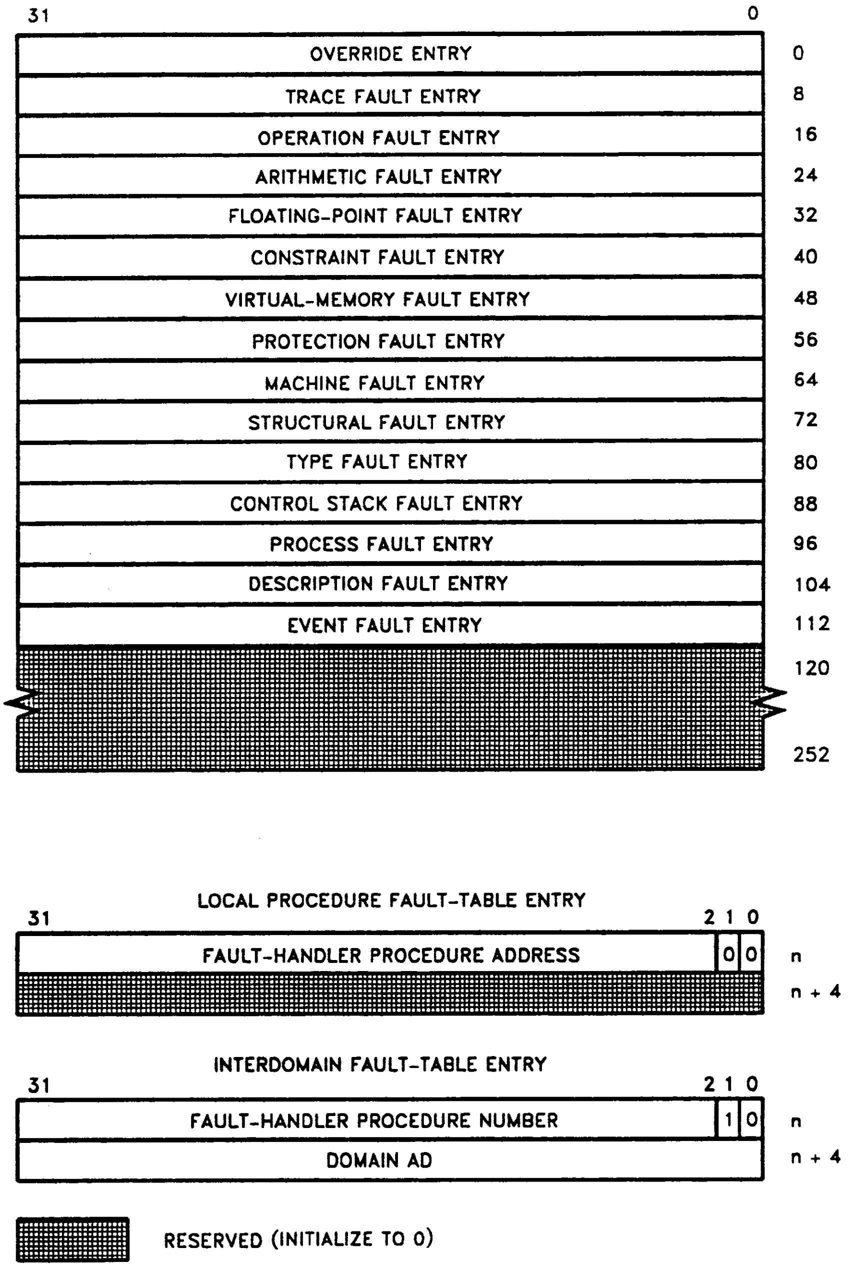


Figure A-12. Fault Table and Fault-Table Entries (Chapter 10)

A.2.7 Process Management

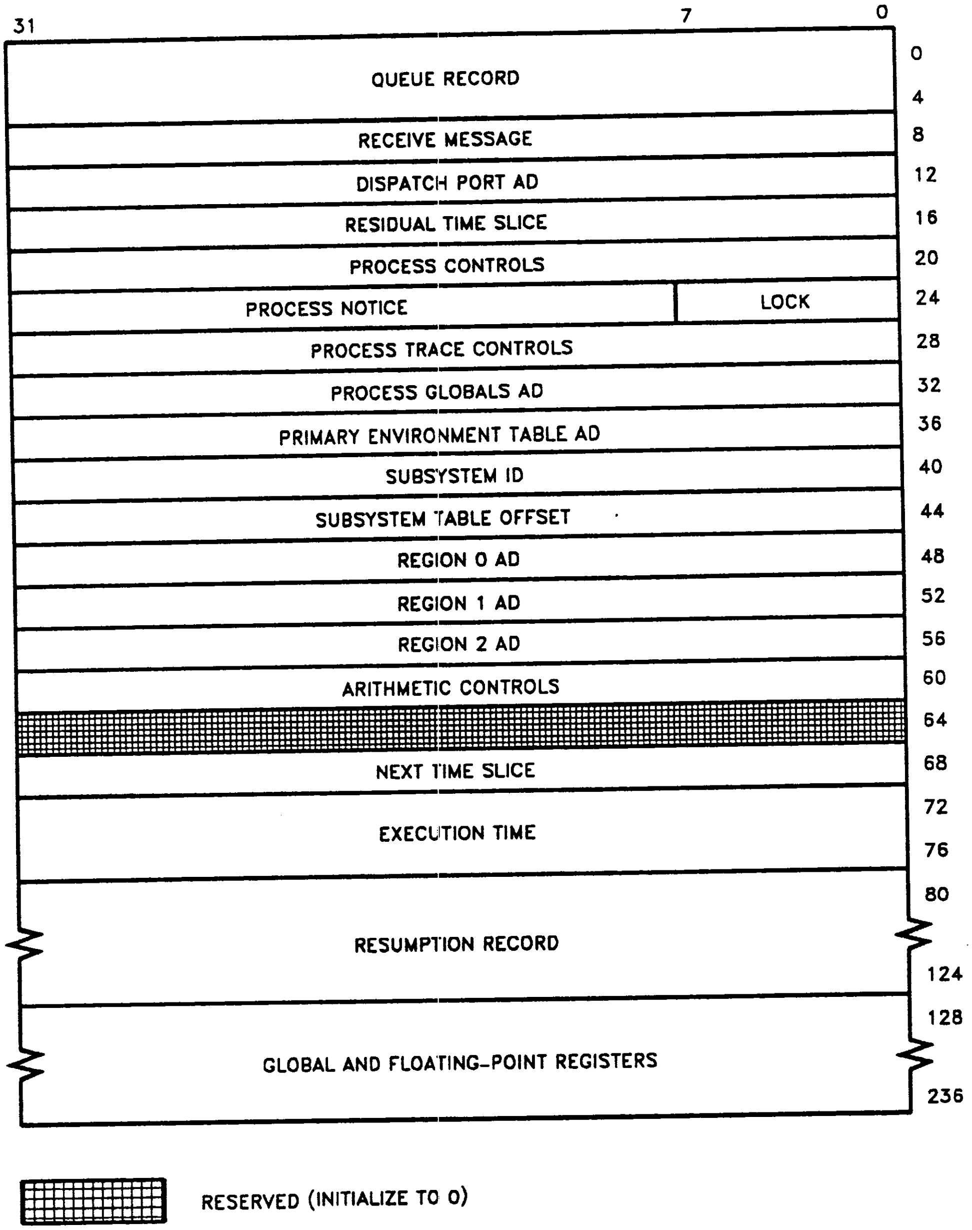


Figure A-13. PCB (Chapter 15)

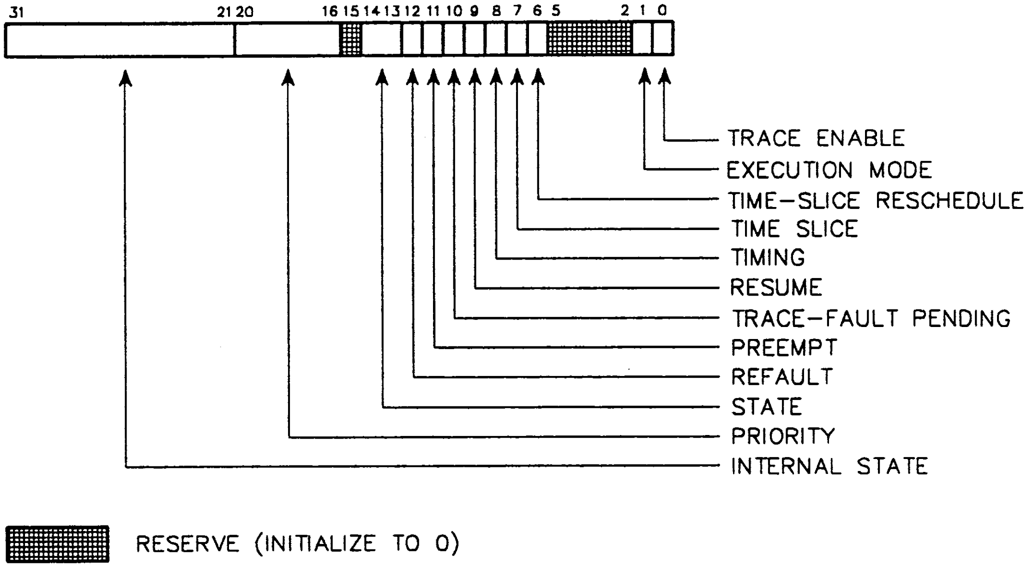


Figure A-14. Process Controls (Chapter 15)

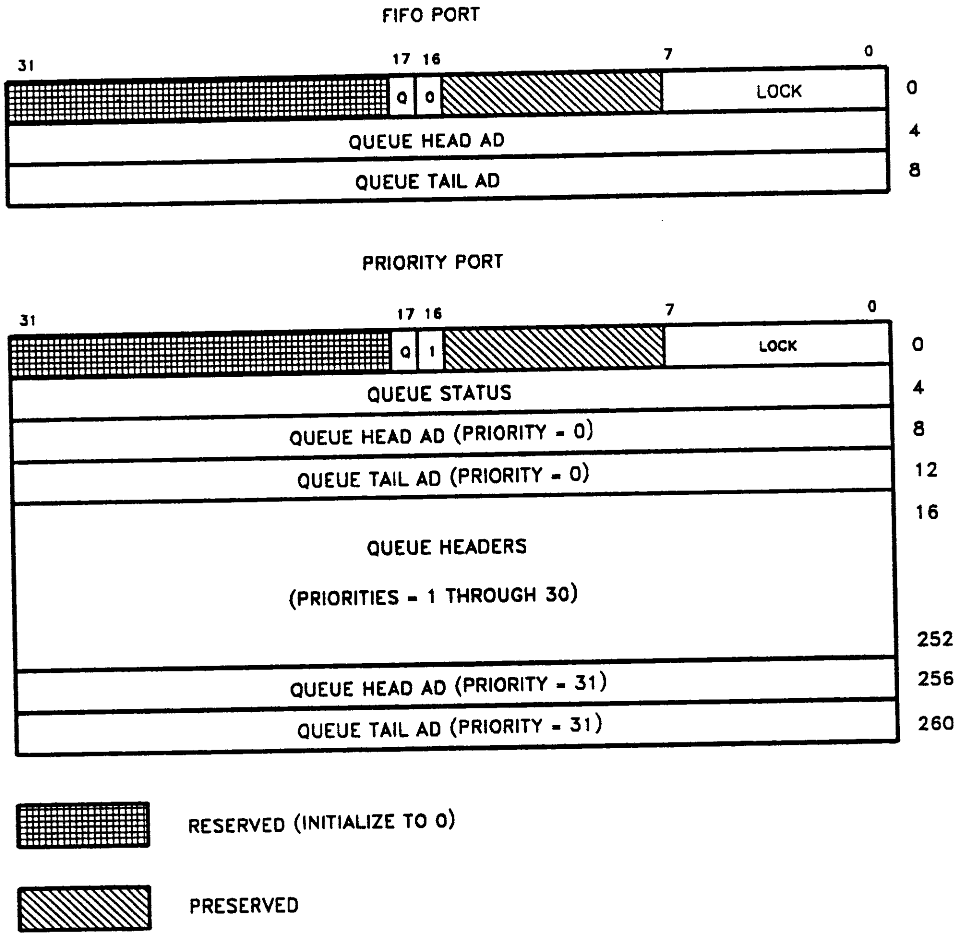

FIFO PORT
31
17 16
7
0
Q
0
LOCK
0
QUEUE HEAD AD
4
QUEUE TAIL AD
8
PRIORITY PORT
31
17 16
7
0
Q
1
LOCK
0
QUEUE STATUS
4
QUEUE HEAD AD (PRIORITY - 0)
8
QUEUE TAIL AD (PRIORITY - 0)
12
16
QUEUE HEADERS
(PRIORITIES - 1 THROUGH 30)
252
QUEUE HEAD AD (PRIORITY - 31)
256
QUEUE TAIL AD (PRIORITY - 31)
260
RESERVED (INITIALIZE TO 0)
PRESERVED

Figure A-15. Ports (Chapter 14)

A.2.8 Trace Control

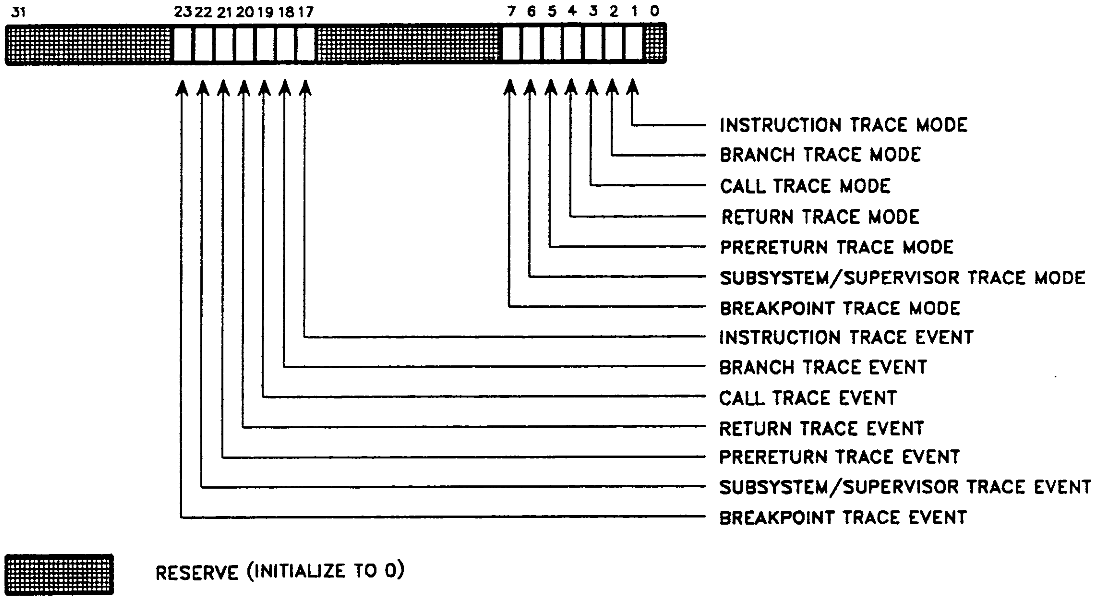


Figure A-16. Trace Controls (Chapter 11)

CONSIDERATIONS FOR WRITING PORTABLE SOFTWARE B

This appendix describes those parts of the processor design that are implementation dependent. This information is provided to facilitate the design of programs and kernel code that will be portable to other similar processors.

B.1 Architecture Restrictions

The following operational aspects are notable considerations:

1. Only the low-order 16 bits of the next-time-slice and residual-time-slice fields in the process object are used. The upper 16 bits are ignored.
2. The minimum value that can be placed in the next-time-slice field is 16 (ticks). Assigning it a value less than 16 can result in endless loops.
3. No length check is performed for port objects during implicit references (such as, dispatching) and when executing the port instructions. FIFO ports are assumed to be at least 12 bytes in length; priority ports are assumed to be at least 264 bytes in length.
4. When the addressing mode is set to physical, the **inspacc** and **ldphy** instructions have an undefined effect when the address operand is a linear address, and the subsystem call and return operations have an undefined effect.
5. On all bus write operations except those of the **synmov**, **synmovl**, and **synmovq** instructions, the processor ignores the BADAC pin (that is, errors signaled on "normal" writes are ignored).
6. The check for out-of-range input values for the **expr**, **exprl**, **logepr**, and **logeprl** instructions is omitted; out-of-range inputs yield an undefined result.
7. Bits 5 and 6 of a machine-level instruction word in the REG and MEMB formats and bits 0 and 1 of the CTRL format are provided to designate special function registers. The processor has no special function registers.
8. Local registers 10, 11, and 12 have their tag bits forced to 0. Thus, moving an AD into any of these registers cause the AD to be converted into a general data word.
9. The processor does not guarantee that the value in register 12 of the current frame is predictable.
10. Simple objects that are used as regions need not be 4K-bytes in length and need not start on a page boundary.
11. When using the REG-format instructions, the m bit for every operand that is not defined by the instruction should be set (that is, code the unused operand as an arbitrary literal). This practice may reduce overhead in some situations.

B.2 Boundary Alignment

The physical-address boundaries on which an operand begins has an impact on processor performance. For the processor, the following is true:

- An operand that spans more word boundaries than necessary (that is, addressing a 32-bit operand on a nonword boundary) suffers a moderate cost in speed because of extra bus and memory cycles.
- An operand that spans a 16-byte boundary suffers a large cost in speed.
- String operands that begin on nonword boundaries suffer a moderate cost in speed. String operands that begin on word boundaries but not on 16-byte boundaries suffer a small cost in speed.

B.3 Faults

As described in Chapter 10, the processor enters the stopped state when a fault is detected while trying to invoke a procedure as the result of a system-error interrupt. When the processor enters the stopped state in this circumstance, it asserts the $\overline{\text{FAILURE}}$ pin.

The size of resumption records conditionally placed on the stack during faults and interrupts is 16 bytes.

B.4 Physical Memory

The upper 16M bytes of physical memory are reserved for special functions of local-bus components, IACs, and the BXU.

B.5 IACs

The mechanism for sending, receiving, and handling IAC messages and the write-external-priority flag is a special implementation of the processor, and may not necessarily be implemented in compatible ways on other similar processors.

B.6 Timing

A tick is defined for as 256 external clock periods (128 internal clock periods). Thus, for a 16-MHz processor (32-MHz external clock), a tick is 8 microseconds. For a 20-MHz processor, a tick is 6.4 microseconds.

The frequency at which an idle processor checks the dispatch port is approximately once every tick.

The frequency at which a processor updates the idle-time field in the processor controls when it is counting idle time is approximately once every 32 ticks.

When the processor is spinning on a lock (that is, when executing a **send**, **receive**, or **signal** instruction), the frequency at which the processor tries the lock is once every tick until it is able to lock it. Provided that the execution timer and end-of-time-slice event are enabled, the process may eventually be suspended. When redispatched, it will resume execution within the instruction and the locking operation will be retried. In the other circumstances where a processor needs to lock a data structure and it is already locked, it will try the lock approximately once every tick until it can lock the data structure.

B.7 Interrupts

The interrupt IAC message, the interrupt pins, and the interrupt register are special implementations for the processor.

B.8 Initialization

The initialization mechanism and procedures described in this manual are specific to the processor.

B.9 Multiprocessor Preemption

The multiprocessor preemption mechanism described in Chapter 16 is specific to this processor. Also, the write external priority flag and the interim priority field in the processor controls are also specific.

B.10 Breakpoints

The breakpoint registers are processor specific.

B.11 Implementation Dependent Instructions

The **synmov**, **synmovl**, **synmovq**, and **synld** instructions are specific to this processor.

B.12 Lock Pin

The $\overline{\text{LOCK}}$ pin is specific to this processor.

Release Note for BiiN™ Systems CPU Architecture Reference Manual

Version Information

This release note applies to Release 1 (R1) BiiN™ Series 60/80 systems, and to *BiiN™ Systems CPU Architecture Reference Manual* in particular.

Installation Procedures

Not applicable to this manual.

General Caveats

None.

Software Caveats

None.

Manual Caveats

None.

References to Nonproduct Utilities

None.

Additional Information

None.

23 August, 1988

INDEX

BiiN™

Please use this form to help us evaluate this manual and improve the quality of future versions. Mailing instructions are on the other side of this form.

If you want to order publications, see Page ii of this manual.

Manual Title: ______________________________ Revision Number (Page ii): __________

Please fill in the squares below with a rating of 1 to 10, with 1 being worst and 10 being best:

[ ] Technical completeness
[ ] Technical accuracy
[ ] Readability
[ ] Organization

[ ] Usefulness of material for your needs
[ ] Quality and relevance of examples
[ ] Quality and relevance of figures

If you gave a 4 or lower in any category, please explain here:

____

____

____

____

Please list any suggestions you have for improving this manual:

____

____

____

____

Please list any technical errors you found:

____

____

____

____

Please tell us your name and address. (If you would like us to call you for more specific comments about this book, please list your phone number as well.)

Name: ____

Address: ____

____

____

Phone (Optional): ____

Thank you for taking the time to fill out this form.

We'd like your comments . . .

Your comments help us produce better documentation. Please fill out the form on the reverse of this card, then fold it, tape it shut, and drop it in the mail. (Postage is free if mailed within the United States.) We will carefully review your comments. All comments and suggestions become the property of BiiN™.

Fold here

BiiN™
Technical Publications Department
2111 NE 25th Ave.
Hillsboro, OR 97124-9975