HEWLETT PACKARD

COMPUTER SYSTEMS - 19447 Pruneridge Ave. Cupertino CA 95014

From: Alan Hewer
Jim Miller
Dave Salomaki
Bert Speelpenning

Date: August 19, 1982

To: ACD Distribution

Subject: VISION Architecture Control Document, Version 5

We are proud to announce the release of Version 5 of the VISION ACD. Version 5 incorporates all changes and clarifications that have previously been transmitted only through memos. This single document provides a stable, self-consistent and complete description of the VISION processor architecture including I/O instructions. A version of the companion document "HP/3000 Compatibility Mode" incorporating the extended CST structure will be available in October.

Version 5 of the ACD replaces version 3, and the copy of version 3 in your possession must be shredded or returned to Bert Speelpenning at CSY. If you wonder what happened to version 4: there is no version 4 nor will there be. The designation "version 5" for the next major release of the ACD somehow gained currency within CSY, and it didn't seem particularly fruitful to buck that trend.

The organization of the architecture description has been thoroughly overhauled in order to improve clarity of exposition. These changes were sufficiently extensive that it was decided not to retain page or section numbers from version 3; change bars were also abandoned. Rather, version 5 is a stand-alone description of the architecture that should be read in full by implementors of VISION-specific products.

Turning to content, the architecture described in version 5 differs from that of version 3 in ways described in earlier memos published by us, as well as in other minor ways. These changes are summarized below. Their net effect is to make the VISION architecture more streamlined and easier to implement cost-effectively in hardware while minimizing the impact on software.

It is a pleasure to acknowledge the help and cooperation we have received in getting the architecture and its description to its current state; the implementation teams have been remarkably patient in helping us evaluate the effect of proposed changes as well as in accomodating those changes we decided to adopt.
Our main efforts will become focussed on monitoring the progress of VISION implementations; we remain committed to resolve problems in the present definition of the VISION architecture that these implementations may uncover.

Summary of significant changes between version 3 and version 5

1. Virtual address space has been cut back from 74 to 64 bits.

2. The number of privilege levels has been reduced from 8 to 4.

3. Object Descriptors have been streamlined to a 4-word format.

4. Procedure linkage has been simplified: the STT-mechanism is no longer required in Vision mode.
   Procedure stack markers have been reduced to three words instead of four; EXIT can distinguish between markers laid down by CALL and CALLX, this significantly streamlines exit from CALL.

5. Some STATUS bits and other machine state (such as the TCB) have been rearranged to allow faster updates to the addressing environment, such as EXIT or IEXIT.

6. Synchronization of caches and TLBs when making changes to the addressing tables has been made the explicit responsibility of operating system software.

7. The encoding of instructions has changed. Instructions or pairs of instructions now occupy a word or a multiple of words. Orthogonality of opcodes and operands has been retained.

8. 8 General registers have been added.

9. Instructions dealing with base registers have been separated out. Base registers are no longer treated as general operands.

10. Several instructions that were marginal in terms of speed-up over their software equivalents were deleted.
    All 16-bit arithmetic and all 12-byte packed decimal arithmetic has been deleted from the architecture.

11. Opcode assignments have been updated.

12. Several definitions of individual instructions have been streamlined.

13. Arguments for trap handlers are pushed in the reverse order. The trap identification number is now always on top of the stack.

14. The interrupt structure for I/O and inter-processor communication has been defined and included.
    I/O instructions for PICMB-based systems and for MPB-based systems have been defined and included.
    The interface to the Control and Support Processor (CSP) has been defined and included.

Not yet included in the Architecture Control Document are:

1. Instructions to support diagnostic capabilities.

2. Description of the data structures to support I/O.

VISION

ARCHITECTURE CONTROL DOCUMENT

VERSION 5

July 31, 1982

Your copy of this document is registered
COPY
NUMBER

INTRODUCTION — CHAPTER 1

1.1 VISION Architecture Control Document

This document provides, for reference purposes, the detailed and rigorous definition of the machine functions performed by VISION compatible computer systems. VISION provides the basis for a multitude of fully compatible systems over time which cover a broad spectrum of price and performance, benefiting from the exploitation of new or evolving technologies and machine organizations.

This is the only authoritative specification of the VISION architecture. It provides machine designers and programmers a complete description of the machine model which will transcend all implementations.

1.2 Architecture Overview

The VISION architecture is a product of the experience gained with the HP3000, HP300 and FOCUS systems. It provides two execution modes. One mode is highly compatible with the HP3000 and allows execution of HP3000 user level object programs. The Vision mode provides advanced information processing capability. The Vision mode is designed to retain the general purpose nature of its predecessors, but with enhanced ability to effectively address both business and technical applications. Vision mode is characterized by a powerful and complete basic instruction set, a wide range of data types, a stack for data allocation and procedure linkage information, data registers to support expression evaluation and addressing registers to support an extremely large task and system address space, paged memory management, a hierarchical protection system, and vector processing facilities.

1.3 Architecture Control

The term "architecture" as used in this document refers to the characteristics of the software/hardware interface of compatible VISION machines. "Hardware" refers to any combination of electronics, electro-mechanics and microcode.

The notion of Architecture Control has been created at HP to aid in the preservation of the investments it and its customers make in hardware and software with VISION based products. This architectural control document attempts to completely and unambiguously describe the features of any model claiming VISION compatibility.

To be successful, the following attributes of the architectural control process are stipulated:

1. This document is the only authoritative specification of the VISION architecture.
2. All models will be monitored for compliance with the architure specification.
3. Deviations from the architecture will be corrected. In the rare case when the cost to change the design or to retrofit installed machines is excessive in relation to the practical value of compliance on that model, deviations are permitted when approval is obtained from all affected group managers, and appropriate provision is made for the exception in the Architectural Control Document.
4. Implementers are instructed to question any doubtful point in the architectural definition rather than make assumptions. The specification occasionally leaves out some aspect of the operation, or the wording may not be clear. In these cases the document should be updated to resolve the point.
5. Continual maintenance and updating of the architecture specification are essential.
6. At any point in time, the management will entrust maintenance of the architecture control document to a person or small group. They will be responsible for resolving conflicts, creating and reviewing document revisions, stopping debate on some issue, etc, through the use of technical and business analysis or executive decision making.

1.4 Intended Audience

This document is intended primarily as reference material for implementors of VISION-compatible products; specifically, implementors of hardware and microcode, implementors of core operating system modules and implementors of Vision compilers.

The first five chapters can be used as a stand-alone introduction to the main VISION features; they cover the VISION addressing structure, which is the most distinguishing characteristic with respect to its predecessors and current competition in the market. To this end these chapters are written in a more tutorial style.

This document must not be shown to and must not be read by non-HP employees.

1.5 Related Documents

"HP/3000 Compatibility Mode", internal, January 1982.

"PICMB ERS", internal.

"MPB ERS", internal.

"Interface Protocols for the Control Support Processor for VCF60 and VCF50", internal, to appear.

"A Proposed Standard for Binary Floating Point Arithmetic", draft 9.3.3 of IEEE task P754.

"Time and Frequency Users' Manual", NBS Special Publication 559.

1.6 Notations and Conventions

Algorithms in this document are described in a pidgin PASCAL. In these algorithms:

- NAMES in capital letters denote processor registers;
- Names with only the first letter capitalized denote temporary or scratchpad values;
- names in lower case denote parameters or operands.

The notation R[0..5] denotes a 6-bit field consisting of bits 0 through 5 of register "R".

The notation (R)[0..31] denotes the 32-bit word found at the byte address contained in register "R".

The numbering of bits and bytes is such that the lowest numbered bit (byte) contains the most significant information.

Numbers are given in decimal (default) or in hexadecimal notation (when preceded with an "!", e.g. !1A denotes the number 26).

1.7 Implementation Guidelines

Experience related to cost-effective implementation of VISION hardware and software will be disseminated to other VISION implementors when such experience becomes available. Emphasis will be on the more subtle implications of the architectural specification; in particular, performance implications. Recommended software practices, if sufficiently important to the performance of at least one VISION implementation, will be included also.

As an example, all VISION hardware will allow data to be addessed on arbitrary byte boundaries. Yet performance will be degraded significantly if data is not aligned on its natural boundaries: half words on half-word boundaries, words on word boundaries, etc. This effect will be felt on any VISION implementation, to varying degrees.

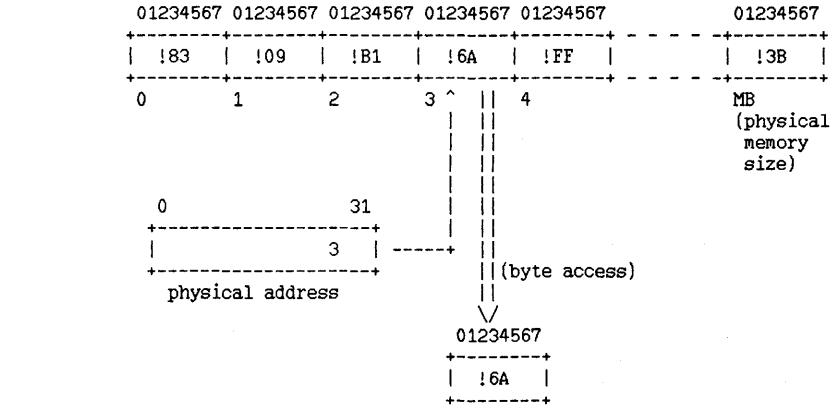
This document includes some implementation guidelines when it was deemed that their inclusion would clarify the issue at hand.

ADDRESS SPACES — CHAPTER 2

2.1 Physical Address Space

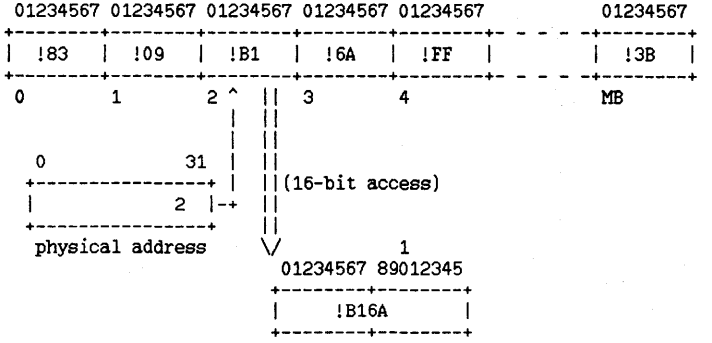
Physical memory is organized in bytes. A byte is a collection of 8 bits. Each byte in physical memory has its own unique number called its physical address. A byte is the smallest addressable unit. A physical address is represented by a 32-bit quantity. Physical memory is the lowest type of memory in the memory hierarchy that is visible to software. Implementations may choose to organize physical memory in words (4-byte quantities) instead of bytes; they may employ caches to reduce average access time, but this must remain transparent to software.
Input/Output is performed using physical addresses. Software uses "logical addresses" (section 2.3) to access memory; these logical addresses are translated by address translation hardware (chapter 3) into the corresponding physical addresses.

Representation of physical memory:



In this example, the physical address is 3, and therefore the byte with identifying number 3 is being addressed. Its contents is !6A (01101010).

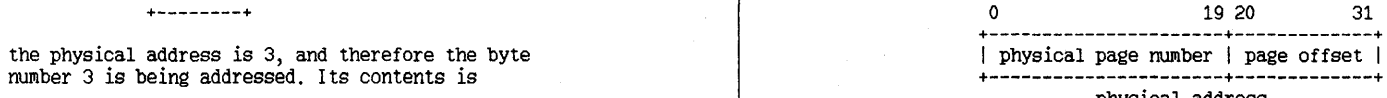
Addresses are also used to access entities larger than a byte. (This will apply similarly to logical and virtual addresses). In this case, the address reflects the lowest numbered byte. The following example shows the result of a 16-bit access using physical address "2":



Physical address space is used for four separate purposes. The first purpose is to provide physical pages (see section 2.1.1) that support virtual memory (section 2.2). The second is to provide a location for the physical page directory (section 2.2.2) and hash table (section 3.4) that constitute the foundation for the address translation algorithm and therefore must reside at a physical address known to hardware. The third is to provide a "scratch area" for each processor, transparent to software (see chapter 9). This scratch area is used to hold data for which use of processor registers would be too expensive; it is also used by the I/O channels when they need to communicate with the processor. The fourth is to provide addressability for I/O buffers. VISION I/O channels only access memory through physical addresses.

2.1.1 Pages

Physical memory is divided into physical pages of 4096 bytes each, partitioning physical addresses into a 20-bit physical page number (PPN) and a 12-bit page offset (POFF), as follows:



2.2 Virtual Address Space

In VISION, the virtual address space is extremely large. Its size is 2^64 bytes. This address space is so large that it frees software for the most part from having to reclaim and repack virtual space no longer in use. Whole data bases can reside in virtual space, if so desired.

Virtual space allows programs to run that require address space in excess of the amount of physical memory available on the machine. This is accomplished through "demand paging": a mode of operation in which part of virtual memory is kept in physical memory and the rest kept on secondary storage; if a request for access to virtual memory cannot be satisfied out of physical memory, the page containing the virtual address is read in from secondary storage (another page may have to be written to secondary storage in order to make place for it in physical memory).

Operating system software policy determines the precise details of where virtual pages are kept on secondary storage and where in physical memory a new virtual page is read in. Hardware is only responsible for detecting "page fault"s: the condition where an access to virtual memory cannot be satisfied out of main memory. On detecting a page fault, hardware will transfer control to the page fault trap handler (section 7.xx). The operating system must then resolve the page fault and transfer control back to the user program in a way that makes the occurrence of the page fault totally transparent to the user program (except perhaps for a noticeable delay).

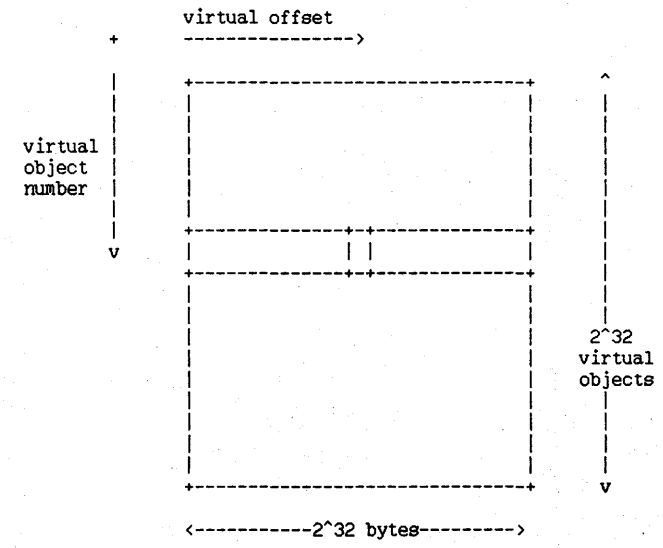
Demand paging in a large virtual space frees writers of software from stringent and inescapable limits on programs due to size of physical memory. Instead, writers of software for a demand paged machine face a concern for "locality of reference". Basically, one program's locality of reference is better than another's if it accesses fewer different virtual pages in a comparable time span. Programs with better locality of reference will require less page swapping and therefore will perform better, other things being equal.

Details on how a virtual address is translated to a physical address in VISION are deferred until chapter 3.

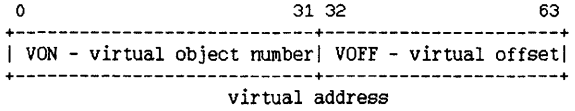
Two different views of virtual address space are given here: one from the perspective of address arithmetic and allocation, the other from the perspective of paging and memory management.

2.2.1 Virtual Address Space: Virtual Objects

Virtual address space is organized as 2^32 "virtual objects" of 2^32 bytes each:

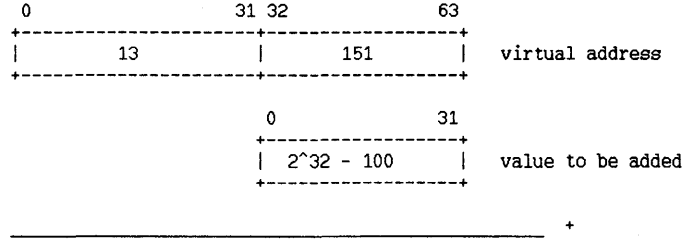


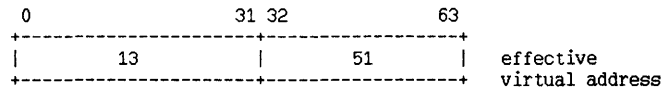
This organization of virtual address space corresponds to a partitioning of a virtual address as follows:



The significance of this subdivision is that all address arithmetic on a virtual address is performed on the 32-bit virtual offset without ever carrying into the virtual object number. The virtual object number is never altered by address arithmetic. The virtual offset is regarded as a two's complement quantity; overflow on address arithmetic is ignored. Address arithmetic on virtual addresses occurs during translation of logical addresses as described in chapter 3.

Example:

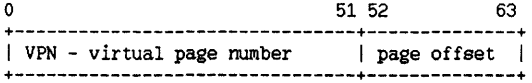




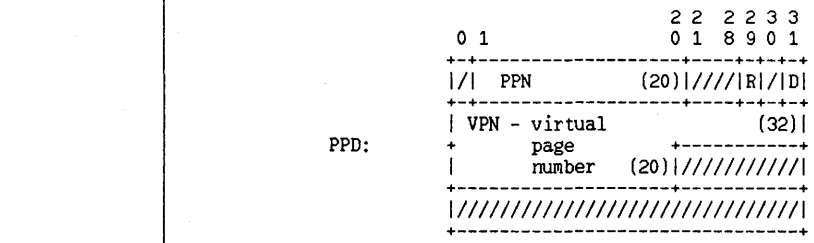
Hence address arithmetic, even if nominally involving 64-bit quantities, can be implemented in hardware with 32-bit ALUs.

2.2.2 Virtual Address Space: Paging

From a paging perspective, a virtual address (VA) is split into a 52-bit virtual page number (VPN) and a 12-bit page offset (POFF), as sketched below. (Some VISION documentation refers to VA and VPN by the names GSVA and GSVP, respectively.)



A physical page is associated with a virtual page; this will typically change over time as pages get swapped in and out, i.e. read and written to secondary storage (usually disk storage). At each moment, the state of the association is contained in the physical page descriptor (PPD), as follows:



where:

PPN - physical page number (20 bits).

VPN - virtual page number (52 bits): the number of the virtual page currently associated with page PPN.

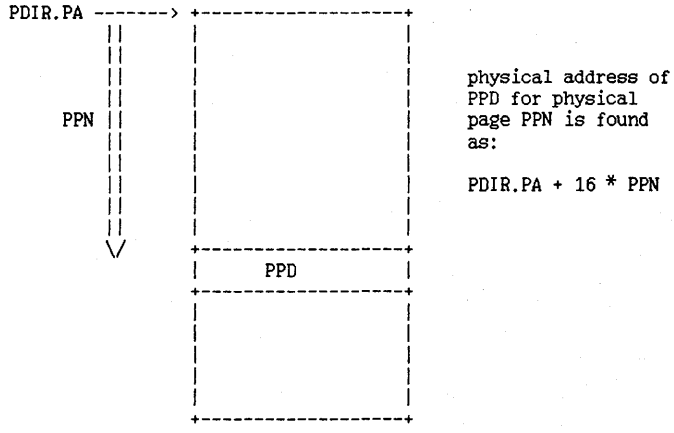
D - dirty bit (1 bit): set to one by hardware if the contents of the physical page has been changed by the processor since it was read in from secondary storage, i.e. when the page on secondary storage is no longer up to date.

R - reference bit (1 bit): set to one by hardware if the physical page has been accessed since the last time software caused the reference bit to get reset.

The PPD has additional fields that are described in section 3.4.

2.2.3 The Physical Page Directory (PDIR)

Each physical page has its own physical page descriptor; all PPDs are collected in the physical page directory (PDIR).



physical page directory (PDIR)

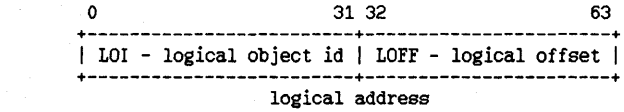
The physical address PDIR.PA is kept in a processor register.

Note that the PPN field contained in the PPD is redundant with the position of the PPD within the PDIR.

2.3 Logical Address Space

Logical address space provides the third and highest level of addressing in the VISION architecture. Logical address space serves to insulate and protect programs from each other. At the same time, logical address space allows operating system software full control over arbitrary patterns of sharing and of access protection between user programs.
All programs run in logical address space. All addresses that are directly constructed by a user program are logical addresses; these are presented to hardware for address translation. Hardware translates logical addresses (via virtual addresses) to physical addresses as detailed in chapter 3.

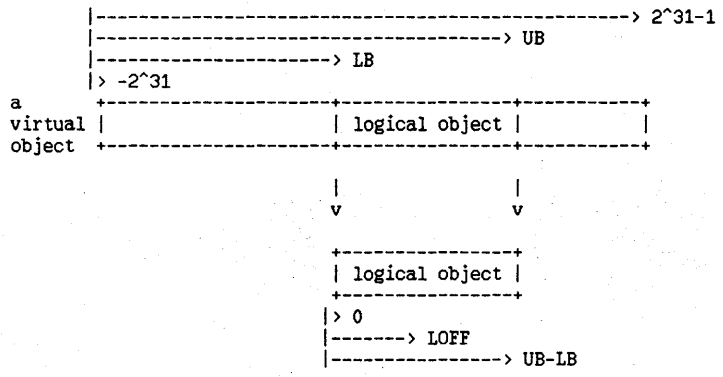
A logical address is a 64-bit quantity. The first 32 bits identify a "logical object"; logical objects are defined in section 2.3.1. The second 32 bits of the logical address contain a "logical offset", i.e. a byte offset relative to the beginning of the logical object. Below is a depiction of a logical address:



logical address

2.3.1 Logical Objects

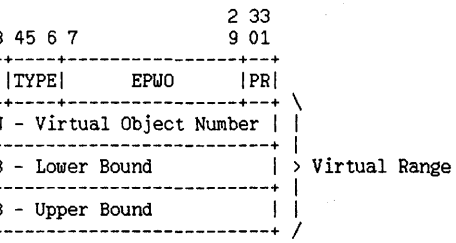
A logical object is a slice of a virtual object that can only be accessed through an Object Descriptor, which enforces access rights and bounds protection. Object descriptors are detailed in section 2.3.2.



The figure above shows how two virtual offsets (LB and UB, lower and upper bound) are used to delineate a slice of a virtual object. Such a slice is called a Virtual Range. The Virtual Range, under protection of access rights, constitutes a logical object.
The bytes in the logical object are numbered from 0 to UB-LB; it is this numbering, relative to LB, that is used by the logical offset LOFF in a logical address.

2.3.2 Object Descriptor Format

Object Descriptors are 16 bytes in size. The format is as sketched below:



where

AR = access rights. This field encodes the access rights to the object allowed at each of the protection levels, as detailed in section 2.3.2.2.
TYPE= object type. The encoding of this field is detailed in section 2.3.2.1.
EPWO= entry point word offset. This field is meaningful only for Vision mode code objects. It is detailed in section 5.2.
PR = prerequisite level. This field is meaningful only for Vision mode code objects. It is detailed in section 5.2.
VON = virtual object number. Identifies the virtual object of which this Object Descriptor defines a slice.
LB = lower bound. See section 2.3.2.3.
UB = upper bound. See section 2.3.2.3.

2.3.2.1 Object Types

Object types are encoded in a 3-bit field in the OD as follows:

0 - Vision mode code object
1 - Reserved object type
2 - Vision mode stack object
3 - data object
4 - HP3000 mode code object
5 - HP3000 mode code object, subject to PCAL trace
6 - HP3000 mode stack object
7 - Reserved object type

The object type "HP3000 code object subject to PCAL trace" is explained in more detail in the "VISION HP3000 mode document". The reserved object types will block access to the object; one of these types may become defined in later versions of the VISION architecture.
All other object types are explained fully in this document.

2.3.2.2 Access Rights

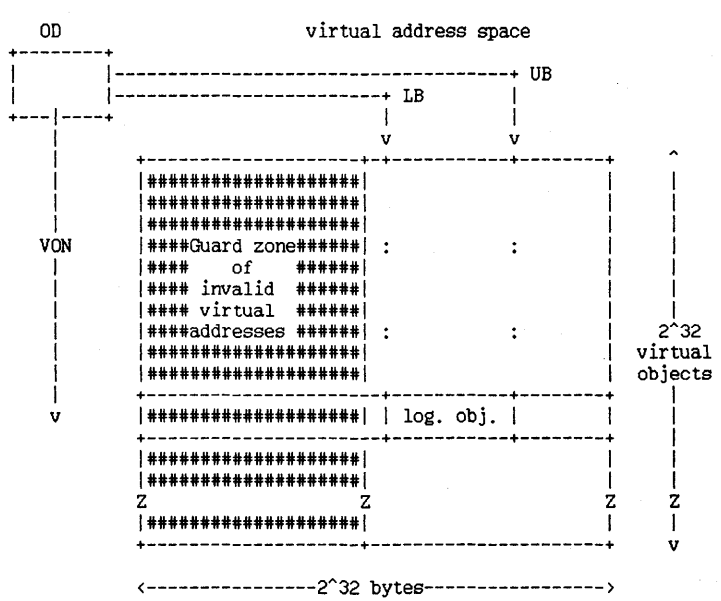
At each moment, the processor runs in one of two execution modes (Vision mode or HP3000 mode) as indicated by the one-bit processor state "XM". In Vision mode, four levels of privilege are supported, ranging from 0 (most privileged) to 3 (least privileged); and the processor will run at one of these four privilege levels, as indicated by the two-bit processor state "XL". In HP3000 mode, the processor runs in either "User state" (which is identified with XL=3) or in "Privileged state" (which is identified with XL=1).
For purposes of protection, accesses are characterized as either "read", "write" or "execute". The following chart defines the conditions for legal access. Illegal accesses cause a trap as defined in chapter 7.

TYPE (from OD)	read	write	execute
Vision code object	XL <= AR[0..1]	illegal	XL <= AR[2..3] & XM=0
Vision stack object	XL <= AR[0..1]	XL <= AR[2..3]	illegal
data object	XL <= AR[0..1]	XL <= AR[2..3]	illegal
HP3000 code object (w/ or w/o trace)	XL <= AR[0..1]	illegal	XL <= AR[2..3] & XM=1
HP3000 stack object	XL <= AR[0..1]	XL <= AR[2..3]	illegal
reserved obj type	illegal	illegal	illegal

Notes: "<=" means less than or equal in an unsigned 2-bit compare.
The objects pointed to by special registers P and Q require special treatment, as detailed in sections 2.3.5 and 2.3.6.

2.3.2.3 Lower and Upper Bounds

Logical objects are slices of virtual objects protected by a lower bound and an upper bound, as sketched below:



Both lower and upper bound are 32-bit two's complement quantities that delineate the logical object; both bounds are inclusive. Any address arithmetic (e.g. indexing) involving a logical object that causes the virtual offset to stray outside the bounds given in the Object Descriptor will result in a trap. (This applies only to addresses actually used in accesses, not to preparatory address calculations.)

Though lower and upper bound are two's complement quantities, their values must always be positive. It is the responsibility of operating software to ensure this. The size of logical objects is therefore limited to 2^31 bytes. Note that zero-length objects can be supported by having UB = LB -1 in the Object Descriptor.

2.3.3 Object Groups

Object Descriptors belong to the program that is executing; more precisely, they are associated with a task.
It is the responsibility of operating system software to implement a policy of protection and security, by setting up only such Object Descriptors on behalf of each task as are needed by the task to perform its rightful function.
It is the responsibility of hardware to enforce the access rights and bounds protection as contained in the Object Descriptor.

Object Descriptors are organized in groups in order to facilitate sharing of objects among tasks. The VISION architecture provides for eight object groups per task. Each group has a data structure (Object Descriptor Table, see section 2.3.4) that maps a logical object id onto an Object Descriptor. Tasks that share all objects in a group can therefore share the Object Descriptor Table for that group as well, resulting in reduced duplication of Object Descriptors and (as an important side effect) faster task creation.

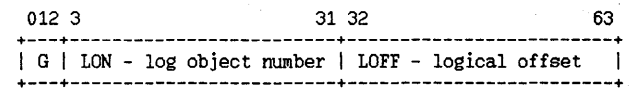
Group zero is the same for all tasks, i.e. logical addresses with a zero group selector translate to the same virtual address regardless of which task is translating it.
Groups one through seven are either shared or task-specific, completely under operating system software control.

The 32-bit logical object id (LOI) serves to locate the Object Descriptor for the logical object. For this purpose, LOI is split into a 3-bit group selector (zero through seven) and a 29-bit logical object number (LON) interpreted relative to the selected group:

where

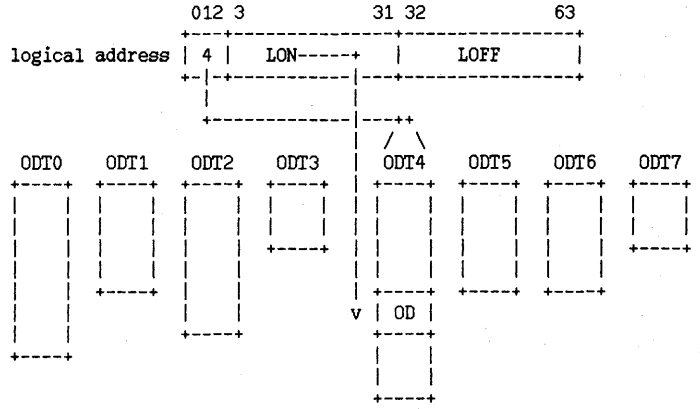
G = group selector (3 bits);
LON = logical object number (29 bits).

The format of a logical address can therefore be depicted as:



2.3.4 Object Descriptor Table

Each object group has an Object Descriptor Table (ODT) which is a linear collection of Object Descriptors indexed by the logical object number. The Object Descriptor Table is itself an object in virtual and logical address space and its location is known to hardware, as detailed in section 2.3.8 and chapter 4. Both virtual lower bound and virtual upper bound of an ODT must be multiples of 16.

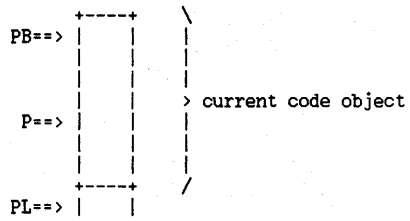


Virtual address of OD =

virtual address of selected ODT + 16 * LON

2.3.5 Current Code Object

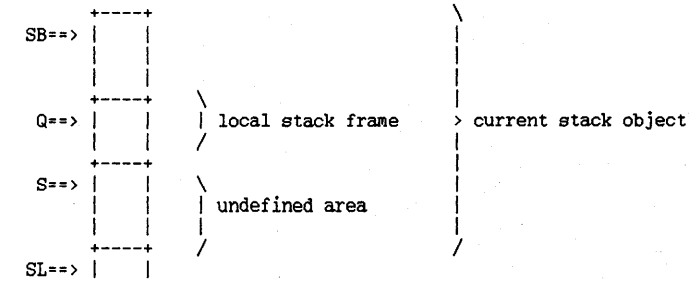
In Vision mode, there is a 64-bit Program Counter P that holds the logical address of the instruction currently executing. Bits P[0..31] contain the logical object id of the object known as the current code object. In VISION documentation, PB is used as shorthand for the (logical address of the) lower bound of the current code object; similarly, PL is used to point to the first byte beyond the current code object.



Read access to the current code object is allowed; this overrides the access right field in the OD. Both LB and UB in the OD for the current code object must be multiples of 4. The size of the current code object must be no more than 2^24 bytes. It is the responsibility of operating system software to ensure LB and UB meet these requirements. P will always be even (P[63]=0); this is ensured by hardware checks in the EXIT and SEXIT instructions.

2.3.6 Current Stack Object

Two Vision mode registers Q and S both hold logical addresses. Bits Q[0..31] are at all times identical to bits S[0..31]; these identify the logical object known as the current stack object. The significance and the use of Q, S and the stack object is detailed more fully in chapter 5. In VISION documentation, SB is used as shorthand for the (logical address of the) lowest numbered byte of the current stack object; SL denotes the first byte beyond the current stack object.



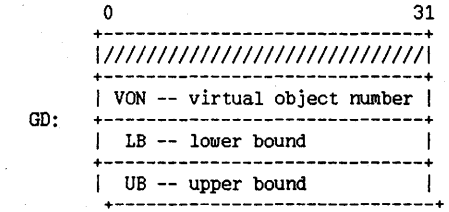
Read and write access at any privilege level to the current stack object is allowed; this overrides the access rights field in the OD for the current stack object. All accesses to the stack object must fall within the bounds SB (inclusive) and S (exclusive); this overrides the upper bound in the OD for the current stack object. Both LB and UB in the OD for the current stack object must be multiples of 4. In addition, UB must satisfy UB < 2^31-4. It is the responsibility of operating system software to ensure that LB and UB meet these requirements.
All changes to Q and S must satisfy SB <= Q <= S <= SL.
(Q and S only change as a side effect of certain instructions such as CALL and EXIT. See chapter 6 for detail.)

2.3.7 Nil Object

Logical object zero in group zero (logical object id = 0) is inaccessible to all software. Operating system software is responsible for maintaining the OD for LOI=0 such that no accesses are legal. To do so, it suffices to set UB = LB - 1. This allows the nil pointer to be represented conveniently by a logical address consisting of 64 zeros.

2.3.8 Group Descriptors

A Group Descriptor (GD) serves to locate an Object Descriptor Table for a particular group. The Group Descriptor for the ODT for group 0 is kept in processor registers: it is machine state. Group Descriptors for groups 1 through 7 are contained in the Task Control Block (see section 2.3.9). GDs occupy 16 bytes and have the format as shown:

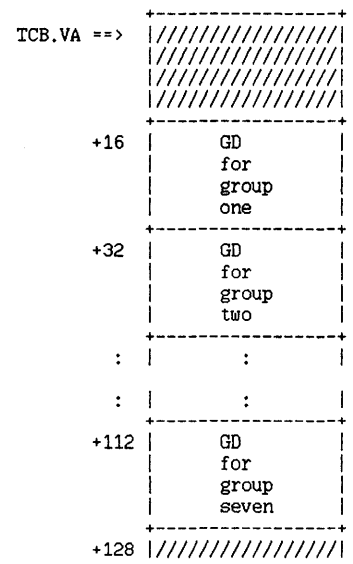


The first word of the Group Descriptor will be detailed in section 4.5. The remainder of the Group Descriptor contains a Virtual Range, as in an Object Descriptor. The Virtual Range locates the ODT in virtual space.
Operating system software is responsible for ensuring that LB and UB are multiples of 16, so that all ODs in the ODT will be aligned on 16-byte boundaries.

2.3.9 Task Control Block

The Task Control Block (TCB) of the currently executing task is a software data structure whose location and layout is known to hardware. The 64-bit virtual address of the TCB is kept in a processor register TCB.VA. The value of this virtual address can be changed by the "LAUNCH" instruction when performing a task switch.

Operating software must ensure that the TCB of the currently executing task is resident in physical memory. TCBs for tasks not currently executing are not in any way constrained by the VISION architecture. The full layout of the TCB is given in section 4.7. Only the part of the TCB involved in defining logical address space is shown here.



The TCB virtual address TCB.VA must be a multiple of 16.

ADDRESS TRANSLATION — CHAPTER 3

Software running on the VISION architecture interacts with memory continually. Software is made up of instructions that must be fetched from memory; memory is read, the data examined, processed, and the results stored back into memory.
To perform these memory accesses, hardware computes logical addresses that are then translated to physical addresses. Computing logical addresses can be as simple as incrementing the program counter (P) on each instruction fetch, or it may involve adding together the displacement value out of an instruction with the contents of the offset part of a base register in order to form what is called the "effective logical address". Logical address computation is detailed in chapter 6. This chapter explains address translation in VISION. Address translation always accompanies a request for access to the memory system; hence it proves convenient to explain address translation in the context of such an access.

3.1 An Access -- its characteristics

A memory access in Vision mode has these three characteristics:

a) a logical address, LA
b) a length in bytes, L
c) a type of access, read/write/execute/semaphore-read

Read and write accesses can be performed on entities that vary in length from 1 through 16 bytes. Execute access (instruction fetch) can be performed on multiples of 4 bytes.
Semaphore-read is a special type of access that can be performed only on 4-byte quantities; it consists of a read followed indivisibly by a write of all ones. For purposes of address translation, the indivisible nature of semaphore-read is of no import; we can treat it in this chapter simply as a read followed by a write to the same address.

A multi-byte read (L>1) is implemented as if it were a sequence of single byte reads at addresses LA, LA+1, .. , LA+L-1, the results of which are collected into a hardware buffer area. If any of the single-byte reads fails (e.g. bounds violation or page fault) the single-byte reads that did succeed are non-destructive: as far as software can tell, no machine state is modified.

A multi-byte write is implemented as if it were a series of single-byte writes at addresses LA, LA+1, .. , LA+L-1, with the proviso that hardware implementations are free to perform the single-byte writes in any order. This means that if any of the single-byte writes fails (e.g. bounds violation or page fault), software cannot make any assumptions about the state of the addressed memory.

Note: an instruction that has a read-write operand (an operand that is both read and written as part of the same instruction) must never write any bytes of the operand unless it can guarantee that writing all bytes of the operand will be completed. (This accomplished trivially on single-processor systems by prefetching the operand; on shared-memory multi-procesor systems it requires that the PDDEL instruction allow any on-going instructions to first complete.)

Instruction fetch is like read in that its effect can be viewed as a sequence of byte-reads into a hardware buffer area, transparently to software.

Without loss of generality it is then possible to describe address translation in terms of single-byte accesses only.

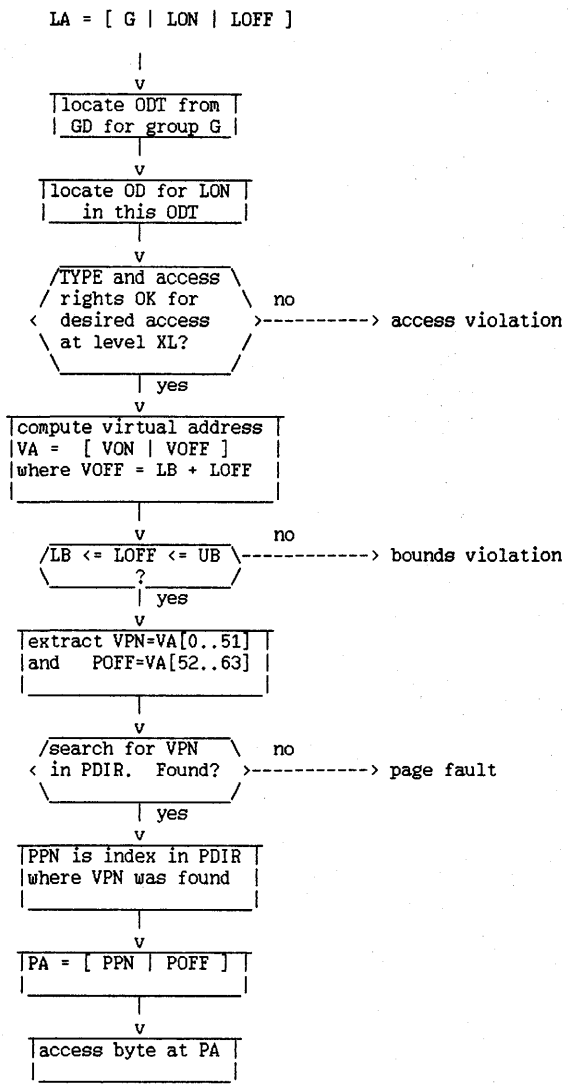
3.2 Access Algorithm

Accessing a byte at logical address LA can be described by an algorithm, developed in detail over the next few sections, that requires the following items of machine state as input:

XL - execution level of the hardware
TCB.VA - virtual address to the TCB of the currently running task
PDIR.PA - physical address of the PDIR, the physical page directory
HASH.PA - physical address of the Hash Table

3.2.1 Schematic overview

The next page shows an overview of address translation for accessing a single byte at logical address LA.



3.2.2 Hardware Shortcuts in Address Translation

The architectural definition of address translation should by no means be read as a precise indication of all the steps performed by hardware in their precise order.
In order to be cost-effective, hardware "must cheat but not get caught". Just as a cache holds recent memory accesses in a faster but smaller type of memory, so can hardware employ various types of devices to speed up address translation by setting aside recent results of address translation. The most trivial example would be locating the eight ODTs. Hardware may do this only once after a task switch and keep the virtual range of each ODT for the current task in some internal register. This involves a trade-off between the speed of the task switch itself and the speed of all subsequent address translations. Hardware may also choose to keep recent pairs of (logical address, physical address) around, or recent pairs of (LOI, OD) and/or recent pairs of (VPN,PPN).
Any such association of recent pairs that are presumably useful in bypassing parts of the address translation algorithm is referred to in this document by the name "TLB", short for "Translation Look-aside Buffer".

Address translation is effected through address translation tables that are in part task-specific, in part system-wide. Changes in address translation are relatively infrequent, e.g. (VPN,PPN) associations become outdated only on page swaps, (LOI, OD) associations become outdated only on task switches and on explicit changes to ODTs brought about by the object management facility in VISION operating system software. Because these changes are relatively infrequent, and because the situations in which they arise are under explicit operating system software control, it is particularly cost-effective to recognize architecturally the existence of some kind of TLB, while leaving the exact nature of the TLB to the discretion of the hardware implementation.

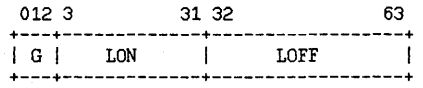
The existence of the TLB is recognized architecturally in VISION by requiring operating software to issue explicit instructions that warn hardware of the fact that the conditions for address translation have changed and that information in the TLB may no longer be up to date. Address translation look-aside in VISION hardware therefore need not be completely transparent with respect to changes in addressing tables.

3.3 Logical to Virtual Address Translation

Logical to virtual address translation for logical addresses in groups one through seven is similar, but not identical, to translation of logical addresses in group zero. The differences are limited to the way the ODT for the group is located. In actual hardware implementation, even these differences may be absorbed in the work performed on a task switch after which the logical to virtual translation is done in the same way for all groups.

3.3.1 Locating the ODT in Virtual Space

Starting out with a logical address:



it is first necessary to locate the ODT for group G in virtual space.

3.3.1.1 Locating the ODT for Group Zero.

Locating the ODT for group 0 is very simple: the Group Descriptor for group 0 (GD0) is kept in a processor register. This Group Descriptor includes a Virtual Range that delineates ODT0 in virtual space. See section 4.5 for more detail.

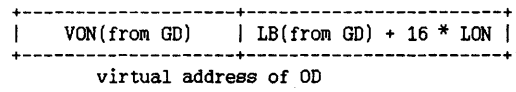
3.3.1.2 Locating the ODT for a Group Other than Zero

Locating the ODT for group G, where G>0, requires accessing the TCB of the currently executing task in order to obtain the Group Descriptor for group G. The TCB is accessible to hardware through the TCB.VA virtual address; TCB.VA is kept in a processor register.

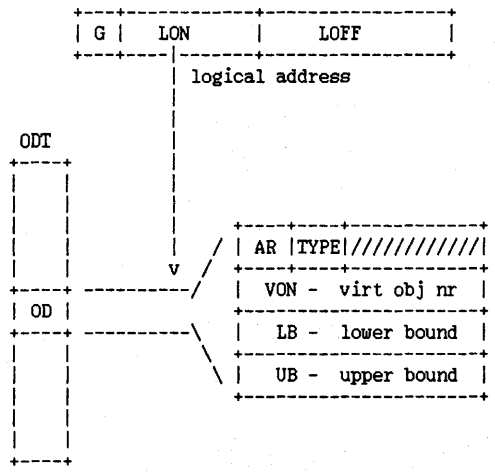
In practice, hardware implementations may choose to access the TCB once on IEXIT and copy the seven Group Descriptors into processor registers.

3.3.2 Locating the OD for the Logical Object

Having located the ODT in virtual space through the Group Descriptor GD, the Object Descriptor OD of the logical object can now be found by computing:



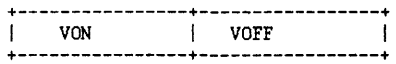
This OD can now be accessed in virtual space using the procedure described in section 3.4. Note: a hardware implementation may choose to cache recent pairs of (LOI, OD).



3.3.3 Computing the Virtual Offset

Having located the OD for the logical object, type checks and access right checks can now be performed. Software must be informed of any violation uncovered in these checks (through the trap mechanism detailed in chapter 7)

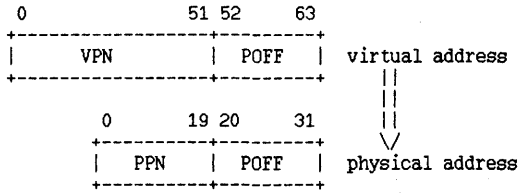
From this OD and the logical offset LOFF, the virtual address



can now be computed as: VOFF = LB + LOFF (in wrap-around 32-bit two's complement arithmetic). However, software must be alerted of a bounds violation if LB <= VOFF <= UB does not hold.

3.4 Virtual to Physical Address Translation

Translating a virtual address (64 bits) to a physical address (32 bits) means translating a 52-bit virtual page number (VPN) into a 20-bit physical page number (PPN) and carrying the 12-bit page offset (POFF) along, as indicated in the sketch.



3.4.1 Physical Page Directory Search

The translation of a virtual page number VPN into a physical page number PPN may give different results over time as pages are swapped in and out of physical memory. It is the responsibility of operating system software to maintain the page directory PDIR (see section 2.2.3) with the help of the instructions PDINS and PDDEL. The page directory PDIR gives the current association of physical pages with virtual pages. For each physical page PPN there is a physical page descriptor PPD (see section 2.2.2) that describes the VPN currently associated with it. In principle, it is therefore sufficient to do a linear scan over the PDIR looking for a PPD that has the correct VPN in it; this identifies the proper PPN or else it establishes that no physical page is associated with this VPN and a page fault is thereby indicated.
However, a linear scan would be unacceptably slow even when hardware keeps enough recent pairs (VPN,PPN) around in a TLB to produce an excellent hit rate. The entire PDIR would have to be scanned in order to establish, for instance, that the virtual page is nowhere in physical memory (page fault trap). The VISION architecture therefore defines a hashing technique in order to speed up the search for the right virtual page number and to speed up detecting a page fault condition.

3.4.2 Overview of Hash Table and Hash Chain

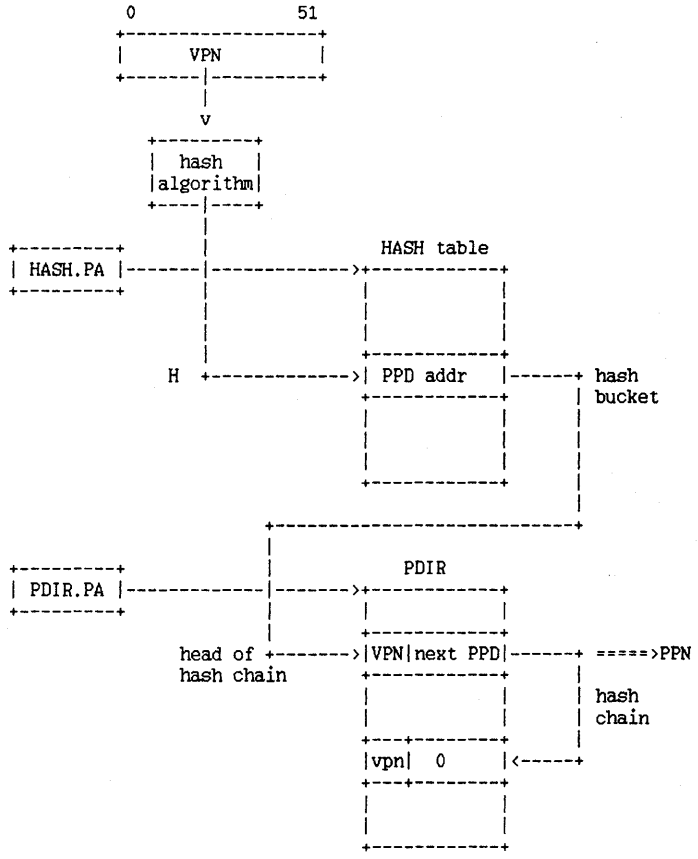
In order to avoid having to scan large parts of the PDIR on a virtual page to physical page translation, a hash value is computed from the VPN:

```
H = hash( VPN );
```

where the hash function "hash" is described in section 3.4.5. All PPDs in the PDIR of virtual pages that have the same hash value H are chained together and the beginning of the chain can be found in the hash table HASH. These chains and the hash table HASH are maintained with the help of the PDINS and PDDEL instructions.

The number of entries in the HASH table (which must be a power of two) should be at least of the same order as the number of entries in the PDIR in order to keep the chains acceptably short. The number of entries in the PDIR is determined by the number of physical pages in the hardware configuration.

Note that by its very nature the pages that make up the PDIR itself must never be absent and the entire PDIR must be contiguous in physical memory. The VISION architecture places the same constraints on the HASH table. Both PDIR and HASH are located through processor registers HASH.PA and PDIR.PA that contain the physical address of each table.



Overview of virtual to physical page translation

Note that the only actions ever performed on a 52-bit VPN are:

a) computing the hash function

b) comparing for equality against a field in the PPD.

The latter can be implemented by two 32-bit compares. The VPN acts purely as a tag and never participates in 52-bit arithmetic.

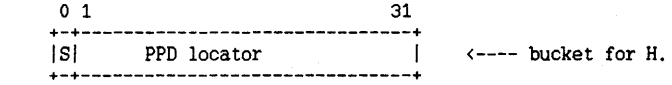
3.4.3 The Hash Table

The Hash Table is a collection of "hash buckets", each 32 bits long. The physical address of the hash bucket is calculated as

HASH.PA + 4 * H, where H = hash(VPN),

as described in section 3.4.5.

The format of a hash bucket is:



PPD locator : this value is the physical byte address of one of the PPDs associated with a virtual page that hashes to the value H. There may be more than one such PPD, in which case the locator will simply point to the head of a linked list; or there may be none, in which case the PPD locator will be zero. A zero value for the PPD locator will indicate a page fault.

S = semaphore bit. Available for use by PDINS and PDDEL to synchronize changes in the HASH and PDIR tables in a shared-memory multiprocessor system. Outside such use, hardware may assume that its value is zero.

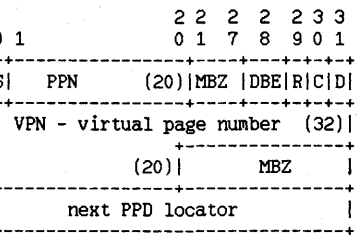
Note: To avoid ambiguity in the definitions shown above, the PDIR must reside in the first half of physical address space. The PDIR must not start at physical page zero.

3.4.4 PPD Format to support Hashing and Related Functions

Each PPD is a 16-byte entity whose physical address is related to the physical page number PPN through the formula

physical address of PPD = PDIR.PA + 16 * PPN.

The PPD format is detailed below:



where

VPN = virtual page number. This is the virtual page currently associated with the physical page PPN. If this field matches the VPN of the virtual address being translated, the right physical page has been found. If no match occurs, the PPD at the "next PPD locator" should be checked for a match.

next PPD locator: a physical byte address used to locate the next PPD that contains a virtual page number that hashes to the same HASH value as VPN. This field is consulted only if the VPN in the current PPD does not produce a match. If the entry is then found to be zero, it means that the end of the chain has been encountered, and a page fault trap is indicated.

PPN = physical page number. This field is redundant as it can be easily derived from the physical address of the PPD, but it is available for hardware use. It is the responsibility of operating system software to ensure that PPN is at all times consistent with the PPD address.

D = dirty bit. Set to one by the processor on each write access to the page. To be cleared by operating syste software when the page is written out to secondary storage. D is not affected by I/O traffic.

C = checkpoint dirty bit. Set to one by the processor on each write access to the page. To be cleared by operating system software. It may be used at the discretion of the operating system.

R = reference bit. Set to one by the processor on each access to the page. Operating system software will clear this bit on a periodic basis such that finding the reference bit set can be interpreted to mean that the page was referenced recently (either explicitly by the program or implicitly by certain prefetch hardware). R is not affected by I/O.

DBE = debug breakrange enabled. Set to one by operating system software whenever the page contains any part of the system or local breakrange. Hardware bypasses the breakrange checks when accessing a page with DBE zero. See section 5.3.2 for more detail.

S = semaphore bit. For hardware use in synchronizing write access to the reference and dirty bits in a shared-memory multiprocessor configuration. Before and after such use, S must be zero.

MBZ = must be zero. It is the responsibility of operating system software never to introduce a non-zero value into this entry. Hardware may assume the field is zero and it need not check this.

SPECIAL NOTES:

1) To avoid ambiguity in the above definition, the PDIR must reside in the first half of physical address space, yet must not start at physical address zero. The physical address PDIR.PA must be a multiple of 16.

2) Software must not access (read or write) the PPDs, but must instead rely on special instructions to deal with them:

   PDDEL - remove a PPD from its hash chain
   PDINS - insert a PPD in its hash chain
   TESTREF - read and reset reference bit

   Only a PPD not currently in a hash chain (e.g. after PDDEL has extracted it) may be accessed by software. These restrictions simplify hardware synchronization.

3) Shared-memory multiprocessor implementations that use write-to TLBs may use the MBZ field as a semaphore area in order to synchronize writing out dirty and reference bits.

3.4.5 The Hash Algorithm

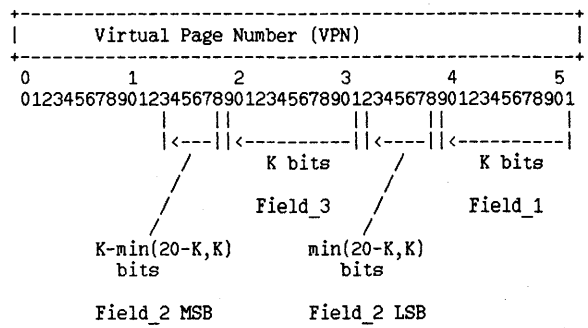
The purpose of the hash algorithm is to break up the long list of physical page descriptors PDIR into a large number of short chains so that, in order to find the physical page corresponding to a given virtual page, only the physical page descriptors in a single short chain need be scanned. In order to keep the chains short, the hash algorithm must succeed at producing different hash values for those virtual pages that are likely to be in use simultaneously. The hash algorithm used in VISION accomplishes that by producing different hash values for those virtual pages that are allocated by the operating system closely together in time. The VISION hash algorithm cannot succeed in doing so without some minimum cooperation from operating system software. In particular, allocation of virtual memory is assumed to be done either contiguously within the same virtual object or else in units of an entire virtual object. The hash algorithm can be defeated by the operating system allocating virtual memory in 2^25 byte slots, for instance.

The hash algorithm presented here is really a family of hash algorithms, parameterized by the single quantity K. It takes a virtual page number as input and produces a 32-bit number:

```
H[0..31-K]  := 0;
H[32-K..31] := hash( VPN );
```

The number of hash buckets in the HASH table must be a power of two: 2^K. The HASH table length is therefore 4*2^K bytes.

The hash algorithm is sketched below. A PASCAL version of the hash algorithm is given on the next page.



a) Field_1 is defined to go from bit 52-K to bit 51.

b) Field_2 is divided into two parts. The most significant bits are defined to start at bit 32-2*K+min(K,20-K) and go to bit 31-K. Note that this part is empty for K<10.
The least significant bits are defined to start at bit 32 or 52-2*K, whichever number is the greatest, and go to bit 51-K.

c) Field_3 is defined to go from bit 32-K to bit 31.

d) Field_1R is defined to be Field_1 with the bits collected in reverse order.

e) The K-bit hash value is obtained from taking the bit-wise exclusive-OR of the fields Field_1R, Field_2 and Field_3.

Here is the PASCAL definition:

```
const K = {value between 5 and 17};
type bitfield:   array[0..31] of 0..1;
     virtpageno: array[0..51] of 0..1;
function hash(VPN: virtpageno): bitfield;
var i,j,m,n: integer;   F1,F2,F3,H: bitfield;
begin
   j := 51-K;  m := 31-K;  n := 52-K;
   for i := 31 downto 32-K do
   begin
       F1[i] := VPN[n];
          n := n + 1;
       F3[i] := VPN[i];
       if j >= 32
       then begin
              F2[i] := VPN[j];
                 j := j-1;
            end
       else begin
              F2[i] := VPN[m];
                 m := m-1;
            end;
   end;
   for i := 0 to 31-K  do H[i] := 0;
   for i := 32-K to 31 do H[i] := F1[i] xor F2[i] xor F3[i];
   hash := H;
end {hash};
```

3.4.6 Page Faults

The virtual to physical address translation either succeeds in finding the physical location corresponding to the virtual location, or it fails. Failure gets reported to software as a page fault trap. Operating system software is responsible for making room in physical memory and bringing in the virtual page from secondary storage and returning control so that the instruction originally causing the page fault can now make progress. It is the responsibility of operating system software to maintain data structures that allow it to locate virtual pages on secondary storage. It is the responsibility of hardware to recover the machine state, on detecting a page fault, that allows the current instruction to be restarted (or "step-restarted", see chapter 7) transparently to the user program after the page fault has been resolved by operating system software.

Certain programs that make up the operating system software cannot themselves sustain page faults during their execution either for inherently logical reasons (e.g. the driver for the paging device must always remain in physical memory), or for timing-dependent reasons. Guaranteeing that all virtual pages accessed by such operating system software are present in physical memory ("resident") is itself the responsibility of operating system software. There are no architecturally defined data structures that prevent physical pages from being swapped out.

PROCESSOR REGISTERS AND MACHINE STATE — CHAPTER 4

This chapter describes the various registers that are available for use by Vision software and also the processor registers that support the VISION addressing structure.

Vision offers 16 programmable registers of 32 bits for general program use and 8 registers of 64 bits specifically to hold logical addresses. VISION's vector processing capability is supported through 8 vector registers, each capable of holding up to 256 elements, each element being capable of holding a 128-bit IEEE floating point number or any smaller data type.
Various status registers record conditions or modify behavior of instruction execution.

The processor registers described below all exist outside the address space. Registers do not have addresses. No "normal-looking" writes to memory will in fact write to a register.
All changes to register values are explicit, i.e. through use of instructions such as MOVEf/tSP or as side-effects of instruction execution as explicitly provided for in chapter 6.
This allows low-end VISION hardware implementations to implement some of the less frequently used processor registers by using hardware-reserved memory locations: locations in physical memory that are not mapped into virtual or logical address space.
Chapter 9 provides details on where hardware-reserved memory should be located.

4.1 General/Index Registers

In Vision mode, software has access to 16 registers X0, X1,.., X15, each of which is 32-bits wide. These registers can be used for general expression evaluation, for passing parameters to procedures, for allocation of local variables in a procedure and for holding index values in address calculations.
Registers can be used singly, in pairs or quads to hold values comprising 32, 64 or 128 bits.
The registers are not typed. Their contents are interpreted according to the type of operation being performed.

4.2 Base Registers

In Vision mode, software has access to 8 registers B0, B1,..,B7, each 64 bits wide and capable of holding a logical address. Registers B6 and B7 are better known as Q and S, respectively; Q and S support stack addressing and the calling mechanism described in chapter 5. The base registers can be loaded and manipulated as detailed in section 6.2.5.

4.3 Program Counter

The Vision program counter is a 64-bit register P. It contains a 64-bit logical address that points to an instruction in the current code object. The value of P changes as instructions are executed; normally, P is incremented to point at the next instruction in sequence. Branches and other transfer of control instructions will cause P to change explicitly. External events such as external interrupts may preempt the currently executing program and force execution to continue at a well-defined place. Transfers of control other than branches will leave a record of the old value of P; this is detailed in chapter 5.

4.4 Status Registers

The status registers combine various fields of machine state in a convenient and compact form. There are six 32-bit words of status, divided into four logical groups:

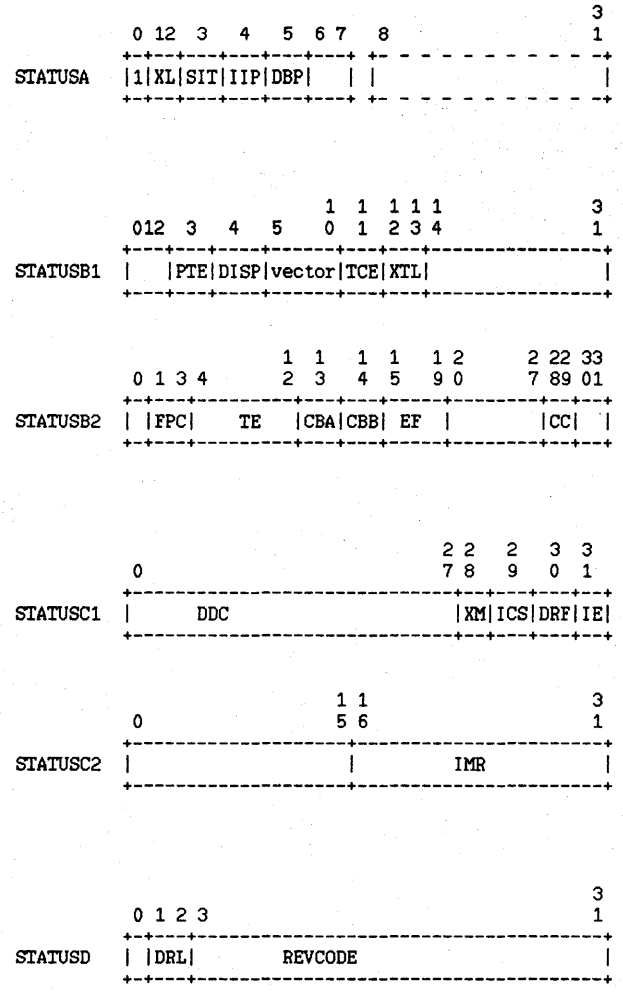
1) STATUSA -- The STATUSA register represents the part of the machine state that is local to the execution of the current code object. STATUSA is shown as an 8-bit register left-justified in a 32-bit word. STATUSA[0..7] is stored in the procedure marker for an external procedure call and restored on the corresponding EXIT. Similarly, STATUSA is stored in an interrupt marker on an interrupt (external or internal), and restored on the corresponding IEXIT.

2) STATUSB -- The STATUSB register represents the part of the machine state that is local to a task activation or the activation of an interrupt handler. For tasks not currently active (suspended by an interrupt or by the DISP instruction) the value of STATUSB is stored in the interrupt marker for that task. STATUSB is restored from the interrupt marker on IEXIT (or LAUNCH). STATUSB is initialized to a known value on an external interrupt and on entering the dispatcher.

3) STATUSC -- The STATUSC register represents the part of the machine state that is local to a CPU/processor. This means that the STATUSC register is replicated for each processor in a shared-memory multi-processor system and its values are specific to each processor in that system.

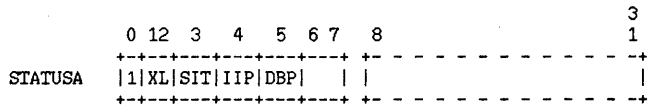
4) STATUSD -- The STATUSD register represent the most global of all status information. This status is shared among all CPUs/processors in a shared-memory multi-processor system. This implies that a change to STATUSD must be communicated synchronously to all processors.

The next page shows an overview of all status registers with their constituent fields.



4.4.1 STATUSA Register

4.4.1.1 Format



STATUSA -- Procedure Status

4.4.1.2 Summary

Field Name		Bit Positions
XL -	Execution Privilege Level	1-2
SIT-	Single Instruction Trace	3
IIP-	Instruction In Progress	4
DBP-	Debug Breakpoint Pending	5

4.4.1.3 XL - Execution Privilege Level

This specifies in which of the four protection rings the processor is currently executing. Ring (or Level) 0 is most privileged and ring 3 is least privileged.

Change of execution privilege level is accomplished by one of three instructions. The CALLX instruction can grant extra privilege or leave it the same, but will never take privilege away. The EXIT instruction (or IEXIT) may take privilege away or leave it the same, but will never grant extra privilege.

The CALLX instruction determines the privilege level of the target code object from the execute access right field in the OD for the code object. This field identifies the least privileged level at which code within that object may execute. Calls to that object automatically begin execution at that level, or the level of the caller, whichever is more privileged.

The EXIT instruction restores the privilege level of the caller if caller and called procedure reside in different code objects. It does this by extracting the privilege level of the caller from the procedure stack marker after applying a consistency check on it. EXIT may never grant the caller more privilege than the current privilege level.

4.4.1.4 SIT -- Single Instruction Trace

This bit can only be set as a side effect of an external EXIT. When this bit is found set at the completion of an instruction, a trap is taken. This allows tracing the execution of software one instruction at a time. The SIT bit is automatically cleared as part of the trap initiation.

4.4.1.5 IIP -- Instruction In Progress

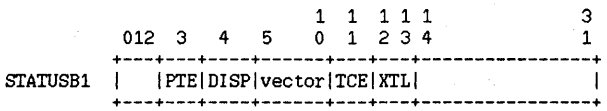
Some instructions can be interrupted before they are completed. These instructions are composed of "steps", as detailed in chapter 6. When such an instruction is in fact interrupted, the IIP bit is set and certain information is pushed onto the stack. When the instruction is resumed after an IEXIT, finding IIP set indicates to hardware that the instruction is being resumed rather than restarted, and hardware acts accordingly. A similar situation arises when a "step-restartable" trap occurs in an interruptible instruction. The EXIT instruction that concludes the trap handler will restore the IIP bit and this indicates to hardware that some instruction steps have already been completed.

4.4.1.6 DBP -- Debug Breakpoint Pending

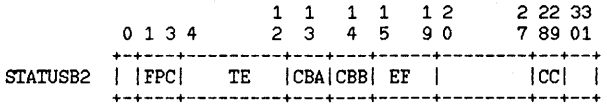
DBP is set to one by hardware whenever an instruction modifies one of the bytes in the system breakrange or the task breakrange (subject to the Debug Ring Level described in section 4.4.4.3). If the DBP bit is found set at the completion of an instruction, a debug trap is taken.
The DBP bit is cleared as part of the debug trap initiation. See section 4.8 for more detail.

4.4.2 STATUSB -- Task/Interrupt Status

4.4.2.1 Format



STATUSB -- Task/Interrupt Status - Privileged



STATUSB -- Task/Interrupt Status - User Accessible

4.4.2.2 Summary

Field name	Word	Bit Positions
PTE -- Procedure Trace Enable	B1	3
DISP -- Dispatcher Running Flag	B1	4
vector-- Vector Register Control	B1	5-10
TCE -- Task Clock Enable	B1	11
XTL -- EXIT threshold level	B1	12-13
FPC -- IEEE Floating Point Control:		
Projective/Affine Mode	B2	1
RM - rounding mode	B2	2-3
TE -- Trap enables:		
Floating Point Operations:		
FLDVDZE -- Divide by zero	B2	4
FLOVFE -- Overflow	B2	5
FLINVE -- Invalid Operation	B2	6
FLUNFE -- Underflow	B2	7
FLINXE -- Inexact Result	B2	8

Field name	Word	Bit positions
Integer Operations:		
INTDVDZE-- Divide by zero	B2	9
INTOVFE -- Overflow	B2	10
Decimal Operations:		
DECDVDZE-- Divide by zero	B2	11
DECOVFE -- Overflow	B2	12
CBA -- Conditional Break Enable, A	B2	13
CBB -- Conditional Break Enable, B	B2	14
EF -- Exception Flags:		
OVF -- Overflow	B2	15
DVDZ -- Divide by zero	B2	16
Floating Point Operations:		
FLINV -- Invalid Operation	B2	17
FLUNF -- Underflow	B2	18
FLINX -- Inexact Result	B2	19
CC -- Condition Code	B2	28-29

4.4.2.3 PTE -- Procedure Trace Enable

When PTE has the value one, all procedure calls (CALL, CALLX and BRX) will cause a restartable trap, subject to the value in the DRL field in STATUSD.

4.4.2.4 DISP -- Dispatcher Running Flag

DISP is one when the dispatcher is running. The dispatcher runs on the bottom of the Interrupt Control Stack. Because DISP is part of STATUSB, it gets saved in the Interrupt Marker (and then cleared). This makes it possible for IEXIT to determine, as it is removing an interrupt marker, whether to return to another interrupt handler or whether to resume or restart the dispatcher. The DISP bit in the Interrupt Marker must not be modified by software.

4.4.2.5 vector -- vector register status

The vector capability of the VISION architecture is described in section 4.11.

4.4.2.6 TCE -- Task Clock Enable

When this flag is set to one, the task clock will be running. This means that the task clock value will be incrementing itself at a fixed rate. When TCE is clear, the value of the task clock will not change unless explicitly changed by program control. See section 7.2.

4.4.2.7 XTL -- EXIT Threshold Level

On executing the EXIT instruction, hardware checks to see if the execution privilege level is being changed to a privilege level less privileged than the value in the XTL field. If so, EXIT will instead trap out and transfer control to the INSXTL trap handler. See section 5.3.6.

4.4.2.8 FPC -- IEEE Floating Point Control

The IEEE floating point standard govens floating point operation in Vision mode. Refer to that publication for further detail.

Projective/Affine mode:
- 0: Projective mode
- 1: Affine mode

This affects the way infinity is treated.

Rounding mode:
- 0: round to nearest unit (breaking ties by rounding to even value)
- 1: round toward plus infinity
- 2: round toward zero
- 3: round toward minus infinity

4.4.2.9 TE & EF -- Trap & Exception Flags

The TE (trap enable) and EF (exception flags) fields are for a number of conditions which can occur during the execution of an instruction which may require special handling by the user program. Five of these relate to floating point operations

and are defined by the IEEE standard. Integer data types follow the normal rules for 2's complement arithmetic whereas decimal data types are described in section 6.3.
When an exception condition occurs, the trap enable bit is consulted to determine whether the exception should result in a trap. If the trap is disabled, the corresponding exeption flag is set. The exception flags act as "sticky bits" that record the occurrence of exception conditions at any time during which traps were disabled. Hardware never resets the exception flags; nor will an exception flag, when set, cause a trap when the trap is subsequently re-enabled.

4.4.2.10 CBA & CBB -- Condition Break Enable Flags

The CBA and CBB fields control the operation of the CHECKA and CHECKB instruction, respectively. CHECKA causes a trap when CBA is found set, and acts as a NOP when CBA is clear. In addition, instructions are provided to set or test the value of the CBA and CBB fields.

4.4.2.11 CC -- Condition Code

The condition code field in the STATUSB register reflects the result of the most recent test or compare operation.
The four values of CC are indicated in this document mostly through mnemonics (CCG,CCE,CCL,CCU) as follows:

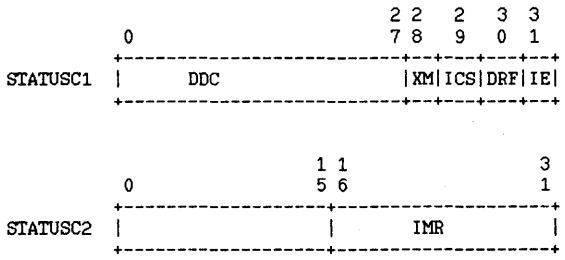
CC	mnemonic	stands for:
0	CCG	condition code "greater"
1	CCL	condition code "less"
2	CCE	condition code "equal"
3	CCU	condition code "unordered"

On a Compare instruction, CC is given the value "CCG" if the first operand is greater than the second operand. With a Test instruction, CC is given the value "CCG" if the first (only) operand is greater than zero. Similarly for CCL and CCE.
CC is given the value CCU only on floating point compare or floating point test when the two values being compared are "unordered" with respect to each other according to the IEEE floating point standard.

Some instructions other than Tests and Compares cause CC to get set. These instructions are explicitly designated in chapter 6. All other instructions leave the condition code field unaltered.

4.4.3 STATUSC -- CPU Status

4.4.3.1 Format



4.4.3.2 Summary

Field name	Word	Bit Positions
DDC -- Dispatcher Disable Count	C1	0-27
XM -- Execution Mode	C1	28
ICS -- On Interrupt Control Stack	C1	29
DRF -- Dispatcher Request Flag	C1	30
IE -- Interrupt Enable/Disable	C1	31
IMR -- Interrupt Mask Register	C2	16-31

4.4.3.3 DDC -- Dispatcher Disable Count

The value of DDC is incremented by the PSDB instruction and decremented by the PSEB instruction. The function of DDC is to monitor when it is appropriate to enter the dispatcher. As long as DDC is non-zero, the dispatcher is not permitted to run.

4.4.3.4 XM -- Excecution Mode

XM defines the current mode of execution. When XM has the value zero, hardware executes in Vision mode. When XM has the value one, hardware executes in HP3000 mode. The value of XM changes as a consequence of executing any of the variants of SWITCH. Also, any transfer of control to the Interrupt Control Stack from HP3000 mode will cause XM to be zero.

4.4.3.5 ICS -- On the Interrupt Control Stack

This flag is set to one when execution switches to the Interrupt Control Stack, e.g. on an external interrupt. This flag is cleared on an IEXIT that returns control to a task.

4.4.3.6 DRF -- Dispatcher Request Flag

This flag is set to one by the DISP instruction if access to the dispatcher is temporarily deferred. In order to enter the dispatcher, the following conditions must be simultaneously satisfied:

a) STATUSC.DDC = 0
b) STATUSC.XM = 0
c) STATUSC.ICS = 0
d) STATUSC.IE = 1

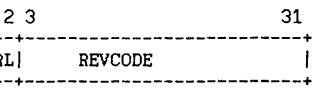
When DRF is one, any action that causes either condition (a) or (b) or (c) or (d) to become satisfied will reexamine all four conditions; if all four are now found satisfied, the dispatcher will be entered.

4.4.3.7 IE & IMR -- Interrupt Enable/Disable & Mask Register

The IMR field determines which external interrupts are allowed to alter the flow of control. If IE=0, no interrupts can alter flow of control, overriding the value of IMR. See section 7.2 for more detail.

4.4.4 STATUSD -- Computer Status

4.4.4.1 Format



4.4.4.2 Summary

Field name		Bit positions
DRL	-- Debug Ring Level	1-2
REVCODE	-- SPU Revision code	3-31

4.4.4.3 DRL -- Debug Ring Level

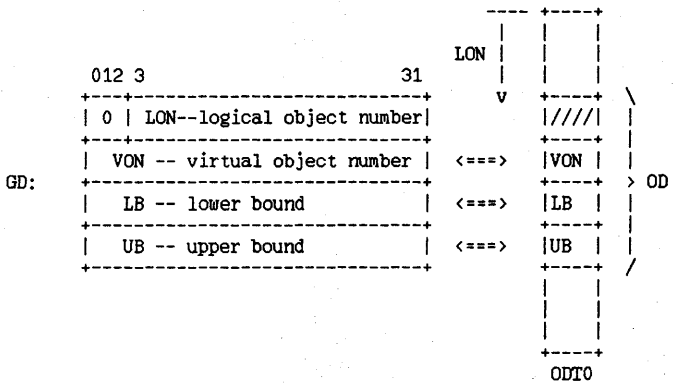
This field defines which execution ring levels are subjected to the debug breakrange traps. Specifically, the System Breakrange trap and the Task Breakrange trap as well as the procedure trace trap are enabled only at ring level DRL and less privileged rings. Typically, DRL equals 1 in order to allow I/O to proceed at full speed even while debugging at all other privilege levels. However, highest privilege code can be debugged too, by setting DRL to zero.

4.4.4.4 REVCODE -- SPU Revision Code

This is a unique number assigned at the factory for each SPU or version of an SPU which differs significantly from a prior version. The most significant 13 bits are unique for each SPU; bits 16-31 identify each significant revision of the SPU. This is NOT a serial number: it is not unique for each unit.

4.5 Group Descriptors

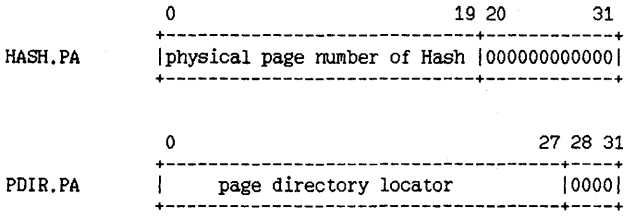
Address translation of logical addresses to virtual addresses requires the hardware to be able to locate the eight Object Descriptor Tables. Eight Group Descriptors (GD) serve to locate these ODTs. The Group Descriptor for group zero is kept in a processor register; it can be thought of as an extension of the STATUSD register. The Group Descriptors for groups one through seven exist in the Task Control Block (TCB) of the currently executing task. All eight Group Descriptors have the same format, which is described below. The TCB is located by hardware through the TCB.VA virtual address; TCB.VA is kept in a processor register that can be considered as an extension of STATUSC. The format of the TCB is described in section 4.7.



The last three words of the Group Descriptor contain the Virtual Range that locates the ODT in virtual space. The first word of the Group Descriptor contains the Logical Object Id of an object in Group 0. This LOI exists for the purpose of allowing certain hardware TLB organizations.
Operating system software must ensure that the Virtual Range contained in the OD identified by LOI is identical to the virtual range contained explicitly in the GD.

4.6 Virtual address translation registers

Hardware must be able to locate the Hash table and the Page Directory in physical memory; this is necessary for being able to do virtual to physical address translation.
Two processor registers serve this purpose; they can be thought of as an extension of STATUSD.



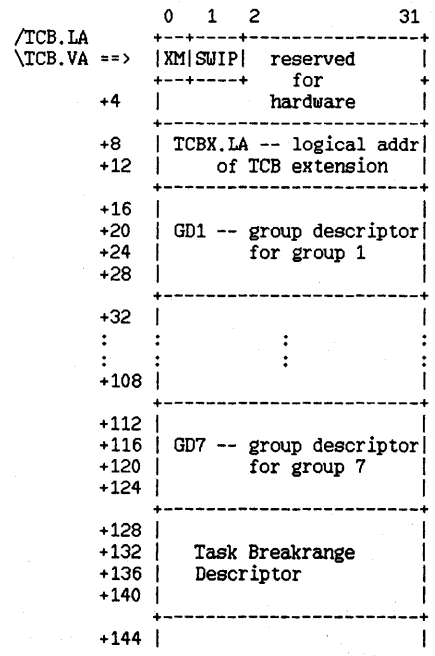
HASH.PA contains the physical address of the Hash table; this address must be page aligned.

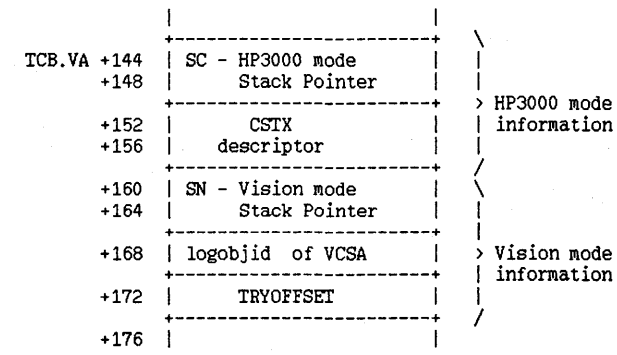
PDIR.PA contains the physical address of the Page Directory PDIR; this address must be aligned on a 16-byte boundary.

4.7 Task Control Block

Hardware needs a certain amount of information in order to execute the current task. This information is stored in the Task Control Block (TCB), located by a register TCB.VA.
This TCB.VA register can be thought of as an extension of STATUSC. TCB.VA must be a multiple of 16. The length of the TCB is 176 bytes. Also, the TCB must be memory resident.

A 64-bit register TCB.LA accompanies TCB.VA; operating system software is responsible for ensuring that the logical address TCB.LA does in fact translate into the virtual address TCB.VA. Moreover, the logical address TCB.LA must have a zero group selector. Hardware implementations are free to use either TCB.LA or TCB.VA to locate the TCB.
A task switch is accomplished by Dispatcher software through simultaneously changing the TCB.VA and TCB.LA registers.





XM -- execution mode of the task. On IEXIT to this task, execution mode STATUSA.XM is set to this value.

SWIP -- switch in progress. This bit is used by IEXIT when a mode switch could not be completed.

TCBX.LA- The logical address of a TCB extension for use by software.

GDi -- Group Descriptors. The format of a Group Descriptor is described in section 4.5.

Task Breakrange Descriptor.
This descriptor is described in section 4.9.

SC -- Logical address of top-of-stack of the HP3000 mode stack used to initialize S on IEXIT.

CSTX Descriptor.
The descriptor locates the CSTX used in HP3000 mode. Its format is the same as described in section 4.10.

SN -- Logical address of top-of-stack of the Vision mode stack used to initialize S on IEXIT.

logobjid of VCSA.
The logical object id of the logical object in use as the Vector Context Save Area. See section 4.11.

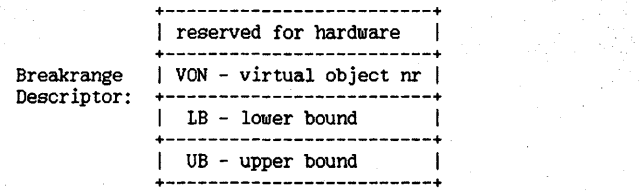
TRYOFFSET.
The stack offset saved by the TRY instruction.

4.8 Breakranges (System and Task)

The VISION architecture supports two breakranges: a System Breakrange and a Task Breakrange. A breakranges involves a range of virtual addresses. When properly enabled, the breakrange will cause a trap to occur at the completion of any instruction that overwrites any byte within the breakrange. This debug aid is discussed in more detail in section 5.3.

The Descriptor for the System Breakrange can be thought of as an extension of STATUSD: once installed and properly enabled, the System Breakrange will cause a trap whenever any task on any processor in a shared-memory multi-processor writes to any byte within that Breakrange. The Task Breakrange Descriptor can be thought of as an extension of STATUSB; it is located in the Task Control Block. When properly enabled, the Task Breakrange will cause a trap when this particular task writes to any byte within this Breakrange.

A Breakrange Descriptor is a 16-byte descriptor with the format shown below.

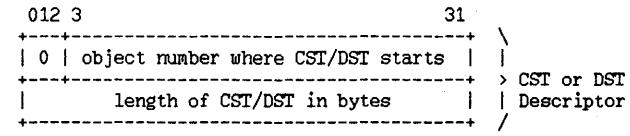


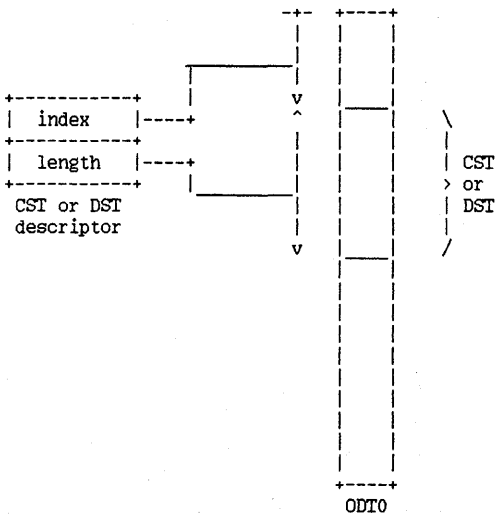
4.9 Interrupt Control Stack location

The Interrupt Control Stack (ICS) is the environment in which hardware interrupt handlers run. Several trap handlers also run on the ICS. This environment is described more fully in section 7.6.
The ICS is a logical object in group 0. When an interrupt is acknowledged, Q is made to point to a location on the ICS just beyond the Dispatcher Marker (see 7.2). This location is called QI. The logical address of QI is kept in a processor register that can be thought of as an extension of STATUSC.

4.10 CST and DST descriptors

HP3000 mode requires a Code Segment Table (CST) as well as a Data Segment Table (DST) to complete its addressing environment. Both CST and DST are tables of ODs. These tables are actually contained within the ODT for group zero. Both CST and DST are found through processor registers containing 64-bit descriptors as indicated below:





4.11 Vector Processing

VISION offers rich support for vector operations in order to significantly increase performance on common array and matrix operations in technical applications.
This section should be read in conjunction with section 6.4.

4.11.1 Vector Registers

A task can address 8 vector registers VR0, VR1,.., VR7. Each VR consists of N elements, numbered 0 through N-1, where N is an implementation-dependent quantity no greater than 256. Each element is 128 bits wide, and can contain values of data types comprising 32, 64 or 128 bits.

Not all N elements in a Vector Register need be filled with data; software may load vectors with fewer elements into a VR.

Vector processing hardware may support up to 15 banks of eight vector registers each. Only one bank is addressable by a task at any time. Multiple banks allow multiprogramming of tasks using vector registers without excessive performance penalty in saving vector register contents on a task switch.

4.11.2 Vector Mask Registers

A task can address 4 vector mask registers, VMR0,..,VMR3. Each contains 256 bits. A bit in a vector mask register corresponds to an element in a vector register.
Each vector instruction designates a vector mask register to govern execution of that instruction. Elements in the vector register corresponding to clear bits in the mask register do not participate in the vector instruction; no results are stored in the vector register element, the original value of the element does not change, and under no circumstance can traps occur for that element.
The values in the mask registers can be created and manipulated through special instructions including vector compare.

4.11.3 Vector Length Register

Various means exist for software to let hardware know how many elements in a vector register should be regarded as meaningful and how many elements should be operated on in a given vector instruction.

4.11.4 Vector Context Save Area

Because vector registers contain a large amount of information, it is not desireable to save all this state on an external interrupt. Interrupt handlers must therefore refrain from using vector instructions.
Vector Registers (and vector mask registers) are saved either explicitly by operating system software or automatically when another task executes a vector instruction that uses the same vector register bank. When vector registers are saved, they are saved on behalf of the task that last used them. Part of the task's context includes the Vector Context Save Area (VCSA), located through the task's TCB, memory resident, and large enough to receive all vector register values.

4.11.5 Vector Processing: Operation

Operating system software may allow any task or any number of tasks to execute vector instructions. VISION implementations may have special purpose hardware, referred to as a Vector Processor (VP), which will improve the performance of the vector operations.

The VP will contain some amount of memory, organized as one or more banks. A bank generally contains all of one task's vector related context, including vector registers, vector mask registers, vector length register, etc. This context is quite large, and requires attention in order to maintain fast interrupt response, ability to multiprogram, and minimal impact on pure scalar tasks.

A task must be assigned a special region of main memory called the Vector Context Save Area (VCSA) before it can execute any vector instructions. Additionally, a task must be assigned a bank in order to use the VP. A task may be denied access to the VP, in which case all of its vector activity occurs through its VCSA; this limits the speed of vector instructions to the speed of memory access.

More than one task may be assigned the same bank. If this occurs, hardware is responsible for saving/restoring the context when a vector task enters execution and a different task's context is in the first task's assigned bank. The hardware will save the first task's vector context into its VCSA, and reload the bank from the new task's VCSA.

Note that if a task which does not use the VP (a scalar task) starts to run, the interrupted vector task's VP context will be protected but remain in the hardware registers. If control returns to the same vector task, the VP context switch will have been avoided entirely. If control returns to a different vector task which has been assigned the same bank, the context switch will occur when the new vector task attempts to execute its first vector instruction.

The vector processor may contain 0 to 15 banks of context. The operating system will designate which bank (if any) a task is allowed to use. A group of tasks assigned the same bank will only compete among themselves for that VP context.

4.11.6 VP Management - Vector Context Save Area

Use of the vector processor by a task is controlled by six bits in STATUSB, altered only by operating system software. Four of these bits specify the bank number, and two specify permission level.

If the bank number is not 0, the number specifies which hardware bank of VP context is to be used by this task.

If the bank number is 0, all vector activity takes place through the VCSA instead of the vector registers. The hardware context consumed is effectively zero, but all operand/result (including VR) accesses proceed at memory speeds. The performance in this mode will be degraded, but will normally be faster than equivalent scalar code. Through this feature operating system software can allow a low priority vector task to execute, while reserving the vector hardware for some very important task.

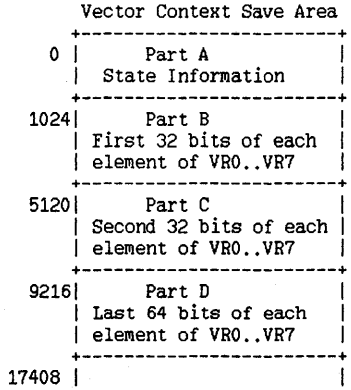
The permission bits must be made non-zero by operating system software if a task is to be allowed use of the vector processor. Before operating system software grants permission, a portion of the VCSA must be locked into memory. This is to:
a) prevent phantom page faults,
b) ensure that a place exists in which to save context should power fail,
c) provide a place to simulate VRs should the implemented hardware be insufficient.

Since the VCSA is quite large, levels of permission exist to restrict a task to using only certain ranges of precision and thus permit operating system software to only lock in a portion of the VCSA. See table below.

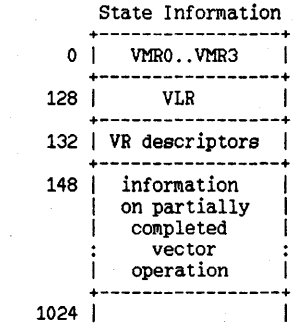
Permission bits value	Meaning
00	No VP use allowed by this task.
01	VP usage restricted to vector data types <= 32 bits; hardware assumes that parts A & B of the VCSA are locked into memory.
10	VP usage restricted to <= 64 bits of vector data; hardware assumes that parts A, B & C of the VCSA are locked into memory.
11	VP usage unrestricted; hardware assumes the entire VCSA is locked into memory.

If insufficient permission has been granted when the task attempts to execute a vector processing instruction, the INSVPPERM trap occurs. Operating system software will typically lock in the additional portion of the VCSA and then redispatch the task.

If the permission bits are not 0, the VCSA logical object id in the TCB identifies the VCSA for that task. The entire VCSA is a 1/4 + 8*(VR len*4/1024) page area. (For 128-element VRs, this is 4 1/4 pages.) Conceptually, the contents of the VCSA at the time of an interrupt are as sketched below.



Note that the VCSA is sketched assuming 128 element VRs. The State Information in Part A is detailed below:



Only the portion of a VR save area corresponding to the VR's active length is meaningful. Bytes 0-127 of State Information contain the VMRs as they exist at the time of interrupt or task switch; bytes 128-131 contain the VLR. Bytes 132-147 describe the width and active length of each VR.

A vector instruction may be interrupted before it is complete. In this case, information sufficient to transparently resume the partially completed operation is stored in bytes 148-1023 of the State Information area.

As explained above, when an interrupt occurs the VCSA is not immediately updated. Two instructions are available to force immediate updating of the VCSA. The first is UVCSA (Update Vector Context Save Area). This is a non-privileged instruction, and contains a two-bit option field to force saving of either VRs, State Information, or both. UVCSA may be used by a trap handler to isolate the elements and operation causing the vector trap. The second instruction, PUVCSA (Privileged Update of Vector Context Save Area) allows specification of the vector bank to be saved, independent of the vector bank in use by the current task. Operating system software may issue this instruction if it is waiting to transfer control to a vector task and wishes to minimize that task's startup time.

MACHINE MODEL — CHAPTER 5

This chapter discusses several topics relevant to the operation of Vision mode brought together under the heading "machine model" because they contribute to a more coherent picture of the model than might be gleaned from the detailed description of individual instructions in chapter 6.

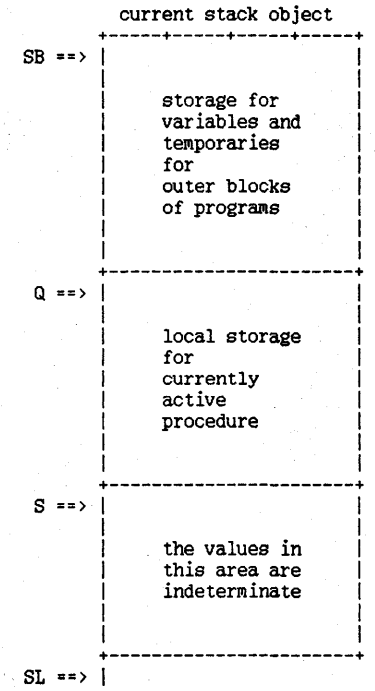
These topics are:

a) stack and stack markers
b) procedure linkage
c) debug support
d) list of supported data types

5.1 The Vision Stack

The Vision mode stack is primarily used for procedure linkage, parameter passing and allocation of local and temporary variables for procedures, to the extent that the registers X0..X15 do not suffice for this. The stack is also used as an implied place to store things, such as parameters when traps are taken or when internal and external interrupts occur.
The registers Q and S always point to the stack. The upper 32 bits of Q and S are identical; they identify the current stack object. Refer to section 2.3.6.
The stack is totally word-oriented: the stack object is word-aligned in virtual address space, and the logical addresses Q and S are both at all times multiples of four. The length of the stack object is a multiple of four as well.

S points to what is called top of stack. Q points to what is called the local stackframe.
S changes under the effect of explicit PUSH and POP instructions and their variants (e.g. DUP, EXTEND, DELETE), and implicitly on traps and interrupts.
Q changes only on procedure calls (CALL, CALLX), on procedure returns (EXIT), through a MOVEtSP, and implicitly on traps.



The logical address corresponding to the first byte in the stack object is denoted by SB; similarly, SL denotes the first byte beyond the stack object. The following relationship among these registers is always guaranteed:

SB <= Q <= S <= SL

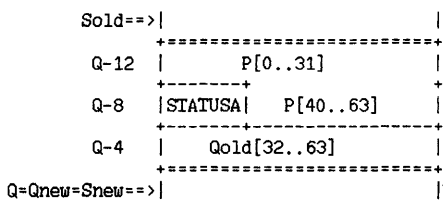
The necessary checks to guarantee this are performed when S or Q change. The area between S and SL is indeterminate. This means that its contents cannot be predicted. Stack bounds checking is more restrictive than bounds checking on ordinary objects. All memory accesses through logical addresses with an LOI equal to that of current Q are checked against S as the upper bound rather than against SL. The EXTEND instruction will increase S without explicitly initializing the area between the previous S and the new S; the newly accessible part of the stack will have contents that are unpredictable.

5.1.1 Procedure Stack Marker

The procedure stack marker is a 3 * 32 bit entity that is pushed on the stack as part of procedure call (CALL, CALLX); at this time a new stack frame is created by setting Q to point to the area in the stack immediately beyond the marker. The marker preserves the information necessary for EXIT to restore the old environment, specifically, the old value of Q.

5.1.1.1 External Procedure Stack Marker

External procedure calls (CALLX) push a 3-word marker as shown:



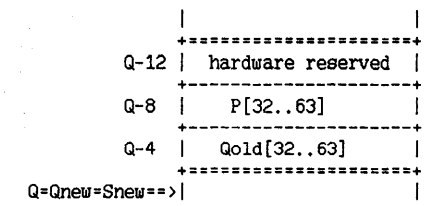
The value of P in the marker is the logical address of the instruction following the CALLX. The size of code objects is restricted to 2**24 bytes so that P[32..39] = 0.
The value of STATUSA contains the privilege level at which the caller ran. STATUSA[0] = 1. Bits STATUSA[1..7] will always be clear when executing CALLX.

Traps and external interrupts also push an external procedure marker. Here the value of P in the marker is the logical address of the instruction that needs to be executed after returning from the handler. The IIP bit and the DBP bit in STATUSA may be set.

An SIT value of one must never be pushed as part of STATUSA in the external procedure marker. Instead, it may only be set in the marker explicitly by software. Such software will use EXIT to cause the SIT flag to get set in the calling program.

5.1.1.2 Local Procedure Marker

The local procedure call (CALL) never causes the execution level to change, nor can it change the current code object. The stack marker for a local call can therefore be simpler than for an external procedure call:

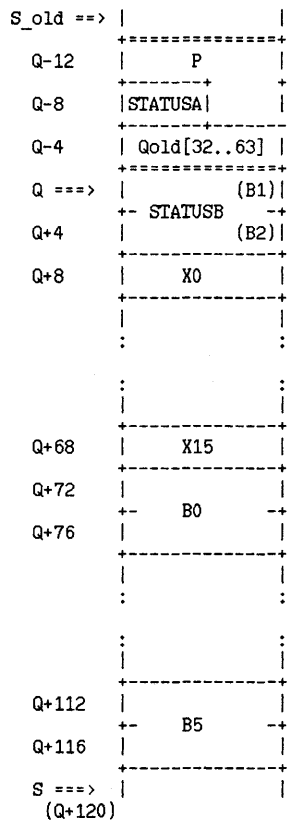


Note that EXIT can unambiguously distinguish between a local procedure marker and an external procedure marker: (Q-8)[0] has the value 0 for a local marker and the value 1 for an external marker. The bits P[32..39] are zero.
Note that both markers occupy the same amount of space.

5.1.2 Interrupt Marker

An interrupt marker is pushed on the stack as part of servicing (acknowledging) an interrupt (external or internal), or when a task transfers control to the dispatcher (with the DISP, PSEB or ENABLE instruction), or upon executing SWITCH.
The marker contains the following information (see IEXIT):

1. the information in an external procedure marker
2. STATUSB
3. general registers X0-X15
4. base registers B0-B5

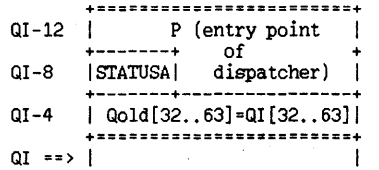


5.1.3 Dispatcher Marker

The Dispatcher is a piece of operating system software that selects the next task to run, in preparation for a task switch. The dispatcher is entered automatically under certain conditions carefully controlled by the DISP, PSDB, PSEB, ENABLE and DISABLE instructions. The dispatcher code is entered through a special procedure marker called the Dispatcher Marker, which is at the base of each Interrupt Control Stack (ICS). This marker is never removed, in contrast to all other stack markers. Entering the Dispatcher is similar but not identical to an EXIT through the Dispatcher Marker: Q will not change, S will get set to Q.

The Dispatcher Marker contains:

1. Program counter for entry point of Dispatcher
2. STATUSA



Just before the Dispatcher starts running, the IEXIT instruction will initialize STATUSB to the value DispatcherStatusBInit, which has the following fields:

Field	Value	Full name of field
DISP	= 1	Dispatcher running flag
PTE	= 0	Procedure Trace Enable
vector	= 0	vector control
XTL	= 3	EXIT threshold level
FPC	= 0	IEEE floating point control
TE	= 0	Trap Enables (none enabled)
EF	= 0	Exception Flags (none detected)
CBA	= 0	Conditional Break Enable, A
CBB	= 0	Conditional Break Enable, B
CC	= 0	Condition Code

5.2 Procedure Linkage

The Vision instruction set provides several forms of procedure call. Their detailed description is given in chapter 6. The instructions CALL and CALLX both create a new stackframe by pushing a procedure marker and then updating Q and S. CALL and CALLX differ primarily in the way they arrive at the target address. This is detailed in section 5.2.1. They also lay down a different marker because EXIT must be able to return to the appropriate environment in each case.

A procedure must be wholly contained in a code object; it cannot span several code objects. Conversely, a code object may have many procedures in it. Procedures within the same code object may call each other using the CALL instruction; this is the fastest procedure call. Code objects are paged, their maximum size is 2^24 bytes; hence there is opportunity to group large numbers of procedures into a single code object. The following reasons may limit in practice the number of procedures that are combined into a single code object:

a) procedures in a code object run at the same privilege

b) if a procedure in a code object is recompiled, all of the code object must typically be relinked

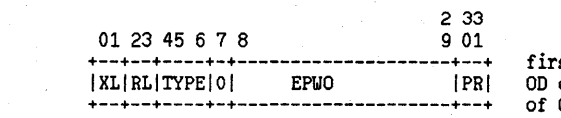
c) the procedures in a code object are not protected from one another; a procedure may jump in the middle of another procedure without being caught

d) often-used procedures can be put in a separate code object and shared among many user programs; this trades off space and link time versus CALLX overhead

Code objects can have multiple external entry points; however, this requires the approach outlined in section 5.2.2.

5.2.1 Entry Point Evaluation

The external procedure call CALLX has as its single operand the logical object id (LOI) of the target code object. This LOI is sufficient to determine the location of the target procedure. The LOI uniquely identifies an Object Descriptor (OD) which has the format described in section 2.3.2. The first word of this OD is included on the next page.



where:

TYP - should indicate a Vision mode code object

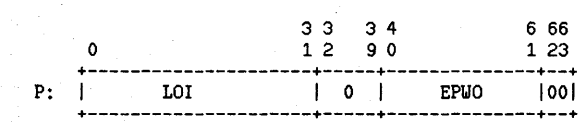
PR - indicates the prerequisite level: the privilege level the caller must already possess before being allowed to complete the procedure linkage

XL - indicates the privilege level at which the target procedure will run

RL - indicates the privilege level required for reading the contents of the code object as data

EPWO - the entry point word offset: indicates the location of the starting point of the target procedure, expressed as a word offset relative to the start of the code object

The new program counter P is constructed from this information as follows:



5.2.2 Multiple Entry Points in a Code Object

It is possible to have multiple external entry points per code object, but only by duplicating Object Descriptors. This means that each external entry point must be associated with a unique logical object id. This is not the same as stating that each external entry point corresponds to a single code object, for these Object Descriptors will share the identical Virtual Range contained in their last three words. These procedures can still call each other freely using CALL instead of CALLX, thus keeping all characteristics of being in a single code object.

5.3 Debug Support

The VISION architecture is rich in features to support debug of software. This section collects these features in one place; however, pervasive object bounds checking has already been covered in chapter 3 as has checking of access rights.

5.3.1 Code Breakpoints

The following instructions are provided and can be inserted into the code stream either at compile time or at run time:

a) BREAK
b) CHECKA
c) CHECKB

Each is capable of generating a trap: BREAK unconditionally, CHECKA and CHECKB conditionally on the setting of an appropriate bit in STATUSB (STATUSB.CBA and STATUSB.CBB respectively). Debug software can use these instructions to provide code breakpoints, code tracing, etc.

5.3.2 Breakranges

The VISION architecture provides two Breakranges that can be used to protect an area from accidental or malicious writes and to trap any software that changes any data anywhere within the Breakrange. The format of these Breakranges is discussed in section 4.8.
The System Breakrange is a Virtual Range kept in a processor register and set by a MOVEtSP instruction. Any write within the Virtual Range causes a breakrange trap when properly enabled.
The Task Breakrange is a Virtual Range kept in the Task Control Block. Any write by that task within the Virtual Range causes a breakrange trap when properly enabled.
Two ways exist to disable the breakranges and totally eliminate their performance impact:

a) setting the DRL field in STATUSD to a non-zero privilege level. This will disable the breakrange for all code at level DRL or more privileged.
b) clearing the "DBE" bit in the PDIR for a particular VPN. Only accesses to pages with the DBE bit set are checked for breakrange traps.

The VISION architecture does not require hardware to run at "normal" speed when accessing an enabled page at an enabled privilege level. Software should anticipate some performance degradation in this situation.

5.3.3 Single Instruction Trace

Software can cause hardware to sequence through user code a single instruction at a time. This is accomplished through the SIT bit in STATUSA. The only way to set this bit is to modify an external procedure marker and then execute EXIT (or IEXIT). EXIT will restore STATUSA with the SIT bit set. This will not have any effect until the next instruction has completed. Any instruction that has SIT set at its beginning will on completion transfer control to the DBSIT trap handler. As part of the trap initiation, SIT will be cleared. In order to determine whether to take the DBSIT trap, hardware only needs to check the SIT bit at the completion of an instruction, provided it delays setting the SIT bit in an external EXIT so as to avoid trapping out on the EXIT instruction itself.

5.3.4 Procedure Trace

If the PTE bit in STATUSB is set and the current privilege level XL is numerically greater (less privileged) than DRL, any execution of CALL, CALLX or BRX will cause the restartable trap DBCALL to be taken.

5.3.5 Object Trace

In the VISION architecture, an access rights violation causes a restartable trap, cf. chapter 7. This trap can be fashioned by software into an object trace capability. Operating system software can manipulate the access rights in the OD of an object such as to cause access rights violations when code attempts to access the object with current privilege level XL numerically greater than DRL. Since this trap is recoverable, execution can resume normally after the debugger has restored the original access rights. The Single Instruction Trace mechanism can be used to regain control at the completion of the instruction in order to reinstate the object trace.

Code object tracing can be performed in a similar manner; here it is the TYPE field in the OD that can be replaced by one specifying a data object. This too is a recoverable trap.

5.3.6 Ring Crossing Trap

This trap is detailed in chapter 6 under EXIT. The STATUSB.XTL (exit threshold level) field can be set to a certain privilege level under operating software control. This allows delaying a certain activity until the task has exited back to sufficiently low privilege on its own accord. Any EXIT (or IEXIT) that drops the current privilege level below (numerically greater) than XTL will cause the Ring Crossing Trap to be taken.

5.4 List of Supported Data Types

Vision mode software supports various data types. Some are supported through a full complement of instructions, some are represented by a few key instructions and/or conversions into fully supported data types. A brief summary follows:

	data types		
# bytes	integer	floating	decimal
1	1	-	-
2	2	-	-
4	4	4F	4D
8	8	8F	8D
16	-	16F	16D

5.4.1 Integers

Vision mode supports both 32-bit two's complement integers and 64-bit two's complement integers with full arithmetic capability including shifts.
16-Bit 2's complement integers are supported through conversion to and from 32-bit integers and by fast detection of 16-bit overflow on 32-bit arithmetic. These conversions are done automatically on loading a 16-bit integer into a 32-bit register or storing a 32-bit register into a 16-bit memory location.
8-Bit unsigned integers are supported through conversion to and from 32-bit integers. The conversion is implied in loads and stores to and from 32-bit registers.

5.4.2 Floating Point

Vision mode performs floating point operations exactly as described in the IEEE floating point standard ("A Proposed Standard for Binary Floating Point Arithmetic", IEEE Task P754).

The standard allows some features to be defined by implementors. These options are defined for Vision mode as follows:

Formats: The formats supported are Single, Double, Quad. These occupy 32 bits, 64 bits and 128 bits, respectively. Quad is defined in the manner provided for "Double Extended" by the Standard.
The 32-bit single precision floating point format has a 1-bit sign, an 8-bit biased exponent and a 23-bit fraction field.
The 64-bit double precision floating point format has a 1-bit sign, an 11-bit biased exponent and a 52-bit fraction field.
The 128-bit quad precision floating point format has a 1-bit sign, a 15-bit biased exponent a 1-bit integer part and a 111-bit fraction field.

Modes: Normalizing mode is provided. All five traps are supported, with individual enable/disable.

Underflow:
Underflow is always checked after rounding.

Operations:
Add, subtract, multiply, divide and conversions between all data types are fully supported by Vision instructions. Remainder, square root, integerize, as well as binary -- decimal conversions must be supported in software.

5.4.3 Decimal

Vision supports packed decimal data types of size 4,8 and 16 bytes. These formats are described in detail in chapter 6.3. Vision also supports conversion to and from packed decimal data of any number of bytes between 1 and 31. External numeric decimal formats are supported through conversion to and from packed decimal.

5.4.4 Logical

Various bit-wise operations are supported on 32-bit quantities. The most significant bit and the least significant bit of a word can be tested individually.

5.4.5 Bit

Instructions to set bits and test bits in arbitrary locations in an object are provided.

5.4.6 Fields

An instruction to deposit a value of an arbitrary number of bits into a 32-bit word is provided.

5.4.7 Byte strings

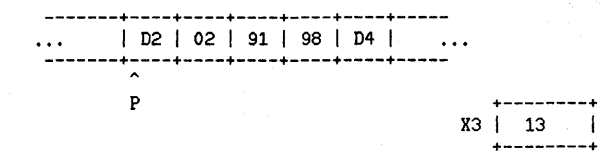
Various instructions to support operations on strings of bytes are provided: move, compare, translate, translate and test.

VISION INSTRUCTION SET — CHAPTER 6

This chapter describes the instruction set available in Vision mode to the programmer who needs to write software in assembly language. Vision compilers will compile standard programming languages to this instruction set.

6.1 Preliminaries

The register P (program counter) contains a 64-bit logical address that points to a variable length entity called an instruction. The instruction is interpreted by hardware and executed; this changes the machine state and/or memory.
The program counter P is among the machine state that changes as a consequence of instruction execution; the default is to advance (increment) P to point to the next instruction in sequence.

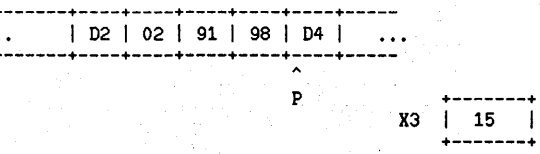


In the sketch above, the pattern !D2029198 is the encoding of a 32-bit instruction that will be rendered here as:

```
ADD4  2, X3
```

This instruction instructs the hardware to perform a 4-byte ADD (32-bit integer addition) on 2 and X3, leaving the result in X3. After executing this instruction, changes will have occurred to X3, to P and to the reference bit "R" in the physical page descriptor for the page containing the instruction. No other machine state nor memory will be affected. In this example, the value of X3 will have been incremented by 2; the value of P will have been incremented by 4 (to skip over the current instruction which occupied 4 bytes).

The situation after executing the "ADD4" instruction will hence be as follows:



The Architecture Control Document does not describe how this effect on the machine state is achieved; different hardware implementations may use quite different means.
In this chapter only the intended effect of an instruction on the machine state and/or memory is given, in a mixture of running text and a Pascal-like algorithm.
Several aspects of instruction execution are so pervasive that they will be described once rather than repeatedly for each instruction.
The most important of these is the way operands are dealt with. Operands are described in section 6.1.1 and their encoding is dealt with in sections 6.1.2 and 6.1.3.
Other such pervasive actions are:

- incrementing the program counter at the end of executing an instruction
- fielding external interrupts at the end of executing an instruction
- detecting page fault on fetching the instruction
- detecting page fault on fetching operands
- setting dirty and reference bits as part of accessing memory
- serving debug traps at the end of an instruction that write into a breakrange;
- checking access rights/bounds violation on any memory access
- trapping on a mismatch of operand and operand attribute

Several of these pervasive actions involve reporting unusual or illegal conditions; refer to section 6.1.8 for more detail.

6.1.1 Operands

In the previous example, "ADD4 2, X3" is an instruction with two operands. The first operand, "2" is called a literal, or a literal operand. This means that the value of the operand can be found right there in the instruction itself.
The second operand, "X3" is called a register operand. The value of this operand is the value currently in the X3 register. The operation called for by the instruction, "ADD4", is an addition; this involves storing the result somewhere. The description for "ADD4" specifies that the result be stored back into the second operand. This means that X3 is not only used to obtain one of the values to be added together, X3 is also designated as the destination for the result.
Note that a register can be a destination for a result, but a literal cannot: the instruction itself must not be changed as a consequence of instruction execution.
There is a third type of operand, called memory operand, which will be detailed in section 6.1.1.3.

In general, a two-operand instruction such as "CMP4" (compare) can have any combination of operand types:

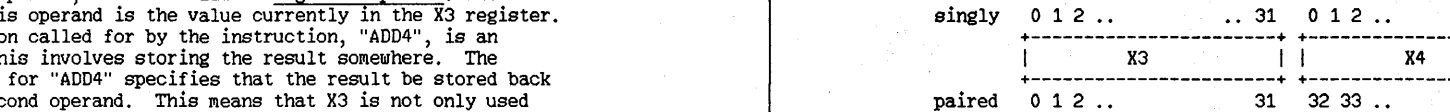
```
CMP4 literal, literal
CMP4 literal, register
CMP4 literal, memory
CMP4 register, literal
CMP4 register, register
CMP4 register, memory
CMP4 memory, literal
CMP4 memory, register
CMP4 memory, memory
```

Each instruction, such as ADD4, determines what combinations of operand types is legal. This is done through attributes as described in section 6.1.5. Illegal combinations are only illegal because of the logic of the situation, such as the inadmissibility of storing into a literal.

The encoding of instructions and their operands is orthogonal, as detailed in section 6.1.2. This means that the encoding places no restrictions on the selection of operand types.

6.1.1.1 Register Operands

Registers X0..X15 are 32-bit entities each. They can be used singly, in pairs, or in quads. A pair of registers can be used as a single 64-bit operand, as follows:



Register pairs always involve consecutive registers Xi and Xi+1; register X15 can form a pair with X0. Register pairs are encoded in an instruction by encoding the first register in the pair.

Register quads can be used to form a single 128-bit operand. Such a quad always involves consecutive registers Xi, Xi+1, Xi+2 and Xi+3; again, register numbers wrap around, such that X0 comes after X15. A register quad is encoded in an instruction by encoding its first register.

A (single) register can also be used to hold operands smaller than 32 bits. Sections 6.1.6 and 6.1.7 go into more detail.

6.1.1.2 Literal Operands

Literal Operands of up to 32 bits long can be encoded in an instruction. When used in instructions that involve data types bigger than 32 bits, such a literal value will be extended to the right size by replicating the "left-most" (most-significant) bit. If the literal represents a two's complement integer this corresponds to sign extension. If the literal is anything else, this may not correspond to sign extension. Literal processing in the VISION architecture does not depend on the data type. When used in instructions that involve data types smaller than 32 bits, the literal value will be left-truncated to the desired number of bits without overflow indication.

6.1.1.3 Memory Operands

Memory operands are operands that have a logical address. This logical address is found in one of the base registers B0..B7 (B6 is better known as Q; B7 as S) and modified through adding an index and/or a displacement. The rules for address arithmetic on memory operands are detailed below. In any case, the result will be a logical address, called the effective logical address. The use of this effective logical address is under control of the instruction.

Typically, the effective logical address is used in a memory access. For example,

```
ADD4  Q, B4
```

will use the address in base register Q to read a 4-byte value from memory in order to add this to the 4-byte value read from memory at the logical address in B4. The result of the addition will be written back as a 4-byte value to memory at the logical address in B4.

The effective logical address is occasionally needed for other purposes. During string moves, the logical addresses of source and target areas are incremented such as to sweep through the areas in memory. In the MOVEADR instruction, the effective logical address of the first operand is itself the value stored in the second operand; thus making the address arithmetic logic available to software, e.g. in building reference parameters. These other uses are the ones where register operands are not suitable; registers cannot be addressed. The set of registers X0..X15 exist outside logical address space.

Base register operands might be regarded as a final way to use what looks like a memory operand. See section 6.1.1.3.2.

6.1.1.3.1 Computing the effective logical address

A memory operand designates a base register, a displacement and an optional index register. The base register is one of B0..B5, Q, S; the index register is one of X0..X15 and the displacement is much like a literal in that it is contained in the instruction itself. The displacement is up to 32 bits in length.

The designated base register contains a logical object id and a logical offset. The effective logical address will have the same logical object id as the designated base register: the address computation does not carry into the object portion of the base register. Effective logical address computation consists of the two's complement addition of the 32-bit logical offset of the base register, the 32-bit displacement and the 32-bit index (if present) and ignoring overflow and/or carry.

Note that this allows implementations to perform the additions in any order.

6.1.1.3.2 Base register operands

Several instructions expect a base register as an operand. This is indicated through the "b" attribute as described in section 6.1.5. A base register operand is encoded like a memory operand; but no memory access is implied and the result of the effective address computation, if performed at all, is irrelevant. For a base register operand, the only relevant field is the base register field in the encoding for the memory operand.

6.1.2 Instruction Encoding

Instructions consist of an opcode and a list of descriptors for each operand.
The opcode identifies uniquely which operation to perform, the data type involved and the number of operands.

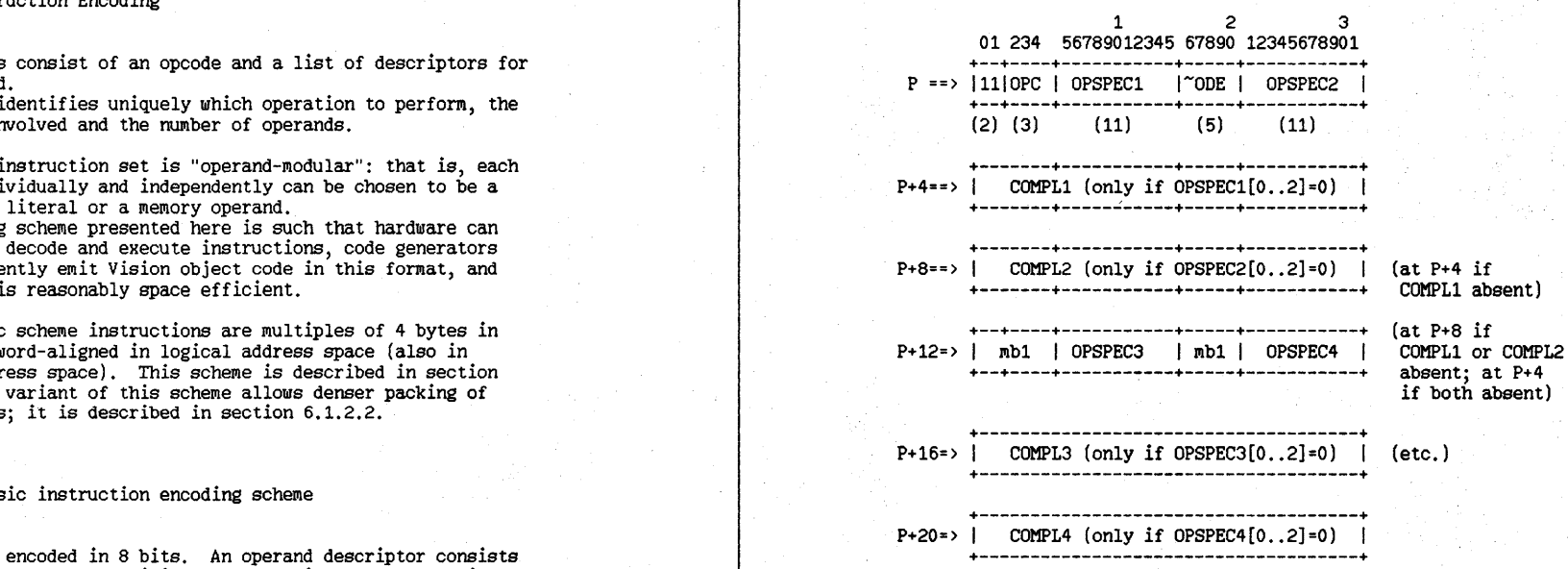
The Vision instruction set is "operand-modular": that is, each operand individually and independently can be chosen to be a register, a literal or a memory operand.
The encoding scheme presented here is such that hardware can efficiently decode and execute instructions, code generators can conveniently emit Vision object code in this format, and the scheme is reasonably space efficient.

In the basic scheme instructions are multiples of 4 bytes in length and word-aligned in logical address space (also in virtual address space). This scheme is described in section 6.1.2.1. A variant of this scheme allows denser packing of instructions; it is described in section 6.1.2.2.

6.1.2.1 Basic instruction encoding scheme

Opcodes are encoded in 8 bits. An operand descriptor consists of an 11-bit operand specifier OPSPEC optionally accompanied by a 32-bit operand completion COMPL. The completion is used when the 11 bits of the operand specifier do not suffice to uniquely determine the operand; these cases correspond to the first 3 bits of the operand specifier OPSPEC[0..2] being zero. Operands are encoded in pairs; if the instruction has an odd number of operands, an additional dummy operand (e.g. literal zero) is specified.

The sketch below shows an instruction with 4 operands. Note that the first two bits of the instruction words are both one in order to identify that the basic format is used. Note also that the 8-bit OPCODE is found by concatenating bits 2..4 and bits 16..20 of the first instruction word. The address of the instruction (e.g. when used as a branch target) is the address of the first byte of the first word of the instruction. Note that, in this format, the instruction address is a multiple of four.



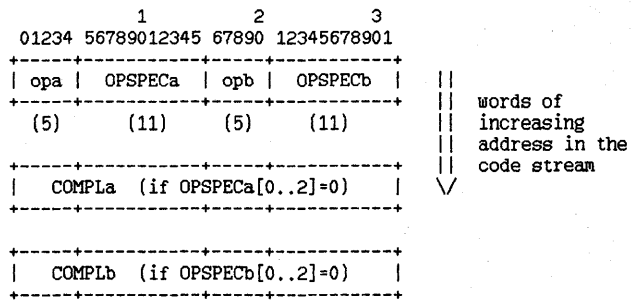
Note: mb1 (must be ones) denotes a field that should consist of all ones. It is the responsibility of software to ensure this; hardware implementations may assume ones in these fields without having to check this.

6.1.2.2 Dense Instruction Encoding Scheme

This is a variant of the basic encoding scheme that allows some instructions to be packed two per word.
A subset of 24 instructions are candidates for this more densely packed scheme. These instructions are such that OPCODE[0] and OPCODE[1] are not both one. Instructions in this subset are all single operand instructions.
If two consecutive instructions are both in this subset, the pair qualifies. For such a pair, the sequence

```
OPCODEa  OPERANDa ;  OPCODEb  OPERANDb;
```

can be encoded as:



Note 1: Opa=OPCODEa[3..7] and opb=OPCODEb[3..7].
(i.e. OPCODEa = !E0 + opa; OPCODEb = !E0 + opb)

Note 2: Bits 0 and 1 of the first instruction word in this packed format are never both one, so the two formats can be distinguished.

Note 3: The second instruction in such a pair can be a branch target. The P-value corresponding to the second instruction in such a pair is taken to be the address of the byte containing "opb"; this is on an even byte this is on an even byte boundary. All branch targets will be even byte addresses.

6.1.2.3 Secondary Instruction Set Encoding

A few of the 8-bit opcodes act as an escape to a secondary instruction set. The opcode "SYS" is one of these.
Most operating system support instructions, including the I/O instruction sets, are in this secondary instruction set.
This section describes their encoding. The instruction "PDDEL" (delete from page directory) will serve as an example.
PDDEL is in the secondary instruction set for "SYS". It has a single operand "ppn". In the opcode assignment chart PDDEL is listed as having a secondary opcode of !09 (hexadecimal) or 9.

The instruction:

```
PDDEL  ppn
```

is encoded as if it were:

```
SYS  9, ppn
```

with the "9" encoded as a short literal (see section 6.1.3.1).
In other words, the secondary opcode is treated as an additional literal operand of the primary escape opcode. Implementations may differ in their results if the secondary opcode is encoded as an operand other than a short literal.

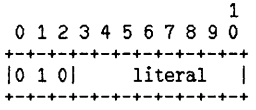
6.1.2.4 Code Bounds Violations

The object identified by P is called the current code object. PB denotes the first byte of the current code object; PL denotes the first byte beyond the current code object; both are multiples of four. Hardware checks P against PB and PL on all transfers of control. Hardware may also check P against PL when executing instructions not involving transfer of control. The effect of executing an instruction that starts within the current code object but has completion words that fall outside of it may differ across implementations.

6.1.3 Operand descriptors

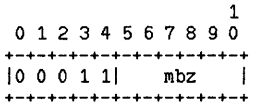
An operand descriptor consists of an 11-bit operand specifier OPSPEC accompanied by a 32-bit operand completion COMPL when OPSPEC[0..2]=000. The formats of the operand specifiers are detailed in the sections below. ("mbz" means "must be zero"; hardware may assume an mbz field to be zero without having to check this.)

6.1.3.1 Short literal



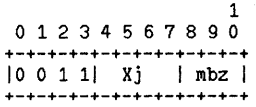
The value of the operand is obtained from the OPSPEC itself, through sign-extension of the 8-bit literal field.

6.1.3.2 Long literal



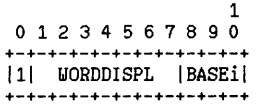
The value of the operand is obtained from the 32-bit COMPL. For data types > 4 bytes this value is then sign-extended.

6.1.3.3 Register operand



The operand is the designated index register. For data types > 4 bytes, a pair or quadruple of consecutive registers is designated.

6.1.3.4 Memory operand (base+short word displacement)



The operand is in memory. The logical address is given by a base register to which is added WORDDISPL*4. Note that WORDDISPL is zero-extended.

6.1.3.5 Memory operand (base-short word displacement)

The operand is in memory. Its logical address is given by a base register to which is added the one-extended WORDDISPL*4.

6.1.3.6 Memory operand (base+long displacement)

The operand is in memory. Its logical address is given by a base register to which is added the two's complement byte displacement found in the 32-bit COMPL.

6.1.3.7 Memory operand (base+index)

The operand is in memory. Its logical address is given by a base register to which is added the two's complement 32-bit value found in the designated index register.

6.1.3.8 Memory operand (base+index+displacement)

The operand is in memory. Its logical address is given by a base register to which is added the two's complement 32-bit value found in the designated index register and also the two's complement 32-bit displacement value found in the 32-bit COMPL in the instruction itself.

6.1.4 Opcode Assignments

The following chart shows the association of opcodes with the instruction name (mnemonic). The 8-bit encoding of the opcode is found by adding the hexadecimal number in the row of the instruction to the hexadecimal number in its column.

OPCODE	+!00	+!01	+!02	+!03	+!04	+!05	+!06	+!07
!00	ERROR	NOP	EXIT	SEXIT	TESTA	TESTB	TESTOV	BREAK
!08	*	*	*	*	PSEB	PSDB	DISP	TRY
!10	DISABLE	ENABLE	INTERRUPT	UNTRY	EXTEND	DELETE	CHECKA	CHECKB
!18	TESTSTRIP	*	*	*	*	BRX	*	*
!20	*	*	QUAD4	*	POP8	*	*	POP16
!28	PUSH1	PUSH2	PUSH8	*	TESTDOWN	UP	DOWN	PUSH16
!30	POP1	POP2	*	*	*	TESTREF	*	TEST16D
!38	*	TEST2	TEST8	TEST4F	TEST4D	TEST8D	TEST8F	TEST16F
!40	AND4	*	*	MPY4F	MPY8	*	MPY8F	MPY16F
!48	NOT4	*	DIV4	DIV4F	DIV8	*	DIV8F	DIV16F
!50	OR4	REM4	NEG4	NEG4F	NEG8	REM8	NEG8F	NEG16F
!58	XOR4	MOD4	ABS4	ABS4F	ABS8	MOD8	ABS8F	ABS16F
!60	CMP1	CMP2	CMP4	CMP4F	CMP8	BCMP8	CMP8F	CMP16F
!68	MOVE1	MOVE2	MOVE4	*	MOVE8	BSET8	*	MOVE16
!70	TESTBIT	ISC42	ADD4	ADD4F	ADD8	BGET4	ADD8F	ADD16F
!78	*	MPY4	SUB4	SUB4F	SUB8	BSET4	SUB8F	SUB16F
!80	MOVEADR	BMOVEADR	*	*	*	*	*	*
!88	*	*	MOVEfSP4	MOVEfSP8	TESTSEMA	*	*	*
!90	*	*	MOVEtSP4	MOVEtSP8	MOVESEMA	*	*	*
!98	CHECKLO	CHECKHI	DUP	OVPUNCH	CVID	CMP4D	CMP8D	CMP16D
!A0	LSL4	ASL4	BCMP4	GETSIGN	CVDI	ADD4D	ADD8D	ADD16D
!A8	LSR4	ASR4	BADD4	VALN	CVAD	SUB4D	SUB8D	SUB16D
!B0	LSL8	ASL8	BSUB4	VALD	CVDA	MPY4D	MPY8D	MPY16D
!B8	LSR8	ASR8	*	*	*	DIV4D	DIV8D	DIV16D
!C0	PROBE	*	MOVEBIT	MOVEC	SRD	MOVED	MOVEBLR	CMPB
!C8	DPF	*	REP	CMPC	SLD	TRANSL	MOVEBRL	CMPT
!D0	POLY4F	POLY8F	POLY16F	SCANUNTIL	*	*	*	*
!D8	*	*	*	*	*	VECTOR	SYS	CONVERT
@!E0	BRG	BRGE	BRGL	BRNU	PUSH4	PUSHADR	POP4	BPOP8
@!E8	BRGU	BRNL	BRNE	BR	TESTLSB	TEST1	TEST4	BTEST8
@!F0	BRN	BRE	BRL	BRLE	CALL	CALLX	*	BREAK
!F8	BRU	BREU	BRLU	BRNG	*	*	*	ERROR

Note 1: the rows marked with "@" contain the instructions that can be packed two per word.

Note 2: the instructions VECTOR and SYS are escapes to a secondary set of opcodes.

The following chart shows opcode assignments for instructions in the secondary instruction set in the escape group for "SYS".

SYS OPCODE	+!00	+!01	+!02	+!03	+!04	+!05	+!06	+!07
!00	IEXIT	SWITCH	*	RSWITCH	IDLE	STOP	*	*
!08	PDINS	PDDEL	SYNCOD	GROWGD0	*	*	*	*
!10	SYNCTCB	SYNCIB	CVLAtVA	HASH	CVVAtPP	LAUNCH	*	*
!18	*	*	*	*	*	*	*	*
#!20	IOW	IOR	IOC	*	*	*	*	*
!28	*	*	*	*	*	*	*	*
!30	*	*	*	*	*	*	*	*
$!38	IFC	WCMD	WBYTE	RBYTE	*	*	*	*
$!40	CHNOP	RCL	PRD	PDA	PAR	RDP	WDP	RIS
$!48	CIS	SIS	*	*	*	*	*	*
!50	*	*	*	*	*	*	*	*
!58	MOVEtCSP	*	*	*	*	*	*	*
---	*	*	*	*	*	*	*	*
!F8	*	*	*	*	*	*	*	*

Note 1: the I/O instructions in the rows marked "#" are defined for MPB-based implementations. On any other implementation these instructions will cause a trap.

Note 2: the I/O instructions in the rows marked "$" are defined for PICMB-based implementations. On any other implementation these instructions will cause a trap.

The chart given below shows the association between vector instructions and their opcodes. All these are secondary opcodes in the "VECTOR" escape group.

VECTOR
OPCODE

	+!00	+!01	+!02	+!03	+!04	+!05	+!06	+!07
!00	VMOVE2	*	VMOVE4	*	VMOVE8	*	VMOVE16	*
!08	VABS	*	VABS4	VABS4F	VABS8	VABS8F	*	VABS16F
!10	*	*	VNEG4	VNEG4F	VNEG8	VNEG8F	*	VNEG16F
!18	*	*	VLSL4	*	VLSL8	*	*	*
!20	*	*	VLSR4	*	VLSR8	*	*	*
!28	*	*	VASL4	*	VASL8	*	*	*
!30	*	*	VASR4	*	VASR8	*	*	*
!38	*	*	*	*	*	*	*	*
!40	*	*	*	*	*	*	*	*
!48	*	*	*	*	*	*	*	*
!50	*	*	VCMPRS4	*	VCMPRS8	*	VCMPRS16	*
!58	*	*	VEXPND4	*	VEXPND8	*	VEXPND16	*
!60	*	*	VACC4	VACC4F	VACC8	VACC8F	*	VACC16F
!68	*	*	*	VACCD4F	*	VACCD8F	*	*
!70	*	*	VMAXL4	VMAXL4F	VMAXL8	VMAXL8F	*	VMAXL16F
!78	*	*	VMINL4	VMINL4F	VMINL8	VMINL8F	*	VMINL16F
!80	*	*	VADD4	VADD4F	VADD8	VADD8F	*	VADD16F
!88	*	*	VSUB4	VSUB4F	VSUB8	VSUB8F	*	VSUB16F
!90	*	*	VMPY4	VMPY4F	VMPY8	VMPY8F	*	VMPY16F
!98	*	*	VDIV4	VDIV4F	VDIV4F	VDIV8F	*	VDIV16F
!A0	*	*	VAND4	VOR4	VAND8	VOR8	*	*
!A8	*	*	VXOR4	*	VXOR8	*	*	*
!B0	*	*	VGATH4	VSCAT4	VGATH8	VSCAT8	VGATH16	VSCAT16
!B8	*	*	VEXT4	VINS4	VEXT8	VINS8	VEXT16	VINS16
!C0	*	*	VREM4	*	VREM8	*	*	*
!C8	*	*	VMOD4	*	VMOD8	*	*	*
!D0	*	*	VCMP4	VCMP4F	VCMP8	VCMP8F	*	VCMP16F
!D8	*	*	*	*	*	*	*	*
!E0	CLRMR	STMR	LDMR	MRNOT	MRAND	MROR	MRXOR	*
!E8	LDVLR	STVLR	RVLRT	*	*	*	*	*
!F0	UVCSA	PUVCSA	IVB	LVB	VINVAL	*	*	*
!F8	*	*	*	*	*	*	VCONVERT	*

6.1.5 Attributes

Attributes can be associated with operands or with instructions.

6.1.5.1 Operand Attributes

Each instruction has an implied number of operands and for each operand of the instruction there is an implied attribute. For example, section 6.2.2 shows the ADD4 instruction in the following way:

```
ADD4  term.r4, sum.rw4
```

Here "term" and "sum" are merely symbolic names for the two operands; ".r4" and ".rw4" are the operand attributes. These attributes are composed of individual elements like "r", "w", and "4".

The "r" attribute indicates that the operand must allow reading from; hence it can be a literal, a register or a memory operand with appropriate read access rights.

The "w" attribute indicates that the operand must allow writing to; hence it must not be a literal, but must be a register or a memory operand with appropriate write access rights.

The "4" attribute indicates that the operand is a 4-byte entity; this has obvious implications for memory operands with respect to memory access and bounds checking.

operand attribute	stands for:	LITERAL	REGISTER	MEMORY
r	read	ok	ok	check OD.TYP&AR
w	write	illegal	ok	check OD.TYP&AR
m	memory	illegal	illegal	ok
b	base	illegal	illegal	ok
c	code	illegal	illegal	if OD.TYP=code
v	vector	(1)	(1)	(1)
1	1-byte	(2)	(2)	check w/ OD.UB
2	2-byte	(2)	(2)	check w/ OD.UB-1
4	4-byte	(2)	(2)	check w/ OD.UB-3
8	8-byte	(2)	(2)	check w/ OD.UB-7
16	16-byte	(2)	(2)	check w/ OD.UB-15

Notes: (1): see section 6.4.
(2): see sections 6.1.6 and 6.1.7.

6.1.5.2 Instruction Attributes

Instruction attributes include the data type of the operation to be performed. For example, in:

```
ADD4F  X3, X5
```

the suffix "4F" indicates that addition is to be performed on a 4-byte Floating point number according to the rules of floating point arithmetic on 32-bit numbers.

A list of data types follows:

instruction attribute	interpretation
1	8-bit unsigned integer, or any 1-byte entity
2	16-bit two's complement integer/any 2-byte entity
4	32-bit two's complement integer/any 4-byte entity
8	64-bit two's complement integer/any 8-byte entity
16	128-bit entity of any type
4F	32-bit IEEE floating point
8F	64-bit IEEE floating point
16F	128-bit IEEE floating point
4D	32-bit packed decimal
8D	64-bit packed decimal
16D	128-bit packed decimal

6.1.6 Sources

A source is a value derived from an operand that is of the right size to be used in the instruction. This section makes explicit the actions to be performed on read operands to turn them into a source. For example, in the instruction:

```
ADD8  1, X3
```

the literal "1" is sign-extended to form a 64-bit source; the register X3 is paired with the register X4 to form a 64-bit register pair acting as a 64-bit source; both 64-bit numbers are then added together in two's complement arithmetic. (The result is then stored according to the rules in section 6.1.7.)

The following chart makes this all expliclt.

	source (in bytes)				
operand descr.	1	2	4	8	16
short literal	as is	SE16	SE32	SE64	SE128
long literal	TR8	TR16	as is	SE64	SE128
register opnd	TR8	TR16	as is	pair	quad
memory operand	R1	R2	R4	R8	R16

where:

SEn = sign-extend to n bits. This always means replicating the lowest numbered bit regardless of data type.

TRn = truncate. This always means discarding all but the n rightmost bits.

pair = pair with following register. X0 follows X15.

quad = pair with following 3 registers.

Rn = Read n consecutive bytes from memory starting at the effective logical address. Check the object type in the OD for the logical object; check the read access rights at the current privilege level; check bound LB and UB-n+1 (both inclusive).

6.1.7 Destinations

The result of an operation may have to be massaged before it can be stored away. This section makes these operations explicit.

operand descr.	destination (in bytes) 1	2	4	8	16
short literal	illegal	illegal	illegal	illegal	illegal
long literal	illegal	illegal	illegal	illegal	illegal
register opnd	ZE32	SE32	as is	pair	quad
memory operand	W1	W2	W4	W8	W16

where:

illegal = a literal operand must not occur in this context.

ZE32 = right-justify and zero-extend to 32 bits

SE32 = sign-extend, i.e. replicating the lowest numbered bit, to 32 bits.

pair = pair with following register. X0 follows X15.

quad = pair with following 3 registers.

Wn = write n consecutive bytes to memory starting at the effective logical address. Use the OD of the object to check type and write access rights. Use the bounds in the OD for bounds checking: LB and UB-n+1. (both inclusive)

6.1.8 Traps

Traps are described in detail in chapter 7. Chapter 6 shows the conditions under which traps occur as the necessary result of instruction execution, but it does not show the parameters passed to the trap handler, nor does it show the pervasive traps and conditions such as power-fail, page fault, etc. In particular, it does not show any of the traps in the list below under the heading of "Opnd" that can occur on operand accessing.

Opnd: can be any of
- OPSPECV
- DATATYPV
- DATAODTV
- DATAARV
- DATABNDSV

AddressingV: can be any of the above, but in accesses other than those involving explicit operands

Arith: can be any of
- INTOVF
- INTDVDZ

FlArith: can be any of
- FLINV
- FLDVDZ
- FLOVF
- FLUNF
- FLINX

DecArith: can be any of
- DECINVL
- DECOVF
- DECDVDZ
- DECINVDG

6.2 Base Instruction Set

6.2.1 Data Movement Instructions

6.2.1.1 MOVEt source.r, destination.w

Move data element. The value of "source" is copied to "destination". The number of bytes moved is implied by the type "t". Partial overlap of the areas containing source and destination may give results that differ across implementations.

```
destination := source;
```

includes: MOVE1 MOVE2 MOVE4 MOVE8 MOVE16

6.2.1.2 MOVEADR operand.m, destination.w8

Move logical address. The 64-bit logical address of "operand" is computed and the result stored in "destination". "Operand" must be a memory operand. The byte that is addressed by "operand" requires neither legal read nor write access, nor need it be within the logical object bounds. This instruction makes the operand-addressing hardware available to software, e.g. for building reference arguments. This instruction also doubles as a way to obtain the value in a given base register; for this usage the assembly language alias "BGET8" is provided.

```
destination := logical_address_of(operand);
```


6.2.1.3 PUSHt source.r

Push data element. The value of "source" is copied into a temporary register of length 4 bytes (for PUSH1, PUSH2, PUSH4); of length 8 bytes (for PUSH8) or 16 bytes (for PUSH16). For PUSH1, "source" is zero-extended to 32 bits; for PUSH2, "source" is sign-extended to 32 bits. The temporary is then pushed onto the stack. After PUSHt, the top-of-stack register S will point to the first byte beyond the data that was moved. On stack overflow, S will be restored to the value it had before the instruction.

```
Temp[0..k] := source;
                {zero-extend for t=1,
                 sign-extend for t=2}
S := S + m;
(S-m)[0..k] := Temp[0..k];
      {Here for  t = 1, 2, 4, 8, 16
           use  k =31,31,31,63,127
           and  m = 4, 4, 4, 8, 16}
```

	PUSH1	PUSH2	PUSH4	PUSH8	PUSH16
Traps:	STKOVF	STKOVF	STKOVF	STKOVF	STKOVF

6.2.1.4 PUSHADR operand.m

Push logical address. The 64-bit logical address of "operand" is computed and pushed onto the stack. "Operand" must be a memory operand.
PUSHADR also doubles as a way to push the value of a base register onto the stack; for this usage the assembly language alias "BPUSH8" is provided.

```
MOVEADR  operand, Temp;
PUSH8    Temp;
```

Traps: STKOVF

6.2.1.5 POPt destination.w

Pop data element. A number of bytes given by "t" are popped off the stack and stored in "destination". In case less than 4 bytes are popped, the top-of-stack register S is further decremented so as to remain word-aligned. On stack underflow, S is restored to the value it had before the instruction.

```
destination := (S-t)[0..p];
S := S - m;

    {Here for t = 1, 2, 4, 8, 16
         use p = 7,15,31,63,127
         and m = 4, 4, 4, 8, 16}
```

	POP1	POP2	POP4	POP8	POP16
Traps:	STKUNF	STKUNF	STKUNF	STKUNF	STKUNF

6.2.1.6 DPF value.r4, shiftcount.r1, mask.r4, target.rw4

Deposit Field. "Value" is deposited in a field of "target" identified by "shiftcount" and "mask". "Mask" is assumed to be of the form
2^31 - (2^fieldsize - 1) * 2^shiftcount
but if it is not, the following definition still applies. However, implementations may substitute a circular shift for the logical shift indicated.

```
MOVE4  value, Val;
LSL4   shiftcount, Val;    {see 6.2.3}
MOVE4  target, Tgt;
AND4   mask, Tgt;          {see 6.2.3}
OR4    Val, Tgt;           {see 6.2.3}
MOVE4  Tgt, target ;
```


6.2.1.7 MOVEC length.r4, source.mr, destination.mw

Counted move of bytes. This instruction moves a string of contiguous bytes starting at the logical address given by the specifier for "source" and of length "length" to the area of equal length starting at the logical address given by the specifier for "destination". If "length" is negative or zero, no bytes are moved. If conditions (a) or (b) are violated, implementations may yield different results; however, in no case should reads or writes to memory be performed in violation of access rights.

a) the source and destination area must not overlap
b) "length" must not be in the destination area

The MOVEC instruction is interruptible, at intervals determined by each implementation.

```
if IIP = 0 then C := 0
else  POP4  C;
IIP := 0;
MOVEADR  source, Fromla;
MOVEADR  destination, Tola;
Lgth := length[0..31] - 1;
while  C < Lgth  do
begin
   Byte := (Fromla + C)[0..7];
   (Tola + C)[0..7] := Byte;
   C := C + 1;
   {if implementation chooses to acknowledge
    external interrupts here, then
    PUSH4  C  and set IIP := 1}
end;
```

Traps: AddressingV

6.2.1.8 MOVEBIT bitindex.r4, source.r1, bitarray.mrw

Move a bit. The least significant bit of "source" is stored in the array of bits (whose first byte is addressed by "bitarray") at the index "bitindex". All other bits of "source" are ignored. "Bitindex" is an arbitrary two's complement integer. Only the address of the byte in which the bit is actually stored need be within the bounds of the logical object. No other bits in the byte are disturbed. Memory interlock is not guaranteed (see TESTSEMA).

```
MOVEADR  bitarray, Addr;
Source_byte[0..7] := source;
Bit := Source_byte[7];
Byte_offset := bitindex[0..28] {sign-extended};
Addr[32..63] := Addr[32..63] + Byte_offset;
Bit_number := bitindex[29..31];
Byte[0..7] := (Addr)[0..7];
Byte[Bit_number] := Bit;
(Addr)[0..7] := Byte;
```

Traps: AddressingV

6.2.1.9 MOVEBLR fillchar, srcl, src, destl, dest

MOVEBLR fillchar.r1, srcl.r4, src.mr, destl.r4, dest.mw

Move bytes left-to-right. This instruction moves a string of contiguous bytes starting at the logical address given by "src" and of length "srcl" to the logical address given by "dest" and of length "destl". If "destl" is <=0, nothing is moved. If "srcl" > "destl", the string is truncated on the right. If "srcl" < "destl", the string is padded on the right with "fillchar". Overlap between "src" and "dest" areas is explicitly allowed: the algorithm below defines the intended effect.

```
if IIP = 0 then C := 0
else  POP4  C;
IIP := 0;
MOVEADR  src,  Lfrom;
MOVEADR  dest, Lto;
Sl := srcl;  Dl := destl;  F := fillchar;
while  C < Dl  do begin
   if C < Sl
   then (Lto+C)[0..7] := (Lfrom+C)[..7]
   else (Lto+C)[0..7] := F;
   C := C + 1;
   { if implementation chooses to acknowledge
     an external interrupt here,
     then PUSH4  C and set IIP := 1 }
   end;
```

Traps: AddressingV

6.2.1.10 MOVEBRL fillchar, srcl, src, destl, dest

MOVEBRL fillchar.r1, srcl.r4, src.mr, destl.r4, dest.mw

Move bytes right-to-left. This instruction moves a string of contiguous bytes starting at the logical address given by "src" and of length "srcl" to the logical address given by "dest" and of length "destl". If "destl" <=0, nothing is moved. If "srcl" < "destl", the string is padded on the left with "fillchar". If "srcl" > "dstl", the string is truncated on the left. Overlap between "src" and "dest" areas is explicitly allowed: the algorithm below defines the intended effect.

```
if IIP = 0  then C := 0
else  POP4  C;
IIP := 0;
MOVEADR  src,  Lfrom;
MOVEADR  dest, Lto;
S1 := srcl;  D1 := destl;  F := fillchar;
while  C < D1  do begin
   if  C < S1
   then   (Lto+C)[0..7] := (Lfrom+C)[0..7]
   else (Lfrom+C)[0..7] := F;
   C := C + 1;
   { if implementation chooses to acknowledge
     an external interrupt here,
     then PUSH4  C, and set IIP := 1 }
   end;
```

Traps: AddressingV

6.2.1.11 TRANSL table.mr, length.r4, source.mr, dest.mw

Translate. Contiguous bytes from "source" are moved to "dest" one at a time by using the byte from "source" to index into "table" and the byte found in "table" is stored at "dest".
A count of "length" bytes is moved; if "length" is not positive, no bytes are moved.

```
if IIP = 0  then  C := 0
else  POP4  C;
IIP := 0;
MOVEADR  source, Lfrom;
MOVEADR  dest,   Lto;
MOVEADR  table,  Ltable;
while  C < length  do begin
   Byte := (Lfrom+C)[0..7];
   Byte := (Ltable+Byte)[0..7];
   (Lto+C) := Byte;
   C := C + 1;
   {if implementation chooses to acknowledge an
    external interrupt here,
    then PUSH4  C, and set IIP := 1 }
   end;
```

Traps: AddressingV

6.2.1.12 DUP wordcount.r4, value.r4

Duplicate. The 32-bit value "value" is pushed onto the stack a "wordcount" number of times. This instruction must be interruptible.

```
if IIP = 0 then C := 1
else POP4 C;
IIP := 0;
while C <= wordcount[0..31] do begin
   PUSH4 value;
   C := C + 1;
   {if implementation chooses to acknowledge
    interrupts here,
    then PUSH4 C and set IIP := 1}
   end;
```

Traps: STKOVF

6.2.1.13 REP wordcount.r4, value.r4, operand.mw

Replicate. The 32-bit value "value" is stored in "wordcount" consecutive words of memory starting at the address of "operand". If the buffer to be initialized with "value" overlaps with either "wordcount" or "value", implementations may produce different results. This instruction must be interruptible.

```
if IIP = 0 then C := 1
else POP4 C;
IIP := 0;
MOVEADR operand, Toaddr;
while C <= wordcount[0..31] do begin
   (Toaddr + 4*C )[0..31] := value;
   C := C + 1;
   {if implementation chooses to acknowledge
    interrupts here,
    then PUSH4 C and set IIP := 1}
   end;
```

Traps: AddressingV

6.2.1.14 EXTEND wordcount.r4

Extend top-of-stack. Base register "S" is incremented by four times the value of "wordcount", which must be positive. On stack overflow S will be restored to its original value and a trap taken. Ring level 1 is required.

```
if XL>1 then Trap"INSPRIV";
if wordcount < 0 then Trap"STKDEXTV";
if S + 4 * wordcount <= SL then Trap"STKOVF";
S : = S + 4 * wordcount;
```

Traps: INSPRIV
STKDEXTV
STKOVF

6.2.1.15 DELETE wordcount.r4

Delete from top-of-stack. Base register "S" is decremented by four times the value of "wordcount", which must be positive. If the new S would end up below Q, the original S will be restored and a trap taken.

```
if wordcount < 0 then Trap"STKDEXTV;
if S - 4 * wordcount < Q then Trap"STKUNF";
S := S - 4 * wordcount;
```

Traps: STKDEXTV
STKUNF

6.2.2 Arithmetic Instructions

This section includes instructions for arithmetic operations on the integer and floating point scalar data types. Decimal arithmetic is covered in section 6.3, vector arithmetic is covered in section 6.4.

6.2.2.1 ADDt term.r, sum.rw

Add. "Term" is added to "sum" and the result is stored in "sum". In case of integer overflow, "sum" is set to the least significant bits of the correct mathematical result.

	ADD4	ADD8	ADD4F	ADD8F	ADD16F
Status:	Ovfl	Ovfl	Flflags	Flflags	Flflags
Traps:	Arith	Arith	FlArith	FlArith	FlArith

6.2.2.2 SUBt term.r, difference.rw

Subtract. "Term" is subtracted from "difference" and the result is stored in "difference". In case of integer overflow "difference" is set to the least significant bits of the correct mathematical result.

	SUB4	SUB8	SUB4F	SUB8F	SUB16F
Status:	Ovfl	Ovfl	Flflags	Flflags	Flflags
Traps:	Arith	Arith	FlArith	FlArith	FlArith

6.2.2.3 MPYt factor.r, product.rw

Multiply. "Factor" is multiplied by "product" and the result is stored in "product". In case of integer overflow, "product" is set to the least significant bits of the correct mathematical result.

	MPY4	MPY8	MPY4F	MPY8F	MPY16F
Status:	Ovfl	Ovfl	Flflags	Flflags	Flflags
Traps:	Arith	Arith	FlArith	FlArith	FlArith

6.2.2.4 DIVt divisor.r, dividend.rw

Divide. "Dividend" is divided by "divisor" and the result stored in "dividend". On integer divide with "divisor" zero, the new value of "dividend" is indeterminate; however, the sign of "dividend" should not be changed. For integer divide, the algebraic result is truncated towards zero. On integer overflow, "dividend" is left as zero.

	DIV4	DIV8	DIV4F	DIV8F	DIV16F
Status:	Ovflow	Ovflow	FlFlags	FlFlags	FlFlags
Traps:	Arith INTDVDZ	Arith INTDVDZ	FlArith	FlArith	FlArith

6.2.2.5 NEGt source.r, destination.w

Negate. "source" is negated (subtracted from zero) and the result is stored in "destination". On integer overflow, "destination" is left as the largest negative value.

	NEG4	NEG8	NEG4F	NEG8F	NEG16F
Status:	Ovflow	Ovflow	FlFlags	FlFlags	FlFlags
Traps:	Arith	Arith	FlArith	FlArith	FlArith

6.2.2.6 ABSt source.r, destination.w

Absolute value. The absolute value of "source" is computed and the result is stored in "destination". On integer overflow, "destination" is left as the largest negative value.

```
if source < 0  then  destination := 0 - source
else  destination := source;
```

	ABS4	ABS8	ABS4F	ABS8F	ABS16F
Status:	Ovflow	Ovflow	FlFlags	FlFlags	FlFlags
Traps:	Arith	Arith	FlArith	FlArith	FlArith

6.2.2.7 REMt divisor.r, dividend.rw

Remainder. The algebraic remainder of the division of "dividend" by "divisor" is computed and the result stored in "dividend". The remainder has the same sign as the old value of "dividend", and is less in absolute value than "divisor". The equation

divisor * quotient + remainder = dividend

always holds. If "divisor" is zero, the magnitude of "dividend" will be indeterminate.

	REM4	REM8
Status:	Ovflow	Ovflow
Traps:	Arith	Arith
	INTDVDZ	INTDVDZ

6.2.2.8 MODt divisor.r, dividend.rw

Modulus. The modulus of "dividend" and "divisor" is computed and the result is stored back into "dividend". The modulus is defined to be the quantity that is positive (or zero) and less than the absolute value of "divisor", and such that the difference of modulus and "dividend" is a whole multiple of "divisor". This definition determines the modulus uniquely, except when "divisor" has the value zero, in which case the magnitude of "dividend" will be indeterminate. Note that the equation

divisor * quotient + modulus = dividend

will not always hold.

	MOD4	MOD8
Status:	Ovflow	Ovflow
Traps:	Arith	Arith
	INTDVDZ	INTDVDZ

6.2.2.9 POLYt degree.r1, polyn.mr, operand.rw

Polynomial evaluation. This instruction computes the value of a polynomial evaluated for the value of "operand", storing the result back into "operand". The polynomial is defined by degree "degree" (interpreted as an unsigned integer) and a table of coefficients, "polyn". The coefficient of the highest order term of the polynomial is addressed by "polyn". All coefficients are stored consecutively in memory. The algorithm below is intended to define the value desired, not the precise sequence in which the calculations are actually performed.

```
X := operand;  Y := 0;  C := 0;
MOVEADR  polyn, Lcoeff;
while  C < degree do
   Y := Y * X + (Lcoeff+C*t)[0..8*t-1];
operand := Y;
```

	POLY4F	POLY8F	POLY16F
Status:	FlFlags	FlFlags	FlFlags
Traps:	AddressingV	AddressingV	AddressingV
	FlArith	FlArith	FlArith

6.2.3 Logical Operations and Shifts

6.2.3.1 AND4 mask.r4, operand.rw4

Logical AND. The bit-wise logical AND of "mask" and "operand" is computed and the result is stored in "operand".

6.2.3.2 NOT4 source.r4, destination.w4

Logical NOT. The bit-wise logical NOT (one's complement) of "source" is computed and the result is stored in "destination".

6.2.3.3 OR4 mask.r4, operand.rw4

Logical OR. The bit-wise (inclusive) OR of "mask" and "operand" is computed and the result is stored in "operand".

6.2.3.4 XOR4 mask.r4, operand.rw4

Exclusive OR. The bit-wise exclusive OR of "mask" and "operand" is computed and the result is stored in "operand".

6.2.3.5 LSLt shiftcount.r1, bitfield.rw

Logical shift left. "Bitfield" is shifted left by "shiftcount" bits and the result is stored back in "bitfield". Zeros are shifted into the least significant bit; bits shifted out of the most significant bits are lost. "Shiftcount" is unsigned; only the rightmost 5 bits (for LSL4) or 6 bits (for LSL8) are significant.

includes: LSL4 LSL8

6.2.3.6 LSRt shiftcount.r1, bitfield.rw

Logical shift right. "Bitfield" is shifted right by "shiftcount" bits and the result is stored back in "bitfield". Zeros are shifted into the most significant bit; bits shifted out of the least significant bits are lost. For interpretation of "shiftcount", see LSLt.

includes: LSR4 LSR8

6.2.3.7 ASLt shiftcount.r1, operand.rw

Arithmetic shift left. "Operand" is shifted left by "shiftcount" bits and the result is stored back into "operand". Zeros are shifted into the least significant bit; bits shifted out of the most significant bit are lost. Overflow occurs if the new sign bit or any of the bits shifted out are different from the original sign bit. For interpretation of "shiftcount", see LSLt.

	ASL4	ASL8
Traps:	Ovfl	Ovfl

6.2.3.8 QUAD4 source.r4, destination.w4

Quadruple. "dest" is given the value of "source" times four.

```
MOVE4  source, destination;
ASL4   2, destination;
```

Traps: Ovfl

6.2.3.9 ASRt shiftcount.r1, operand.rw

Arithmetic shift right. Divide the integer value of "operand" by 2**shiftcount, truncating toward zero and store the result back into "operand". See LSLt for interpretation of "shiftcount". Note: for negative values of "operand" this is not the same as a straight sign-propagating right shift.

includes: ASR4 ASR8

6.2.4 Compares and Tests

6.2.4.1 CMPt source1.r, source2.r

Compare. The condition code CC is set depending on the result o the comparison of the values of "source1" and "source2". For CMP2, CMP4 and CMP8 a two's complement compare is performed; for CMP1 an unsigned integer compare is performed. For CMP4F, CMP8F and CMP16F comparison is performed according to the IEEE floating point standard; note that this can result in "unordered".
Condition codes are set as follows:

```
CCG,  if  source1 > source2
CCE,  if  source1 = source2
CCL,  if  source1 < source2
CCU,  if  sources are "unordered" (IEEE only)
```

	CMP1	CMP2	CMP4	CMP8	CMP4F	CMP8F	CMP16F
Status:	CC	CC	CC	CC	CC	CC	CC
					FlFlags	FlFlags	FlFlags
Traps:					FlArith	FlArith	FlArith

6.2.4.2 TESTt source.r

Compare to zero. This instruction is merely a short form of `CMPt source, 0'.

	TEST1	TEST2	TEST4	TEST8	TEST4F	TEST8F	TEST16F
Status:	CC	CC	CC	CC	CC	CC	CC
					FlFlags	FlFlags	FlFlags
Traps:					FlArith	FlArith	FlArith

6.2.4.3 CMPC length.r4, stringa.m, stringb.m, index.w4

Counted Compare. This instruction compares two streams of bytes "stringa" and "stringb" until the first non-equal byte has been encountered or until "length" bytes have been compared. The condition code in STATUSB is set depending on the unsigned compare of the last pair of bytes examined. "Index" is set to the number of the first non-equal byte. If interrupted, the number of bytes left to compare is pushed onto the stack and the instruction-in-progress flag is set.

```
MOVEADR  stringa, Ala;
MOVEADR  stringb, Bla;
if IIP = 0 then C := 0
else  POP4  C;
IIP := 0;
while  C < length and
       (Ala+C)[0..7] = (Bla+C)[0..7]
do begin
   C := C + 1;
   {if implementation chooses to acknowledge
    external interrupts here, then
    PUSH4  C  and set IIP := 1 }
   end;
if  C >= length  then CC := CCE
else if (Ala+C)[0..7] > (Bla+C)[0..7] then CC := CCG
else CC := CCL;
index := C;
```

Status: CC
Traps: AddressingV

6.2.4.4 TESTLSB source.r1

Test least significant bit. The condition code is set to CCG if the least significant bit of "source" is 1, otherwise the condition code is set to CCE.

```
Byte[0..7] := source;
if byte[7] = 1 then CC := CCG
else CC := CCE;
```

Status: CC

6.2.4.5 TESTOV

Test overflow. The condition code is set to CCG if the overflow exception flag is set, else the condition code is set to CCE. The overflow flag is left clear.

```
if STATUSB.OVF = 1 then CC := CCG
else CC := CCE;
STATUSB.OVF := 0;
```

Status: CC
Overflow

6.2.4.6 TESTA

Test conditional break enable. If the "CBA" trap is enabled, set the condition code to CCG, otherwise to CCE.

```
if STATUSB.CBA = 1 then CC := CCG
else CC := CCE;
```

Status: CC

6.2.4.7 TESTB

Test conditional break enable. If the "CBB" trap is enabled, set the condition code to CCG, otherwise to CCE.

```
if STATUSB.CBB = 1 then CC := CCG
else CC := CCE;
```

Status: CC

6.2.4.8 TESTBIT bitindex.r4, bitarray.mr

Test a bit. The condition code is set depending on the value of a bit in a bit array at the index "bitindex". The bit array must be in memory (it cannot be in a register) and its first byte must be addressed by "bitarray". If the bit is found set, the condition code is set to CCG, else it is set to CCE.

```
MOVEADR bitarray, Addr;
Bytela[0..31] := Addr[0..31];
Byte_index[0..31] := bitindex[0..28]; {sign-extend}
Bytela[32..63] := Addr[32..63] + Byte_index;
Byte := (Bytela)[0..7];
Bit_num := bitindex[29..31];
if Byte[Bit_num] = 1 then CC := CCG else CC := CCE;
```

Status: CC
Traps: AddressingV

6.2.4.9 SCANUNTIL charset.mr, string.mr, index.rw4

Scan string until condition satisfied. The string of characters (bytes) pointed to by "string" is scanned for a character that satisfies a particular condition. "Index" must be initialized by software; SCANUNTIL increments "index" (ignoring any overflow) continually as the search proceeds. The condition to be satisfied by the character is encoded as a 256-bit bit array (similar to a Pascal set). Bits found set in the bit array "charset" signify that the corresponding character satisfies the condition. If the logical address of "charset" is at or within 32 bytes of the object's upper bound, an addressing violation trap is raised. This instruction must be interruptible; "index" contains sufficient information to restart.

```
MOVEADR string, St;
index := index - 1;
repeat index := index + 1;
   Char := (St+index)[0..7]; {zero-extend}
   {implementations may choose to
    acknowledge an interrupt here}
   TESTBIT Char, charset; {charset[Char] }
   until CC = CCG;        {     = 1      }
```

Status: CC NOT affected
Traps: AddressingV

6.2.4.10 CMPB fillchar, lgtha, srca, lgthb, srcb, index

CMPB fillchar.r1, lgtha.r4, srca.mr, lgthb.r4, srcb.mr, index.w4

Compare bytes. "Srca" is compared to "srcb" and the condition code set. The shorted string is considered padded with "fillchar". "Index" identifies the offset where bytes started to differ. CCG and CCL refer to the unsigned compare fo the bytes at that location.

```
C := 0;  Flag := 1;
MOVEADR  srca, La;
MOVEADR  srcb, Lb;
while ( C < lgtha  or
        C < lgthb  and
        Flag = 1 ) do begin
   A := fillchar;  B := fillchar;
   if C < lgtha then A := (La+C)[0..7];
   if C < lgthb then B := (Lb+C)[0..7];
   if A <> B then begin
      if A > B then CC := CCG
      else CC := CCL;
      Flag := 0;
      end;
   end;
if Flag = 1 then CC := CCE;
index := C;
```

Status: CC
Traps: AddressingV

6.2.4.11 CMPT table, fillchar, lgtha, srca, lgthb, srcb, index

CMPT table.mr, fillchar.r1, lgtha.r4, srca.mr, lgthb.r4, srcb.mr, index.w4

Compare bytes, translated. This instruction resembles CMPB except in that compares are made of the bytes in "table" indexed by the data bytes in the strings rather than of the actual data bytes themselves.

Status: CC
Traps: AddressingV

6.2.5 Base Register Instructions

6.2.5.1 BGET8 source.b, destination.w8

Get address in base register. See under "MOVEADR".

6.2.5.2 BSET8 source.r8, dest.b

Set base register to logical address. Load the 64-bit logical address from "source" into the designated base register. The logical object id of the logical address must be valid. The logical offset of the logical address need not be within object bounds.

```
j := base_reg_designator_of(dest);
if j >= 6 then Trap"Opnd";
Bj[0..63] := source[0..63];
Group := source[0..2];
Object:= source[3..31];
if Object*16 > length_of_ODT_of(Group)
   then Trap"AddressingV";
```

Traps: AddressingV

6.2.5.3 BMOVEADR source.m, dest.b

Move logical address to base register. This instruction is like MOVEADR, but the result is stored in the base register designated by "dest". This instruction doubles as a base-register-to-base-register move. An assembler language alias "BMOVE8" is provided for this usage.

```
MOVEADR  source,Temp;
BSET8    Temp, dest;
```

Traps: AddressingV

6.2.5.4 BMOVE8 source.b, dest.b

Move base to base register. See under "BMOVEADR".

6.2.5.5 BGET4 source.b, dest.w4

Get offset of base register. Store 32-bit logical offset of the base register designated by "source" into "dest". "Source" must be a memory operand, according to the ".b" attribute.

```
j := base_reg_designator_of(source);
dest := Bj[32..63];
```

6.2.5.6 BSET4 source.r4, dest.b

Set offset of base register. Load "source" into the 32-bit logical offset of the designated base register.

```
j := base_reg_designator_of(dest);
if j >= 6 then Trap"Opnd";
Bj[32..63] := source[0..31];
```

6.2.5.7 BPUSH8 source.b

Push a base register. See under "PUSHADR".

6.2.5.8 BPOP8 dest.b

Pop into a base register. Eight bytes are popped off the stack and loaded into the designated base register.

```
POP8   Temp;
BSET8  Temp, dest;
```

Traps: AddressingV

6.2.5.9 BADD4 term.r4, dest.b

Add to offset of base register. The 32-bit value "term" is added to the logical address in the base register designated by "dest" using wrap-around 32-bit arithmetic. Overflow and carry is ignored.

```
j := base_reg_designator_of(dest);
if j >= 6 then Trap"Opnd";
Bj[32..63] := Bj[32..63] + term;
```

6.2.5.10 BSUB4 term.r4, dest.b

Subtract from offset of base register. The 32-bit value "term" is subtracted from the logical address in the base register designated by "dest" using wrap-around 32-bit arithmetic. Overflow and carry is ignored.

```
NEG4  term, Temp;
BADD4 Temp, dest;
```

6.2.5.11 BCMP4 sourcea.b, sourceb.r4

Compare offset of base register. The 32-bit offset of the base register designated by "sourcea" is compared using two's complement arithmetic with the value of "sourceb". Condition codes are set to reflect the result of the comparison. No overflow can occur.

```
j := base_reg_designator_of(sourcea);
if      Bj[32..63] > sourceb then CC := CCG
else if Bj[32..63] = sourceb then CC := CCE
else CC := CCL;
```

Status: CC

6.2.5.12 BCMP8 sourcea.b, sourceb.r8

Compare base register with logical address. The 64-bit logical address in the base register designated by "sourcea" is compared for equality with the logical address in "sourceb". If the two logicl addresses are equal, CCE is set; otherwise, implementations may set either CCG or CCL arbitrarely.

```
j := base_reg_designator_of(sourcea);
if Bj[0..63] = sourceb then CC := CCE
else  CC := CCG  {or CCL};
```

Status: CC

6.2.5.13 BTEST8 source.b

Test base register for NIL. The base register designated by "source" is compared to a logical address of all zeros.

```
BCMP8  source, 0;
```

Status: CC

6.2.6 Transfer of Control

6.2.6.1 BR{GLEU} target.r4

Branch. Depending on the match between the condition code field in the status register and the mask comprised of four bits in the opcode, execution continues with either the next instruction in sequence or with the instruction explicitly designated by "target". If a match is found then the branch is taken. The target address of the branch is found by adding "target"*2 to the value of P P at the beginning of the branch instruction itself. If the branch is not taken, and the target is in any way illegal, implementations may differ in whether an Opnd trap is raised on "target".

```
mask[0] := 1- OPCODE[3];  mask[3] := OPCODE[4];
mask[1..2] := OPCODE[6..7];
if CC=CCU and Unordered_trap enabled
   and ( mask[0]=1 or mask_1]=1 )
then Trap"Invalid Operation";
if    mask[0] = 1 and CC = CCG
   or mask[1] = 1 and CC = CCL
   or mask[2] = 1 and CC = CCE
   or mask[3] = 1 and CC = CCU
then P := P + target * 2;
```

Instruction	Mnemonic	Assembler aliases
Branch Never	BRN	
Branch Unordered	BRU	
Branch Equal	BRE	BRZ BREVEN BRNOV
Branch Equal or Unord	BREU	BRZU
Branch Less	BRL	BRM
Branch Less or Unord	BRLU	
Branch Less or Equal	BRLE	BRMZ
Branch Not Greater	BRNG	BRLEU
Branch Greater	BRG	BRP BRODD BROV BRBUSY
Branch Greater or Unord	BRGU	
Branch Greater or Equal	BRGE	BRPZ
Branch Not Less	BRNL	BRGEU
Branch Greater or Less	BRGL	
Branch Not Equal	BRNE	BRGLU
Branch Not Unordered	BRNU	BRGLE
Branch Always	BR	

Traps: CODEBNDSV
FLINV

6.2.6.2 CALL target.r4

Procedure call. A procedure marker is pushed onto the stack and control is passed to "target", interpreted as a 32-bit half-word offset relative to the start of the CALL instruction. CALL requires the procedure to be within the current code object.

```
IIP := 1-IIP; if IIP=1 and PTE=1 then Trap"DBCALL";
S := S + 4;  {pushes garbage}
PUSH4 P[32..63];
PUSH4 Q[32..63];
Q := S;
P := P + target * 2;
```

Traps: STKOVF
CODEBNDSV
DBCALL

6.2.6.3 CALLX loi.r4

External call. A procedure marker is pushed onto the stack and control is passed to the entry point specified in the OD for "loi". "Loi" contains the high 32 bits of a logical address into the target object.

```
IIP := 1-IIP; if IIP=1 and PTE=1 then Trap"DBCALL";
IIP := 0;
PUSH8 Preturn;
(S-4)[0..6] := STATUSA;
PUSH4 Q[32..63];
Q := S;
if OD(loi).TYP <> VisionCode then Trap"CODETYPV";
if STATUSA.XL > OD(loi).PR then Trap"CODERINGV";
STATUSA.XL  := OD(loi).XL;
Ptarget[ 0..31] := loi;
Ptarget[32..61] := OD(loi).EPWO;
Ptarget[62..63] := 0;
P := Ptarget;
```

Traps: STKOVF
CODETYPV
CODEBNDSV
CODERNGV
DBCALL

6.2.6.4 BRX loi.r4

External Branch. Control is transferred to the target indicated in the OD for "loi". Loi contains the high 32 bits of a logical address into the target code object.

```
IIP := 1-IIP;  if IIP=1 and PTE=1 then Trap"DBCALL";
IIP := 0;
Ptarget[0..31]  := loi;
Ptarget[32..61] := OD(loi).EPWO;
Ptarget[62..63] := 0;
if OD(loi).TYP <> VisionCode then Trap"CODETYPV";
if    STATUSA.XL > OD(loi).PR
   or STATUSA.XL > OD(loi).XL
then Trap"CODERINGV";
if (Q-8)[0] = 0 then begin
   (Q-8)[0] := 1;
   (Q-8)[1..2] := STATUSA.XL;
   (Q-12)[0..31] := P[0..31];
   end;
P := Ptarget;
```

Traps: CODETYPV
CODERINGV
CODEBNDSV
DBCALL

6.2.6.5 EXIT

Exit from procedure. This instruction can be used to return from a procedure called with CALL or CALLX. The procedure marker located at Q contains the necessary information to restore the context of the caller. If the caller executed in a different code object than the current one, a number of checks are made.

```
if (Q-8)[0] = 1 then begin
   {external exit}
   Pobject := (Q-12)[0..31];
   Poffset := (Q-8)[8..31], zero-extended;
   ST_return := (Q-8)[0..7];
   if STATUS.XL > OD(Pobject).XL
   then Trap"STKCONSISTV";
   if ST_return.XL > STATUSB.XTL
   then Trap"INSXTL";
   end
else begin
   {internal exit}
   Pobject := P[0..31];
   Poffset := (Q-8)[0..31];
   ST_return := STATUSA;
   end;
Q_offset := (Q-4)[0..31];
if Q_offset < 0 or Q_offset > Q[32..63] - 12
then Trap"STKCONSISTV";
if Poffset[31] = 1
and (implementation chooses to detect this)
then Trap"INSODDP";
Poffset[31] := 0;
S[32..63]   := Q[32..63] - 12;
Q[32..63]   := Q_offset;
P[0..31]    := Pobject;
P[32..63]   := Poffset;
STATUSA     := ST_return; {SIT bit not to
                           take effect until
                           next instruction}
```

Status: restored from marker on external exit
Traps: INSXTL
STKCONSISTV
CODEBNDSV
INSODDP

6.2.6.6 SEXIT

Subroutine exit. This instruction can be used to return from a subroutine called with a PUSH4; BR combination. The value of Q is not affected.

```
POP4  Returnoffs;
if Returnoffs[31] = 1 and {implementation
   chooses to detect this condition}
then Trap"INSODDP";
Returnoffs[31] := 0;
P[32..63] := Returnoffs;
```

Traps: STKUNF
CODEBNDSV
INSODDP

6.2.6.7 BREAK parameter.r4

Breakpoint. This instruction always causes a breakpoint trap. The value of STATUSD.DRL has no effect.

```
Trap"DBBREAKINS";
```

Traps: DBBREAKINS

6.2.6.8 ERROR

Error. This instruction always causes a trap for all users.

```
Trap"INSERROR";
```

Traps: INSERROR

6.2.6.9 NOP

No operation. Continues with the immediately following instruction.

6.2.6.10 CHECKA parameter.r4

Conditional break. If the "CBA" enable bit is set, a trap is taken. If "CBA" is disabled, no Opnd trap should be raised even if "parameter" is somehow illegal; instead "parameter" should be ignored.

```
if STATUSB.CBA = 1 then Trap"DBCHECKA";
```

Traps: DBCHECKA

6.2.6.11 CHECKB parameter.r4

Conditonal break. If the "CBB" enable bit is set, a trap is taken. If "CBB" is disabled, no Opnd trap should be raised even if "parameter" is somehow illegal; instead "parameter" should be ignored.

```
if STATUSB.CBB = 1 then Trap"DBCHECKB";
```

Traps: DBCHECKB

6.2.6.12 CHECKLO source.r4, lobound.r4

Check lower bound. If "source" is less than "lobound", a bounds check trap occurs. The comparison is a two's complement 32-bit compare.

```
if source < lobound then Trap"INSCHKLO";
```

Traps: INSCHKLO

6.2.6.13 CHECKHI source.r4, hibound.r4

Check upper bound. If "source" is greater than "hibound", a bounds check trap occurs. The comparison is a two's complement 32-bit compare.

```
if source > hibound then Trap"INSCHKHI";
```

Traps: INSCHKHI

6.2.7 Interaction with Machine State

6.2.7.1 MOVEfSP4 selector.r1, destination.w4

Move from special register. This selects a certain register or dedicated memory location based on the value of "selector". This register or memory location is then right justified, zero filled and stored in the 32-bit "destination". An INSMOVSPL violation occurs when either the value of the selector does not correspond to any entry in the following list or when the current execute level does not match the level required for reading the selected register.

selector	#bits	req'd XL	Assembler alias
0 condition code	2	3	GetCC
1 rounding mode	2	3	GetRM
2 exit threshold	2	3	GetXTL
3 execute level	2	3	GetXL
4 flpt trap enable	5	3	GetTEFLP
5 int trap enable	2	3	GetTEINT
6 dec trap enable	2	3	GetTEDEC
7 flpt mode	2	3	GetFPCMODE
8 STATUSA	32	3	GetSTATA
9 STATUSB1	32	3	GetSTATB1
10 STATUSB2	32	3	GetSTATB2
11 TRYoffset	32	3	GetTRY
12 cond break A	1	1	GetCBA
13 cond break B	1	1	GetCBB
14 task clock enable	1	1	GetTCE
15 STATUSC1	32	1	GetSTATC1
16 Interrupt Mask	16	1	GetIMR
17 STATUSD	32	1	GetSTATD
22 HASH.PA	32	1	
23 HASH.LENGTH	32	1	
24 PDIR.PA	32	1	
25 PDIR.LENGTH	32	1	

Traps: INSMOVSPL

6.2.7.2 MOVEtSP4 selector.r1, source.r4

Move to special register. This instruction selects a special hardware register or dedicated memory location based on the value of "selector". The value of "source" is stored into this register or location. The least significant bits of "source" are used in the assignment, without any overflow indication. A trap is taken when the selector does not match any of the entries in the following table or if the current ring level does not match the required ring level.

selector	#bits	req'd XL	Assembler Alias
0 condition code	2	3	SetCC
1 rounding mode	2	3	SetRM
2 exit threshold	2	< source	SetXTL
3 flpt trap enable	5	3	SetTEFLP
4 int trap enable	2	3	SetTEINT
5 dec trap enable	2	3	SetTEDEC
6 flpt mode	3	3	SetFPCMODE
7 STATUSB2	32	3	SetSTATB2
8 Q_offset	32	3	SetQ
9 task breakrange LOI	32	3	SetTBR
10 cond break A	1	1	SetCBA
11 cond break B	1	1	SetCBB
12 task clock enable	1	0	SetTCE
13 Interrupt mask	16	0	SetIMR
14 Debug ring level	2	0	SetDRL
15 sys breakrange LOI	32	0	SetSBR

Status: depends on selector
Traps: depends on selector
Opnd
INSMOVSPL
STKCONSISTV (if setting Q offset to value outside SB and S)

6.2.7.3 MOVEfSP8 selector.r1, destinaiton.w8

Move from special register. This instruction is used to obtain the contents of a special hardware register or dedicated memory location identified by the value of "selector". Values of "selector" not represented in the following list cause the trap "INSMOVSPL" to be raised.

selector	#bits	req'd XL	Assembler Alias
0 program counter	64	3	GetP
1 ODT0.LA	64	1	
2 TCB.LA	64	1	GetTCB
3 TCBX.LA	64	1	GetTCBX
4 interval timer	64	1	
5 task clock	64	1	
6 time of century	64	1	
7 QI.LA	64	1	

Traps: INSMOVSPL

6.2.7.4 MOVEtSP8 selector.r1, source.r8

Move to special register. This instruction stores the value of "source" into the special hardware register or dedicated memory location identified by "selector".

selector	#bits	req'd XL	Assembler Alias
0 TCBX.LA	64	0	SetTCBX
1 interval timer	64	0	
2 task clock	64	0	
3 time of century	64	0	
4 QI.LA	64	0	
5 DST descriptor	64	0	
6 CST descriptor	64	0	

Traps: dependent on selector
INSMOVSPL

6.2.7.5 TRY

Mark the stack with the TRYoffset. When paired with UNTRY, TRY supports the try/recover construct of MODCAL. The old value of TRYoffset is pushed onto the stack and the current value of S is recorded in TRYoffset (hence TRYoffset points to the location in the stack where the previous value of TRYoffset is kept). UNTRY is used to undo this sequence. The back chain of TRYoffsets is much like the back chain of Qoffsets but under total software control independent of CALL/CALLX. TRY must not be executed on the ICS.

```
if STATUSC.ICS = 1 then Trap "TRYV";
PUSH4 TRYoffset;
TRYoffset := S[32..63];
```

Traps: STKOVF
TRYV

6.2.7.6 UNTRY destination.w4

Remove one TRY marker. This instruction undoes the action performed by TRY. This causes the previous value of TRYoffset to become reestablished. UNTRY must not be executed when on the ICS. Note that the TRYoffset need not be on top of the stack when UNTRY is executed, nor is it popped off.

```
if STATUSC.ICS = 1 then Trap"TRYV";
Temp[32..63] := TRYoffset;
destination  := TRYoffset - 4;
Temp[0..31]  := S[0..31];
TRYoffset    := (Temp - 4)[0..31];
```

Traps: TRYV
AddressingV

6.2.8 Instructions that interact with the address space

6.2.8.1 PROBE ring.r1, access.r1, address.r8, length.r4

Probe access rights. This instruction sets condition codes dependent on the legality of accessing the address range given by "address" and "length". PROBE tests whether in the ring level specified by "ring" the type of access represented by "access" would be legal everywhere in the logical address range starting at "address" and ending at "address"+"length"-1. Here a negative "length" is treated as 0.

Encodings:	ring	access
0	0	memory_read
1	1	memory_write
2	2	instruction_fetch
3	3	
4	caller's	

Values not in the list above will cause an INSPROBE trap.

The resulting conditon code settings are as follows:
CCL: the object does not exist or the indicated access is illegal.

CCE: the indicated access is legal but the indicated address range is not wholly within the object.

CCG: the indicated access is legal at the indicated privilege level over the entire address range specified.

Status: CC
Traps: INSPROBE

6.2.8.2 TESTREF ppn.r4

Test reference bit. Returns the state of the reference bit for the physical page "ppn" in the condition codes and then clears the reference bit. "Ppn" gives the physical page number. If the reference bit in the PPD for the page is found set, then CCG is returned, otherwise CCE.

The reference bit in the PPD is then cleared.

Any address translation aid (TLB) must synchronize itself with the contents of the PPD as part of the execution of TESTREF.
Note that TESTREF only provides a snapshot: immediately after executing TESTREF some other processor may access the page; this would not be reflected in the condition codes.
Ring 0 privilege is required.

Status: CC
Traps: INSPRIV

6.2.8.3 PDINS ppn.r4

Insert page into PDIR. This instruction takes the Physical Page Descriptor (PPD) identified by the physical page number "ppn" and inserts it in the proper Hash chain. The PPD must be entirely initialized before using this instruction, except for the link field. The Virtual Page Number (VPN) in the PPD itself is used to compute the hash value H that locates the proper chain. The PPD will be inserted as the first link in the chain. No other PPDs in the PDIR will be changed. If the PPD for "ppn" is already linked into a hash chain before PDINS is executed, the results are undefined. PDINS requires ring 0 privilege.

```
    if XL > 0 then Trap"INSPRIV";
    PPDpa := PDIR.PA + 16 * ppn;
    Pp  := (PPDpa)[1..20];      {zero-extend}
    if ppn <> Pp then Trap"ADRPDIR";
    VPN := (PPDpa + 4 )[0..51];
    Bucketpa := HASH.PA + 4 * hash( VPN );
/   Link := (Bucketpa)[0..31];
|   (PPDpa + 12)[0..31] := Link;
\   (Bucketpa)[0..31] := PPDpa;
```

Note: the bracketed portion must be synchronized with other hardware access to the hash bucket in a shared-memory multi-processor system.
Such a system may use bit 0 of the hash bucket (Bucketpa)[0] as a semaphore bit.
This bit must be returned to 0.
See PDDEL for further detail.

Traps: INSPRIV
ADRPDIR

6.2.8.4 PDDEL ppn.r4

Delete from PDIR. The Physical Page Descriptor PPD for the physical page with physical page number "ppn" is removed from its hash chain.
Ring 0 privilege is required.

```
PPDpa := PDIR.PA + 16 * ppn;
VPN := (PPDpa + 4)[0..51];
Linkpa := HASH.PA + 4 * hash( VPN );
repeat
   Oldlinkpa := Linkpa;
   Linkpa := (Linkpa + 12)[0..31];
   if Linkpa = 0 then Trap"ADRPDIR";
   until Linkpa = PPDpa;
(Oldlinkpa+12)[0..31] := (PPDpa+12)[0..31];
```

Notes: (consult carefully when implementing a VISION machine capable of running as a shared-memory multi-processor)

1) Address translation aids (TLB) must be synchronized (by hardware) with the state of the PDIR/HASH before hardware may execute the instruction following PDDEL.

2) In a shared-memory multi-processor system, implementations must guarantee that read-write operands never fault on the write. The burden for ensuring this can be placed entirely on the implementation of PDDEL. This requires PDDEL to complete a handshake with all processors in the system before the instruction following PDDEL executes.

3) Various functions compete for access to hash bucket and PPDs and these functions must be carefully synchronized by hardware. These functions are: address translation; writing dirty/reference bits; PDINS; TESTREF; PDDEL.
Each hash bucket and each PPD has a bit for semaphore use by hardware. It is sufficient to lock the appropriate hash bucket for the entire duration of each function. However, doing so might add overhead to writing dirty/reference bits. The following scheme is also sufficient: when writing dirty/reference bits lock only the PPD; when translating addresses lock hash bucket and each PPD in the chain and unlock each immediately after reading its contents; PDINS locks the hash bucket; PDDEL locks two consecutive links in the chain (starting with the hash bucket) and unlocks the first one only after it has obtained the lock for the third one.
Hardware must unlock all semaphores when a trap occurs.

Traps: ADRPDIR

6.2.8.5 CVLAtVA operand.m1, virtaddr.w8

Convert logical address to virtual address. The virtual address corresponding to the logical address of "operand" is computed and stored in "virtaddr". Level 1 privilege is required. The reference bit for the page containing "operand" is not affected.

Traps: INSPRIV

6.2.8.6 HASH virtaddr.r8, hashindex.w4

Hash address. The 64-bit virtual address "virtaddr" is converted to a hash index which is stored in the 32-bit "hashindex". Level 1 privilege is required. Bits 52..63 of "virtaddr" are ignored.

Traps: INSPRIV

6.2.8.7 CVVAtPP virtaddr.r8, ppn.w4

Convert virtual address to physical page number. The 64-bit "virtaddr" is translated to find the physical page on which it resides. It returns a 20-bit physical page number, right justified and zero-extended. However, if the page is absent, "ppn" is set to -1. Level 0 privilege is required. The reference bit for the addressed page is not affected.

```
VPN := virtaddr[0..51];
if page VPN is currently present then
   ppn := physical_page_number_of_(VPN)
else
   ppn := -1;
```

Traps: INSPRIV

6.2.8.8 GrowGD0 newlength.r4

Grow group zero ODT. This instruction informs hardware that the length of the Object Descriptor Table for group zero has been increased. The Group Descriptor for group zero is updated to reflect this, in all processors in a shared-memory multi-processor system. Ring 0 privilege is required. It is the responsibility of operating system software to ensure that the newly addressable ODs in group zero are properly initialized.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
if   GD0.UB <  GD0.LB + newlength
then GD0.UB := GD0.LB + newlength;
```

Traps: INSPRIV

6.2.9 Instructions for Tasking and Synchronization

6.2.9.1 DISABLE oldi.w1

Disable interrupts.

```
if STATUSA.XL > 1 then Trap;
oldi := STATUSC.IE;
STATUSC.IE := 0;
```

Traps: INSPRIV

6.2.9.2 ENABLE oldi.r1

Enable interrupts.

```
if STATUSA.XL > 1 then Trap;
STATUSC.IE := oldi;
```

Traps: INSPRIV

6.2.9.3 INTERRUPT pr.r4

Cause processor interrupt at priority "pr".

```
if STATUSA.XL > 0 then Trap;
pri := pr[28..31];
IPR[ pri,processor ] := set;
```

Traps: INSPRIV

6.2.9.4 PSDB

Pseudo Interrupt Disable. The Dispatcher Disable Count (DDC) is incremented. This inhibits dispatching of new tasks. It does not disable external interrupts. The PSDB/PSEB pair can be used as a very efficient way to protect critical regions in a uni-processor system. PSDB/PSEB pairs can be nested. Ring 1 privilege is required.

```
if XL > 1 then Trap"INSPRIV";
if DDC < 0 then Trap"INSDDCV";
if DDC > 2**27-1 then Trap"INSDDCV";
DDC := DDC + 1;
```

Traps: INSPRIV
INSDDCV

6.2.9.5 PSEB

Pseudo Interrupt Enable. This instruction removes one inhibition on dispatching new tasks and so undoes the effect of the most recent PSDB. PSEB requires ring 1 privilege. If a dispatch request is pending (DRF = 1), conditions for entering the dispatcher are checked and an attempt is made to enter the dispatcher. These conditions must be satisfied before the dispatcher can be entered:

```
STATUSC.DDC = 0
STATUSC.XM  = 0
STATUSC.ICS = 0
STATUSC.DRF = 1
STATUSC.IE  = 1
```

The PSDB/PSEB can also be used to protect regions within the dispatcher code itself; in this case the DPF flag must be ignored.

```
if XL > 1 then Trap"INSPRIV";
if DDC <= 0 then Trap"INSDDCV";
DDC := DDC - 1;
if      STATUSC = 3  then  DISP
else if STATUSC = 7  then begin
   if STATUSB.DISP = 1 then  DRF := 0;
   end
```

Traps: INSPRIV
INSDDCV

6.2.9.6 DISP

Dispatch. This instruction is used to enter the dispatcher as soon as is practicable. The only way to enter the dispatcher is through this instruction.
If the dispatcher cannot be entered right away, the Dispatch Request Flag is set. Ring 1 privilege is required. The following conditions must hold before the dispatcher can be entered:

```
/  STATUSC.DDC = 0
|  STATUSC.XM  = 0
|  STATUSC.ICS = 0
\  STATUSC.IE  = 1
```

```
if XL > 1 then Trap"INSPRIV";
if STATUSC1 = 1 or STATUSC1 = 3 then begin
   PUSH_INTERRUPT_MARKER;  TCB.SN := S;
   STATUSC.ICS := 1;  STATUSC.DRF := 0;
   execute_case_2_of_IEXIT;
   end
else
   STATUSC.DRF := 1;
```

Status: either unchanged or loaded from Dispatcher marker
Traps: INSPRIV
STKOVF

6.2.9.7 LAUNCH tcbla.r8, tcbva.r8

Launch a task. This instruction is used by the dispatcher to start execution of the task identified by "tcbla" and "tcbva". The new current TCB is located at "tcbla" in logical address space and at "tcbva" in virtual address space. It is the responsibility of operating system software to ensure that "tcbla" and "tcbva" are indeed logical and virtual address to one and the same task control block. Level 0 privilege is required.

```
if STATUSC.ICS=0 then Trap"INSPRIV";
if Q <> QI then Trap"STKCONSISTV";
TCB.LA := tcbla;  TCB.VA := tcbva;
GD1 := TCB.GD1;
 :      :
GD7 := TCB.GD7;
execute_case_1_of_IEXIT;
```

Traps: INSPRIV
STKCONSISTV

6.2.9.8 IEXIT

Interrupt Exit. This is used at completion of an interrupt handler (either external or internal). A trap occurs if the instruction is executed other than on the ICS. Q must either point to the dispatcher marker or to an interrupt marker, otherwise results are unpredictable. If any of the pages of the ICS are absent, results are unpredictable. If IEXIT returns control to a task, the TCB of that task must be resident. If any pages on the task's stack containing the interrupt marker are absent, or if that stack is in a stack overflow condition, the appropriate trap is taken which runs as the bottom routine on the ICS (at QI). Neither TCB nor the task stack object are modified in any way. There are three cases of IEXIT which are sorted as follows:

Case 1: IEXIT should return control to a task without involving the dispatcher.
This case obtains if Q=QI, while DRF=0 or dispatching is otherwise disabled.

Case 2: IEXIT should run the dispatcher to have it select a task to LAUNCH.
This case obtains if DRF=1 (dispatcher request flag), dispatching is not disabled, and no interrupt handler is pending. Note that it is possible for the dispatcher to preempt itself.

Case 3: IEXIT should resume whatever code was running prior to the interrupt handler. This may be a lower priority interrupt handler that was pending or the dispatcher.

The IEXIT description uses these uninterruptible sequences:

```
RESTORE_REGS:      begin  POP8B B5; .. POP8B B0;
                     POP4 X15; .. POP4 X0;  POP8 STATUSB;
                     end

RESTORE_RETURN:    begin  S := Q + 120;  RESTORE_REGS;
                     EXIT;
                     end

RESTORE_HP3000:    begin  `POP2' DelQ;  Q := S - DelQ;
                     `POP8' STATUSB;   `POP2' Z.OFFSET;
                     `POP2' DL.OFFSET; `POP2' DB.OFFSET;
                     `POP2' DB.DST;
                     end
```



```
IEXIT:  if STATUSC.ICS = 0 then Trap"INSPRIV";
        if Q = QI and STATUSC1 <> 7 then begin
case_1:    {return to task}
           STATUSC.ICS := 0;  XM := TCB.XM;
           STATUSC.IE  := 1;
           if XM = 0 then begin
              {return to Vision mode}
              S := TCB.SN[0..63];  Q := S - 120;
              if TCB.SWIP = 0 then RESTORE_RETURN
              else  P := "SWITCHN" trap label;
              TCB.SWIP := 0;
              end
           else begin
              {return to HP3000 mode}
              S := TCB.SC[0..63];
              RESTORE_HP3000;                    \ don't allow
              if TCB.SWIP = 0 then `EXIT 0'      / interrupts
              else  P := "SWITCHC" trap label;
              TCB.SWIP := 0;
              end
           end
        else if Q=QI or (STATUSC1=7 and (Q)[4]=1) then begin
case_2:    {start dispatcher}
           Q := QI;  DRF := 0;
           STATUSB := DispatcherStatusBInit;
           EXIT <<but leave S at Q>>  {Q doesn't change}
           end
        else
case_3:    {resume code running before interrupted}
           RESTORE_RETURN;
```

Note 1: implementations may substitute for the test Q = QI the test (Q-4)[0..31] = QI[32..63].

Note 2: "STATUSC1 = 7" summarizes the condition that dispatching is both desired (DRF=1) and possible (DDC=0, etc).

Status: restored from marker
Traps: INSPRIV
STKUNF
STKCONSISTV
SWITCHN
SWITCHC
AddressingV on all base register loads

6.2.9.9 SWITCH

Switch to HP3000 mode. See chapter 10.5.2.3.

6.2.9.10 RSWITCH

Reverse switch. See chapter 10.5.2.4.

6.2.9.11 IDLE

Idle loop. This instruction will cause no activity visible to software to occur until an external interrupt is raised. In a shared-memory multi-processor, no memory bandwidth should be consumed by this processor when in IDLE. This requires ring 0 privilege.

Traps: INSPRIV

6.2.9.12 STOP

Stop. This instruction will cause the hardware to save all its cached values into their home locations in main memory, then release the memory bus and to wait for a hardware reset or some other condition not defined by this document. The intended use is for stop after power-fail. This instruction requires ring 0 privilege.

Traps: INSPRIV

6.2.9.13 SYNCOD loi.r4

Synchronize changes to an OD. This instruction serves to warn hardware that the Object Descriptor corresponding to the logical object "loi" in the address space of the currently executing task on the processor executing the SYNCOD instruction has been changed. Hardware behavior of all processors in a shared-memory multi-processor system will reflect the new value of the OD no earlier than when the OD is changed in memory and no later than at the completion of the SYNCOD instruction, at the discretion of the hardware implementation.
However, if the logical object whose OD is being affected matches the logical object id of any of the logical addresses in the following list, the effect of SYNCOD is undefined:

P; Q,S; B0,...,B5; TCB.LA; QI;

these reflect operating system errors. Similarly, if the OD change modifies the address or length of the TCB or ICS(QI) of a task currently executing on another processor in the same shared-memory multi-processor system, the effect of SYNCOD is undefined.
SYNCOD requires all other processors in a shared-memory multi-processor system to re-load their current values of P,Q,S,B0..B5 and for them to transfer control to a trap handler if their values are now invalid.
SYNCOD requires ring 0 privilege.

Traps: INSPRIV
AddressingV

6.2.9.14 SYNCTCB tcb.r8

Synchronize task control block. This instruction warns hardware that some Group Descriptors in the Task Control Block at logical address "tcb" have changed. The behavior of address translation hardware will reflect the changes in the Group Descriptors no earlier than when the changes occur in memory and no later than when SYNCTCB is executed with the proper value of "tcb", the exact time being implementation dependent.

Traps: INSPRIV
AddressingV

6.2.9.15 SYNCIB operand.mc, length.r4

Synchronize instruction buffer. This instruction must be used to synchronize hardware whenever code is modified or when code comes into or goes out of existence through ODT modification. "Operand" identifies the first byte affected; "length"(in bytes; if negative, zero is used) indicates how many consecutive bytes are affected. Ring 0 privilege and code access to "operand" is required.

Traps: INSPRIV
CODEODTV
CODEBNDSV

6.2.9.16 TESTSEMA sema.mrw4, result.w4

Semaphore test. This instruction is used for synchronization with other processors in a shared-memory multi-processor system. It is essentially a "test and set". The old value of "sema" is copied into "result". Then bit 0 of "sema" is set to 1. Reading the most significant byte of "sema" and storing back the modified value is all done in a single uninterruptible operation. No memory access on behalf of any processor is allowed to intervene between the read of "sema" and the write of the modified value of "sema".
The other three bytes of "sema" can be fetched either simultaneously to fetching the first byte or later (jointly, or individually) but not before. Condition code CCE is set when bit 0 of "sema" was found clear, CCG is set when bit 0 was found set.
Note: for any restartable trap detected after the first byte has been modified, hardware must restore the first byte to its original value before passing control to the trap handler.

```
Byte[0..7] := sema[0..7];    \ uninterruptible by
sema[0] := 1;                / other processors.
if Byte[0] = 1
then CC := CCG
else CC := CCE;
result[0..7] := Byte;
result[8..31] := sema[8..31];
```

Status: CC

6.2.9.17 MOVESEMA source.r4, sema.mw4

Move semaphore. This instruction copies the value of "source" into "sema" in one indivisible memory operation. No other hardware activity is allowed to cause any part of "sema" to change until MOVESEMA has completed.

```
sema := source;
```

6.2.9.18 DOWN sema.mrw4

Down semaphore (P). This instruction performs the fast path of the Down_semaphore operation (also known as `P') or else traps out to the trap handler for the slow path. "sema" contains a bit for locking out other processors in a shared-memory multi-processor system; it also contains a 31-bit signed integer count. The address of the next instruction and the address of "sema" are passed to the SEMADOWN handler.

```
Label:  TESTSEMA sema, Temp;          \ busy-wait for
        BRBUSY Label;                 / hardware semaphore
        Count := Temp[1..31]; {sign-extended}
        if Count = -2**30 then Trap"SEMAOVF"
        else begin
           Count := Count - 1;
           MOVESEMA  Count, sema
           if Count < 0 then Trap"SEMADOWN"
           end;
```

Status: CC NOT affected
Traps: SEMAOVF
SEMADOWN

6.2.9.19 TESTDOWN sema.mrw4

Test and Down semaphore. This instruction attempts a DOWN, but rather than trap out to software for the slow path, it sets a condition code to reflect this fact and continues.

```
Label:  TESTSEMA  sema, Temp;    \ busy-wait for
        BRBUSY Label;            / hardware semaphore
        Count := Temp[1..31]; {sign-extended}
        if Count = -2**30 then Trap"SEMAOVF"
        else begin
           TEST4  Count;
           if Count > 0 then Count := Count - 1;
           Count[0] := 0;
           MOVESEMA  Count, sema;
           end;
```

Status: CC
Traps: SEMAOVF

6.2.9.20 UP sema.mrw4

UP semaphore (V). This instruction performs the fast path of the Up_semaphore operation (also known as `V') and traps to software for the slow path. "Sema" is a 32-bit quantity in memory that contains a `hardware-semaphore' bit and a 31-bit signed integer count. UP increments the count. When the count remains negative, the slow path is taken. The trap handler presumably launches the first task on the queue of tasks waiting for this semaphore.
Note: for any restartable trap detected after the semaphore word "sema" has been modified, hardware must restore its original value before passing control to the trap handler. The semaphore word is NOT restored when taking the "SEMAUP" trap.

```
Label: TESTSEMA  sema, Temp;      \ busy-wait for
       BRBUSY  Label;             / hardware semaphore
       Count := Temp[1..31] {sign-extended};
       if Count = 2**30-1 then Trap"SEMAOVF"
       else begin
          Count := Count + 1;
          if Count <= 0 then begin
             Count[0] := 1;  {NOT superfluous!}
             MOVESEMA  Count, sema;
             Trap"SEMAUP"
             end
          else MOVESEMA  Count, sema
          end;
```

Status: CC NOT affected
Traps: SEMAOVF
SEMAUP

6.2.10 Arithmetic Conversion

6.2.10.1 ISC42 source.r4, destination.w2

Integer size check. The 16 least significant bits of "source" are moved to "destination". If the 17 most significant bits are not all the same, overflow is raised.

```
destination[0..15] := source[16..31];
if source > 2^15-1 or source < -2^15
then raise integer overflow;
```

Traps: Ovflow

6.2.10.2 CONVERT subopcode.r1, source.r, destination.w

Convert types. "CONVERT" uses a subopcode to determine from what type to what type conversion is desired. The subopcode also controls rounding behavior when any floating point arithmetic is involved.

"Source" is interpreted to be of type "Src_T" and converted to the "Dst_T" type and stored in "destination".

Rnd	meaning
0	round towards nearest unit, (if tie, round to even)
1	round towards negative infinity
2	round towards zero
3	round according to STATUSB.FPC.RM

Src_T, Dst_T	meaning
0	32-bit integer
1	64-bit integer
2	32-bit IEEE floating point
3	64-bit IEEE floating point
4	128-bit IEEE floating point

Conversion to and from decimal data is covered in 6.3.

Status: Ovfl
Unfl
Traps: Arith
FlArith

6.3 Decimal Instructions

6.3.1 Packed Decimal Numbers

The packed decimal number format used in many of the instructions in the cobol and decimal group is described here. External numeric format is described later in this introduction. In packed decimal format, a decimal digit is encoded in a nibble, using bit patterns 0000-1001 to encode 0-9. Two decimal digits are packed to a byte; the least significant decimal digit is packed in a byte together with a nibble encoding the sign. Packed decimal numbers may contain 0-31 digits. This amounts to 1-32 nibbles (counting the sign nibble), or 1-16 bytes. Packed decimal numbers always occupy an integral number of bytes, even if the number of decimal digits is even and therefore the number of nibbles is odd. A packed decimal number with an even number of digits must have a 0000-nibble as its most significant nibble such as to fill out the byte. The address of the packed decimal number is the address of the byte containing the most significant digit. The standard sign nibble has the value 1100 for positive and 1101 for negative. Any other value for the sign nibble may produce unexpected results in packed decimal arithmetic. However, VALD (validate decimal) accepts sign nibbles 0000-1011, 1110 and 1111 as alternatives for positive and will change them to 1100. By software convention, 1111 may be regarded as "unsigned". Decimal arithmetic on packed decimal numbers will always produce results with a standard sign nibble. Negative zero (i.e. a packed decimal number 0 with a negative sign nibble) is not produced by packed decimal arithmetic. Using negative zero in packed decimal arithmetic may produce unexpected results. However, VALD accepts negative zero and will change it to positive zero.

The following comments apply equally to all packed decimal instructions described in sections 6.3.3.1 through 6.3.3.12.

a) It is the responsibility of software to ensure that the "fill-out" nibble in a packed decimal with an even number of digits does indeed have the value zero. Hardware may treat this nibble as a normal significant decimal digit in a source operand. It is the responsibility of the hardware never to introduce a non-zero value in the "fill-out" nibble as a consequence of decimal arithmetic. It is the responsibility of hardware to set the overflow bit (or take the overflow trap) based on whether the resulting packed decimal number fits in the number of decimal digits specified in the instruction. Hardware must never overflow into the "fill-out" nibble.

b) When decimal overflow occurs with the overflow trap enabled, the source operands will be left unchanged by the instruction. A destination operand may receive a value that differs across hardware implementations (unless it coincides with a source operand).

c) It is the responsibility of software to ensure that there is no partial overlap between source operands and destination operands in the same instruction. It is the responsibility of hardware to ensure that total identity between source and destination operand is handled "correctly" (defined as producing the same result as would be obtained by completely pre-reading the source operand into a processor-private temporary area).

d) The length of a packed decimal operand is expressed by giving the number of decimal digits; in other words, the sign nibble is not counted, nor the "fill-out" nibble. Some packed decimal instructions include an explicit operand specifying the length of the packed decimal value; in others the length is implied by the opcode: for these, the length is either 7, 15 or 31 digits, which corresponds to 4, 8 and 16 bytes.
On loading decimal values in registers, they are always left-filled with zeros to reach 7, 15 or 31 digits.
Explicit length operands must be checked by hardware to ensure they are between 0 and 31. The "DECINVL" trap is taken for invalid length operands.

6.3.2 External Decimal Numbers

The external numeric format uses a decimal representation of a number with one digit per byte. each digit is encoded in ASCII (48-57 corresponds to '0' - '9'). The sign is encoded by an "overpunch" in the least significant digit. An external numeric number consists of zero or more leading ASCII blanks followed by zero or more ASCII digits followed by an overpunched ASCII digit. Overpunched digits follows the conventions in the table below:

digit value	positive overpunch	negative overpunch	unsigned
0	'{'	'}'	'0'
1	'A'	'J'	'1'
2	'B'	'K'	'2'
3	'C'	'L'	'3'
4	'D'	'M'	'4'
5	'E'	'N'	'5'
6	'F'	'O'	'6'
7	'G'	'P'	'7'
8	'H'	'Q'	'8'
9	'I'	'R'	'9'
(0			' ')

The CVAD instruction recognizes a number in this external format and converts it to a number in packed decimal representation.
The CVDA instruction converts a number in packed decimal format to external numeric format as described here.
VISION does not directly support some decimal formats known as external numeric with leading overpunched sign, external numeric with trailing separate sign, external numeric with leading separate sign and unsignedexternal numeric. However, three special instructions TESTSTRIP, GETSIGN and OVPUNCH allow these other external numeric formats tobe converted to external numeric with trailing overpunched sign.
TESTSTIP will strip the overpunched sign while recording the original sign information in the condition code. GETSIGN will extract the sign information from an overpunched sign digit and format it as an ASCII sign digit. OVPUNCH combines an unsigned digit and an ASCII sign digit and will produce from them the the corresponding sign-overpunched ASCII character.

6.3.3 Decimal Instruction Set

6.3.3.1 ADDtD term.r, sum.rw

Add decimal. "Term" is added to "sum" and the result is stored in "sum". The operands must be in the standard packed decimal format (such as produced by VALD), otherwise results may differ across implementations. A decimal overflow occurs if all of the digits of the result do not fit in "sum"; if overflow is disabled, the left-truncated result is stored in "sum", else "sum" is left unchanged.

	ADD4D	ADD8D	ADD16D
Status:	Ovfl	Ovfl	Ovfl
Traps:	Opnd	Opnd	Opnd
	DECOVF	DECOVF	DECOVF

6.3.3.2 SUBtD term.r, difference.rw

Subtract decimal. "Term" is subtracted from "difference" and the result is stored in "difference". The operands must be in the standard packed decimal format (such as produced by VALD), otherwise results may differ across implementations. A decimal overflow occurs if all of the digits of the result do not fit in "difference"; if overflow is disabled, the left-truncated result is stored in "difference", else "difference" is left unchanged.

	SUB4D	SUB8D	SUB16D
Status:	Ovfl	Ovfl	Ovfl
Traps:	Opnd	Opnd	Opnd
	DECOVF	DECOVF	DECOVF

6.3.3.3 MPYtD factor.r, product.rw

Multiply decimal. "Factor" is multiplied by "product" and the result is stored in "product". The operands must be the standard packed decimal format (such as produced by VALD), otherwise the results may differ across implementations. A decimal overflow occurs if all of the digits of the result do not fit in "product"; if overflow is disabled, the left-truncated product is stored in "product", else "product" is left unchanged.

	MPY4D	MPY8D	MPY16D
Status:	Ovfl	Ovfl	Ovfl
Traps:	Opnd	Opnd	Opnd
	DECOVF	DECOVF	DECOVF

6.3.3.4 DIVtD divisor.r, quotient.rw

Divide decimal. "Quotient" is divided by "divisor" and the result is stored in "quotient". The operands must be in the standard packed decimal format (such as produced by VALD), otherwise the results may differ across implementations. DIVD truncates, i.e. it rounds towards zero. Decimal overflow occurs if all of the digits of the result do not fit in "quotient"; if overflow is disabled, the left-truncated result is stored in "quotient", else "quotient" is left unchanged. If "divisor" is zero, the result is indeterminate.

	DIV4D	DIV8D	DIV16D
Status:	Ovfl	Ovfl	Ovfl
Traps:	Opnd	Opnd	Opnd
	DECDVDZ	DECDVDZ	DECDVDZ

6.3.3.5 CMPtD sourcea.r, sourceb.r

Compare decimal. The condition code in the status word is set depending on the result of the comparison of the decimal values of "sourcea" and "sourceb". Both operands must be in the standard packed decimal format (such as produced by VALD), otherwise results may differ across implementations.
The condition code is set to:

CCG, if sourcea > sourceb,
CCL, if sourcea < sourceb,
CCE, if sourcea = sourceb.

	CMP4D	CMP8D	CMP16D
Status:	CC	CC	CC
Traps:	Opnd	Opnd	Opnd

6.3.3.6 TESTtD source.r

Test decimal. This is a short form of CMPtD source, 0.

	TEST4D	TEST8D	TEST16D
Status:	CC	CC	CC
Traps:	Opnd	Opnd	Opnd

6.3.3.7 SLD count.r1, length.r1, source.r, dest.w

Shift left decimal. The packed decimal value in "source" is shifted left by "count" decimal places and the result is stored in "dest". Both "source" and "dest" have "length" digits. This is equivalent to a MOVED from "source" to "dest" followed by a MPYD of "dest" by a power of ten.

Status: Ovfl

6.3.3.8 SRD count.r1, length.r1, source.r, dest.w

Shift right decimal. The packed decimal value in "source" is shifted right by "count" decimal places and the result is stored in "dest". Both "source" and "dest" have "length" digits. This is equivalent to a MOVED from "source" to "dest" followed by a DIVD of "dest" by a power of ten.

Status: Ovfl

6.3.3.9 MOVED length.r1, source.r, dest.w

Move decimal. The packed decimal "source" with "length" decimal digits is moved to "destination" of the same length.
If "source" is in a register, only the least significant "length" digits are moved, with overflow indication if any of the most significant digits in the register (register pair, quad) are non-zero.
If "dest" is in a register, the decimal number is padded with zero digits to fill up either 1,2 or 4 registers.

6.3.3.10 VALD length.r1, operand.rw

Validate decimal. The packed decimal string "operand" is checked for validity as a decimal number. If "length" is even, the "fill-out" nibble is made zero. Each digit is checked to be in the range 0000-1001. The sign nibble is made standard by replacing 0000-1001 , 1110 or 1111 with the value 1100. If all digits are zero, but the sign is negative, the sign is changed to positive. "Length" indicates the length in digits of the packed decimal number.

Traps: DECINVL
DECINVDG

6.3.3.11 CVDI length.r1, source.r, dest.w8

Convert packed decimal to integer. The packed decimal number in "source", with "length" decimal digits, is converted to a two's complement 64-bit integer. The result is stored in "dest". "Source" must be in standard packed decimal format (such as produced by VALD), otherwise results may differ across implementations.

Traps: Ovfl

6.3.3.12 CVID length.r1, source.r8, dest.w

Convert integer to packed decimal. The 64-bit two's complement integer value in "source" is converted to a number in packed decimal format and padded or truncated to fit "length" decimal digits. The result is stored in "dest". If "dest" is in a register, the result is further padded to occupy 4,8, or 16 bytes.

Traps: Ovfl

6.3.3.13 TESTSTRIP operand.rw1

Test and strip sign from overpunched ASCII digit. The ASCII digit "operand" is changed to an unsigned digit according to the table below. The condition code is set to CCG if the digit is found in the positive column, to CCL if the digit is found in the negative column and to CCE if it is found in the unsigned column. If "operand" is not in the table, an invalid digit trap occurs.

positive	negative	unsigned	all become:
`{'	`}'	`0'	`0'
`A'	`J'	`1'	`1'
`B'	`K'	`2'	`2'
`C'	`L'	`3'	`3'
`D'	`M'	`4'	`4'
`E'	`N'	`5'	`5'
`F'	`O'	`6'	`6'
`G'	`P'	`7'	`7'
`H'	`Q'	`8'	`8'
`I'	`R'	`9'	`9'
		(` '	`0')

Status: CC

6.3.3.14 GETSIGN operand.r1, sign.w1

Get sign. The sign-overpunched ASCII digit "operand" is examined and its sign (according to the table above) is recorded in "sign", using ASCII `+' for positive, `-' for negative and ` ' for unsigned. Invalid values for "operand" will generate an invalid digit trap.

Traps: DECINVDG

6.3.3.15 OVPUNCH sign.r1, operand.rw1

Create digit with overpunched sign. "Sign" must be ASCII `+', `-' or ` ' (blank). "Operand" must be one of the ASCII digits from `0' to `9'. "Operand" is changed into the corresponding element of the positive column or the negative column of the table above, depending on "sign". If "sign" is ` ', no change occurs. Invalid values for "sign" or "operand" generate an invalid digit trap.

Traps: DECINVDG

6.3.3.16 VALN length.r1, operand.rw

Validate external numeric decimal. This instruction checks to see if the external numeric decimal "operand" obeys the following format: zero or more ASCII blanks followed by zero or more ASCII encoded digits the last one of which is optionally overpunched with a sign according to the table on the previous page. "Length" indicates the length of "operand" in bytes. All leading blanks are converted into ASCII `0'. Negative zero is converted to positive zero.

Traps: DECINVDG

6.3.3.17 CVAD length.r1, source.r, dest.w

Convert external numeric decimal to packed decimal. The "source" of "length" bytes is interpreted as an external numeric decimal number with trailing overpunched sign and converted to packed decimal format. The packed decimal result is padded to 4,8 or 16 bytes (no more than needed given "length") and stored in "dest". If source does not obey the format checked for and produced by VALN, indeterminate results occur.

6.3.3.18 CVDA length.r1, source.r, dest.w

Convert packed decimal to external numeric decimal. The "source" is interpreted as a packed decimal number. The "dest" is an external numeric decimal of "length" bytes. The length of "source" is 4,8 or 16 bytes, as derived from "length". If "source" does not obey the format checked for and produced by VALD, indeterminate results occur.

6.4 Vector Instruction Set

This section describes the vector instruction set. Vector Registers and the vector context save area are described in chapter 5.

Vector instructions are all in a secondary instruction set, in the "VECTOR" escape group. Opcode assignments are shown in section 6.1.4.

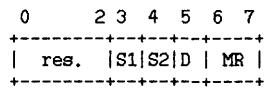
A memory vector is an array of values that are evenly spaced in memory. An example would be a row or a column of a matrix in a Fortran program. Vector instructions can perform operations (such as addition) on entire memory vectors, at speeds higher than a corresponding software sequence. Vector instructions can also perform operations between memory vectors and scalars (e.g. multiplying all elements of a memory vector by two). Memory vectors are therefore characterized by their starting address, their number of elements, and the distance between consecutive elements. This distance between elements (in bytes) is called the stride of the vector. A memory vector with a stride of zero degenerates to a scalar.

A major feature of the VISION architecture is the inclusion of vector registers. A vector register can be loaded with all or part of a memory vector, offering the potential of eliminating even memory access speed as a limit on vector performance.

Most vector instructions operate on vector operands, which are either vector registers or memory vectors or scalar registers (X0..X15). Vector operands are indicated by the vector attribute(".v"). Under the vector attribute, an operand descriptor for a literal (short or long) is given a different meaning: the least significant 3 bits of the literal are interpreted as a vector register number, selecting VR0..VR7. Under the vector attribute, a memory operand is re-interpreted as a memory vector, as detailed below. A register operand is interpreted as a scalar.

To encode a memory vector, either one or two operand descriptors are needed. The second operand descriptor is needed when a memory vector has a stride different from the default value. The default stride is such that the memory vector is entirely contiguous in (virtual) memory.
The first operand descriptor of a memory vector (treated as a ".m" operand) designates the starting address of the vector.

The vector opcode of a vector instruction does not uniquely determine how many operand descriptors participate in the instruction. A special first operand, called vector qualifier, contains the necessary information. This operand must be encoded as a short literal. Its value is interpreted as follows:



where:

S1 -- first source stride. If one, the first source operand uses an explicit stride. This implies that two operand descriptors are involved in this source operand.

S2 -- second source stride. If one, the second source operand uses an explicit stride.

D -- destination stride. If one, the destination operand uses an explicit stride.

MR -- mask register. Selects one of four mask registers. Their function is detailed below.

res -- reserved. Hardware masks out this field.

The vector qualifier determines which operands carry explicit strides. These strides are encoded as ".r4" operands that follow all other operands in the vector instruction.

An explicit stride is only meaningful if the corresponding operand is indeed a memory operand, indicating a memory vector. A vector register specifier or a scalar cannot make use of an explicit stride; in this case the bit in the vector qualifier is ignored.

6.4.1 Boundary conditions

a) Any overlap between source and destination will produce results that may differ across implementations. Total identity of source and destination operands is allowed. For example, software may expect

```
VADD4  B5.0, B5.0, B5.0
```

to result in all values of the array at B5 to get doubled. However, software should not expect

```
MOVE4  1, B5.0
MOVE4  1, B5.4
VADD4  B5.0, B5.4, B5.8
```

to compute the Fibonacci series.

b) Any overlap within the destination vector itself due to small values of the stride will produce results that may differ across implementations.

c) All vector operations are interruptible. Chapter 5 provides more detail.

d) For vector operations that have corresponding operations on scalars in the base instruction set, the values returned on e.g. overflow are the same as in the base instruction set with overflow trap disabled. If the trap is enabled, the vector operation stops as soon as the condition occurs on an element, and identifying information is passed to the trap handler.

6.4.2 Vector Arithmetic Operations

6.4.2.1 VMOVEt vqual.r1, source.vr, dest.vw

Vector move.

includes: VMOVE2 VMOVE4 VMOVE8 VMOVE16

6.4.2.2 VADDt vqual.r1, terma.vr, termb.vr, sum.vw

Vector add. Elements of "sum" are set to the sum of elements of "terma" and "termb".

includes: VADD4 VADD8 VADD4F VADD8F VADD16F

6.4.2.3 VSUBt vqual.r1, terma.vr, termb.vr, diff.vw

Vector subtract. Element-wise difference of "terma" and "termb" is stored in "diff" vector.

includes: VSUB4 VSUB8 VSUB4F VSUB8F VSUB16F

6.4.2.4 VMPYt vqual.r1, facta.vr, factb.vr, prod.vw

Vector multiply. Element-wise product of "facta" and "factb" is stored in "prod" vector.

includes: VMPY4 VMPY8 VMPY4F VMPY8F VMPY16F

6.4.2.5 VDIVt vqual.r1, divd.vr, divsr.vr, quot.vw

Vector divide. Element-wise division of "divd" by "divsr" with the result being stored in "quot".

includes: VDIV4 VDIV8 VDIV4F VDIV8F VDIV16F

6.4.2.6 VNEGt vqual.r1, source.vr, neg.vw

Vector Negate. Element-wise subtract of "source" from zero, storing the result in the vector "neg".

includes: VNEG4 VNEG8 VNEG4F VNEG8F VNEG16F

6.4.2.7 VABSt vqual.r1, source.vr, abs.vw

Vector Absolute. Element-wise absolute value (negate if negative), storing the result in the vector "abs".

includes: VABS4 VABS8 VABS4F VABS8F VABS16F

6.4.2.8 VREMt vqual.r1, divd.vr, divsr.vr, rem.vw

Vector remainder. Element-wise remainder of division of "divd" by "divsr" is stored in "rem".

includes: VREM4 VREM8

6.4.2.9 VMODt vqual.r1, divd.vr, divsr.vr, mod.vw

Vector modulus. Element-wise modulus of division of "divd" by "divsr" is stored in "mod".

includes: VMOD4 VMOD8

6.4.2.10 VLSLt vqual.r1, shiftcount.vr, target.vrw

Vector logical shift left. Element-wise left shift of the vector "target", leaving the result in "target". Note that the shiftcount itself is a vector.

includes: VLSL4 VLSL8

6.4.2.11 VLSRt vqual.r1, shiftcount.vr, target.vrw

Vector logical shift right. Element-wise right shift of the vector "target", leaving the result in "target". Note that the shiftcount itself is a vector.

includes: VLSR4 VLSR8

6.4.2.12 VASLt vqual.r1, shiftcount.vr, target.vw

Vector arithmetic left shift. Element-wise arithmetic left shift of the vector "target", leaving the result in "target". "Shiftcount" is itself a vector.

6.4.2.13 VASRt vqual.r1, shiftcount.vr, target.vw

Vector arithmetic right shift. Element-wise arithmetic right shift of the vector "target", leaving the result in "target". "Shiftcount" is itself a vector.

6.4.3 Vector Logical Operations

6.4.3.1 VAND4 vqual.r1, facta.vr, factb.vr, and.vw

Vector "AND". Computes element-wise and bit-wise "AND" of the vectors "facta" and "factb" and stores the result in the vector "and".

6.4.3.2 VOR4 vqual.r1, terma.vr, termb.vr, or.vw

Vector "OR". Computes element-wise and bit-wise "OR" of the vectors "terma" and "termb" and stores the result in the vector "or".

6.4.3.3 VXOR4 vqual.r1, terma.vr, termb.vr, xor.vw

Vector "XOR". Computes element-wise and bit-wise exclusive "OR" of the vectors "terma" and "termb" and stores the result in the vector "xor".

6.4.4 Vector Compare and Vector/Scalar Hybrids

6.4.4.1 VCMPt vqual.r1, field.r1, srca.vr, srcb.vr, mrsel.r1

Vector Compare. "Field" indicates the type of compare (>, >=, etc.) to be performed. The four least significant bits of "field" indicate G,L,E,U respectively. All elements of "srca" are compared with the corresponding elements of "srcb" and the corresponding bit of the mask register (selected by the mask register selector "mrsel" is set to one if the comparison holds.

includes: VCMP4 VCMP8 VCMP4F VCMP8F VCMP16F

6.4.4.2 VACCt vqual.r1, terms.vr, sum.rw

Vector Accumulate. Adds all elements of "terms" to the old value of "sum".

includes: VACC4 VACC8 VACC4F VACC8F VACC16F

6.4.4.3 VACCDt vqual.r1, terms.vr, sum.rw

Vector Accumulate (Double Precision). Adds all elements of "terms" to the old value of "sum". "Sum" has double the number of bytes of "terms".

includes: VACCD4F VACCD8F

6.4.4.4 VMAXELt vqual.r1, terms.vr, maxind.w4

Find maximum element of vector. "Maxind" is set to the index of the maximum element of the vector "terms". In case of ties, the index of the earliest is chosen.

includes: VMAXEL4 VMAXEL8 VMAXEL4F VMAXEL8F VMAXEL16F

6.4.4.5 VMINELt vqual.r1, terms.vr, minind.w4

Find minimum element of vector. "Minind" is set to the index of the minimum element of the vector "terms". In case of ties, the index of the earliest is chosen.

includes: VMINEL4 VMINEL8 VMINEL4F VMINEL8F VMINEL16F

6.4.4.6 VEXTt vqual.r1, terms.vr, index.r, value.w

Extract element from vector. The element of "terms" at index "index" is fetched and stored into the scalar "value".

includes: VEXT4 VEXT8 VEXT16

6.4.4.7 VINSt vqual.r1, terms.vw, index.r, newval.r

Insert element into vector. The element of "terms" at index "index" is modified to reflect the value "newval".

includes: VINS4 VINS8 VINS16

6.4.4.8 VCOMPRSt vqual.r1, terms.vr, compressed.vw

Compress vector. The mask register indicated in "vqual" governs which elements of "terms" to keep and which to discard. The kept elements are collected in "compressed". VCOMPRS will work correctly in place, i.e. when "terms" and "compressed" are memory vectors with identical starting address and identical stride.

includes: VCOMPRS4 VCOMPRS8 VCOMPRS16

6.4.4.9 VEXPNDt vqual.r1, terms.vr, expanded.vw

Expand vector. The mask register indicated in "vqual" governs which elements of "expanded" are set to elements of "terms" and which to set to zero. The order of the elements of "terms" is preserved. If VEXPND is used in place, results are indeterminate.

includes: VEXPND4 VEXPND8 VEXPND16

6.4.4.10 VGATHt vqual.r1, source.vr, index.vr, destination.vw

Vector Gather. Contiguous elements of "destination" are set to to those elements of "source" indexed by the contiguous elements of "index". The mask register indicated by "vqual" applies to "destination" and "index". "Index" is in units of bytes.

includes: VGATH4 VGATH8 VGATH16

6.4.4.11 VSCATt vqual.r1, source.vr, index.vr, destination.vw

Vector Scatter. Contiguous elements of "source" are put in those elements of "destination" indexed by the contiguous elements of "index". The mask register indicated by "vqual" applies to "source" and "index". "Index" is in units of bytes. Those elements of "destination" not indexed are left unchanged.

includes: VSCAT4 VSCAT8 VSCAT16

6.4.5 Vector Housekeeping

6.4.5.1 RVLR

Reduce Vector Length Register. The vector length register VLR is reduced by the current segment length. Condition codes are set to reflect the new value of VLR. See section 5.xx for details.

Status: CC

6.4.5.2 LDVLR source.r4

Load vector length register.

6.4.5.3 STVLR dest.w4

Store vector length register.

6.4.5.4 VINVAL vrmask.r1

Vector invalidate. The Vector Registers corresponding to ones in the 8-bit "vrmask" have their active lengths set to zero.

6.4.5.5 UVCSA

Update vector context save area. The values of the vector registers are stored in the Vector Context Save Area.

6.4.5.6 PUVCSA tcb.mr

Privileged update of VCSA. The values of the vector registers are stored in the Vecor Context Save Area of the designated task ("tcb" points to the Task Control Block of this task). PUCSVA requires ring 0 privilege.

Traps: INSPRIV

6.4.5.7 IVB tcb.mr

Invalidate vector bank. Invalidates vector bank belonging to the designated task. Ring 0 privilege required.

6.4.5.8 LVB tcb.mr

Load vector bank. Load the vector bank corresponding to the designated task from its VCSA. Ring 0 privilege required.

6.4.6 Operations on Mask Registers

In the following instructions, "mrselect" is an operand that selects one of the four Vector Mask Registers. Only bits mrselect[6..7] are relevant. Bits mrselect[0..5] are ignored. Note that the Mask Registers are at most 256 bits long.

6.4.6.1 CLRMR mrselect.r1

Clear mask register. Set selected mask register to all zeros.

6.4.6.2 STMR mrselect.r1, destination.w16

Store mask register. Store selected mask register in "destination".

6.4.6.3 LDMR mrselect.r1, source.r16

Load mask register. Load selected mask register from the value in "source".

6.4.6.4 MRNOT mrselect.r1

Complement mask register. Change all zeros in the selected mask register to ones and vice versa.

6.4.6.5 MRAND mrasleect.r1, mrbselect.r1

"AND" mask registers. Zero out all bits in mask register "mrb" that have zeros in the corresponding location in mask register "mra".

6.4.6.6 MROR mraselect.r1, mrbselect.r1

"OR" mask registers. Set all bits in mask register "mrb" to one that have ones in the corresponding location in mask register "mra".

6.4.6.7 MRXOR mraselect.r1, mrbselect.r1

"XOR" mask registers. Complement all bits in mask register "mrb" that have a one in the corresponding location in mask register "mra".

6.4.7 Vector Conversion

6.4.7.1 VCONVERT vqual.r1, typer.r1, source.vr, dest.vw

Vector Conversion and Round. This instruction allows vector conversion from one type to another as specified in "typer". The vector "source" is converted and the result is stored in the vector "dest".
Bits typer[2..4] determine the type of "source",
bits typer[5..7] determine the type of "dest".
Bits typer[0..1] are ignored.
Data types are encoded in "typer" as follows:

0	4-byte integer
1	8-byte integer
2	4-byte IEEE floating point
3	8-byte IEEE floating point
4	16-byte IEEE floating point
5-7	illegal

All conversions, including conversions from floating point numbers to integers, obey the rounding mode in STATUSB.FPC.RM.

6.5 I/O Instructions

This sections details the instructions that deal with the I/O backplane and the I/O channels.

6.5.1 PICMB-based VISION systems

The PICMB instruction set can be broken down into two levels, primitives (level 1) and functions (level 2). These classes correspond to the PICMB protocol as defined in the PICMB ERS. The primitives represent the lowest level of activity on the PICMB; hence routines which make use of these must perform all bus protocols themselves. The function level instructions represent various functional combination of primitives. This level performs some of the PICMB protocol for the programmer, while retaining maximum flexibility. These two levels provide complete functionality for communicating with external devices. Each of these levels will be described in detail. The notation used in the following descriptions is intended to reflect the actual operation of the various data and control lines.

6.5.1.1 PICMB Primitives

6.5.1.1.1 IFC

Interface Clear. Causes hardware to assert the Interface Clear Line for one bus cycle.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
IFC := true;
```

6.5.1.1.2 WCMD command.r1

Write command. Send a command byte to the channel adapter.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
cobegin
   CDF := true;
   PICMB.CB.DATA := command[0..7];
   coend;
```

6.5.1.1.3 WBYTE data.r1, end.r1

Write byte. Causes hardware to write a byte to the channel adapter. If "end" has a non-zero value, then the END line will also be asserted. If the channel does not assert the SRT line within x milliseconds, the byte will not be transfered and a timeout condition will be indicated by setting the condition code to CCL. A successful transfer will be indicated by CCE.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
if not SRT then CC := CCL
else begin
   CC := CCE;
   cobegin
      if end <> 0 then END := true;
      PICMB.CB.DATA := data
      coend
   end;
```

Status: CC

6.5.1.1.4 RBYTE data.w1

Read byte. Causes hardware to read a byte from the channel adapter into "data". If the channel does not assert the SRT line within x milliseconds the data will not be read and a timeout condition will be indicated by setting the condition code to CCL. A successful transfer will be indicated by CCE. A successful transfer when the END signal is asserted will be indicated by CCG.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
if not SRT then CC := CCL
else begin
   data := PICMB.CB.DATA;
   if END then CC := CCG
   else CC := CCE
   end;
```

Status: CC

6.5.1.2 Functional PICMB Instructions

The level 2 instructions correspond to the PICMB.CB commands as specified in the PICMB ERS. Some of these commands are global in nature and others are local. The global commands affect all channels in a system and require no channel address. The local commands are directed to a specific channel and require a channel address operand. When a local command is executed, all channels in the system which are not addressed will go into an idle state until a global command is issued or until they are locally addressed. Global commands often return an 8-bit vector; the PICMB supports up to 8 channels.

6.5.1.2.1 CHNOP

Channel no operation. A NOP command is issued to all channels.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
WCMD  0;
```

6.5.1.2.2 RCL response.w1

Roll Call. This command is issued to all channels.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
WCMD  !10;
```

6.5.1.2.3 PRD response.w1

Poll Ready for Data. This command is issued to all channels.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
WCMD  !20;
RBYTE  response;  <<no timeout>>
```

Status: CC

6.5.1.2.4 PDA response.w1

Poll Data Available. This command is sent to all channels.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
WCMD  !30;
RBYTE  response;  <<no timeout>>
```

Status: CC

6.5.1.2.5 PAR response.w1

Poll Attention Requests. This command is sent to all channels.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
WCMD  !40;
RBYTE  response;  <<no timeout>>
```

Status: CC

6.5.1.2.6 RDP channel.r1, dest.w16, length.w1

Read Data Packet. This command is sent to the designated channel and the data packet received is stored in "dest". "length" will be set to the number of bytes in the data packet; if this is less than 16, the remainder of "dest" will not be changed.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
PDA  Temp[0..7];
Ch := channel AND 7;
if Temp[Ch] = 0 then CC := CCL
else begin
   Cmd := !50 + Ch;
   WCMD  Cmd;
   C := 0;
   repeat
      RBYTE  (dest+C)[0..7];
      C := C + 1;
      until CCL or CCG or C>15;
   length := C;
   end;
```

Status: CC

6.5.1.2.7 WDP channel.r1, data.r16, length.rw1

Write Data Packet. This command is sent to the specifically designated channel. The first "length" bytes of "data" is sent to the channel.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
PRD  Temp[0..7];
if Temp[Ch] = 0 then CC := CCL
else begin
   Cmd := !60 + Ch;
   WCMD  Cmd;
   C := 0;
   repeat
      cobegin
         if C = length then END := true;
         WBYTE  (data+C)[0..7];
         coend;
      C := C + 1;
      until C>=length or C>15 or CCL;
   if CCL then length := C;
```

Status: CC

6.5.1.2.8 RIS channel.r1, status.w1

Read Immediate Status. This command is sent to the specifically designated channel. Its status is returned and stored in "status".

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cmd := !70 + Ch;
WCMD  Cmd;
RBYTE  status;
```

Status: CC

6.5.1.2.9 CIS channel.r1, status.r1

Clear Immediate Status. This command is sent to the designated channel. The 8-bit status byte of the addressed channel is cleared in all bit positions corresponding to a zero in "status".

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cmd := !80 + Ch;
WCMD  Cmd;
WBYTE  status;
```

Status: CC

6.5.1.2.10 SIS channel.r1, status.r1

Set Immediate Status. This command is sent to the designated channel. The 8-bit status byte of the addressed channel is set in all bit positions corresponding to a one in "status".

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cmd := !90 + Ch;
WCMD  Cmd;
WBYTE  status;
```

Status: CC

6.5.2 MPB-based systems

The MPB is the memory-processor-bus defined and designed by STO in Colorado.

All I/O instructions for MPB based systems are "local": they address a specific channel by specifying the channel number. The channel number is mapped at configuration time to a slot number on the MPB backplane. This ranges from 0..7. Though the MPB backplane and the IOP channel were designed as a pair, the Vision I/O instructions for the MPB are designed to allow channels other than the IOP to connect to the MPB.

6.5.2.1 MPB-based Instructions

6.5.2.1.1 IOW channel.r4, control.r4, data.r4

I/O Write. This writes the data word "data" to the channel designated by "channel". "Control" is a modifier interpreted by the channel.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cntr := control;
Cntr[0..5] := 0;
Cntr[19..21] := MPB_channel_number_of_
                    originating_CPU;
Write Ch, Cntr, Data  {to MPB};
```

6.5.2.1.2 IOR channel.r4, control.r4, data.w4

I/O Read. This reads the data word "data" from the channel designated by "channel". "Control" is a modifier interpreted by the channel.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cntr := control;
Cntr[0..5] := 0;
Cntr[19..21] := MPB_channel_number_of_
                    originating_CPU;
Read Ch, Cntr, Data  {from MPB};
```

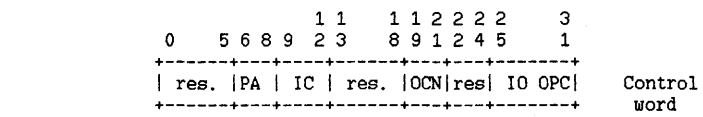
6.5.2.1.3 IOC channel.r4, control.r4

I/O Control. This performs a control function on the channel designated by "channel". "Control" is a modifier interpreted by the channel.

```
if STATUSA.XL > 0 then Trap"INSPRIV";
Ch := channel AND 7;
Cntr := control;
Cntr[0..5] := 0;
Cntr[19..21] := MPB_channel_number_of_
                    originating_CPU;
Control Ch, Cntr  {on MPB};
```

6.5.2.2 Interpretation of the control word on the IOP

The control word detailed in the previous section has a well-defined meaning when used with the IOP channel. This is sketched below. More detail is available from the FOCUS I/O ERS.



Here:

res. = reserved

PA = either subchannel number or device adapter number. The IO opcode "IO OPC" will decide which.

IC = interface control. This four bit field allows for control of the HPIO lines shown below:
IC1 -> BP[0] HPIO bus primitive/interface control
IC2 -> CEND HPIO channel end
IC3 -> CBYT HPIO channel byte
IC4 -> BP[1] HPIO bus primitive/interface control

OCN = channel number of cpu originating the command

IO OPC= IO opcode. This 7-bit value is defined as shown in the following section.

6.5.2.3 IOP Opcodes

6.5.2.3.1 Read commands

Name	Mnemonics	Hexadecimal
Read DMA Current Address	RDA	7
Read DMA Current Count & Status	RDCS	8
Read DMA Mask	RDMK	B
Read Interrupt Mask	RIMK	16
Read Interrupt Request	RIRQ	1C
Interface Poll	IFPL	27
Read DMAPA	RDPA	2D
Read Data Buffer	RDB	2E
Read IOP Revision	RIRV	2F
Read Interface Status & Flag	RISF	3F
Read Interface Status	RIST	43
Read Interface Flag	RIFG	44
Read DMA Termination Field	RDTF	45
Read IOP registers & Suspend	RIRS	48
Read Interface Device End & Burst Request	RDEB	4E
Read Interface FRn	RIF n	(8*n + 1)

6.5.2.3.2 Write Commands

Name	Mnemonics	Hexadecimal
Write DMA Status	WDS	3
Write DMA Count	WDC	4
Write DMA Start Address	WDA	5
Write DMA Termination Field	WDTF	6
Write Interrupt Mask	WIMK	15
Write Interrupt Message	WIMG	1B
Set Interrupt Level	SIL	1D
Write DMAPA	WDPA	2C
Write MPB Channel Number	WMCN	30
Write Attention Poll Mask	WAMK	46
Write IOP Registers & Resume	WIRR	4B
Write Interface FRn	WIF n	(8*n + 2)

6.5.2.3.3 Control Commands

Name	Mnemonics	Hexadecimal
Clear DMA Mask	CDMK	C
Enable DMA	EDMA	D
Disable DMA	DDMA	E
Start DMA, Enable Interrupt & Clear IRQ	SDEC	F
Start DMA & Clear IRQ	SDC	10
Start DMA	SD	13
Start RAMD, Clear IRQ	SRC	14
Request Interrupt	RINT	17
Clear Interrupt Request	CIRQ	18
Enable Interrupt	EINT	1E
Disable Interrupt	DINT	1F
Reset I/O Bus	RIOB	28
Reset IOP	RIOP	2B
Clear Address Lockout Mode	CALM	33
Set Address Lockout Mode	SALM	34
Initiate Cmd Exec from Interrupt	ICEI	38
Disable IOP Command Execution	DCE	3B
Disable Command Execution & Enable Interrupt	DCEI	3C
Turn LED On	LON	3D
Turn LED Off	LOFF	3E
Clear Attention Acknowl. Bit	CAAK	47

6.5.2.3.4 IOP Command Execution

Name	Mnemonics	Hexadecimal
Increment And Branch		35
Skip on Status False		36
Skip on Flag False		37
Write RIF Result to Memory	WRIF	4C
Write Count & Status to Memory	WCSM	4D

6.6 Diagnostics Interface

6.6.1 MOVEtCSP messlen.r4, messpa.r4, replylen.r4, replypa.r4, error.w1

Move message to CSP. Send to the CSP (Control and Support Processor) a message consisting of a string of bytes, "messlen" long, starting at physical address "messpa". An area in physical memory is reserved for a reply, if there is one, of length "replylen" (in bytes), starting at physical address "replypa". Message and reply format may differ across VISION implementations. Refer to the "Protocol of the Control and Support Processor for VCF60 and VCF50" document. Receipt of a reply from the CSP will generate an internal interrupt IICSPREPLY. Actual transmission of the message and the reply may occur at any time between execution of the MOVEtCSP instruction and the IICSPREPLY. During this interval, the message and reply areas in physical memory must not be accessed by software. This isntruction requires Ring 0 privilege. "Error" can have the following values:

0 = Instruction accepted, transmission beginning
* 1 = This VISION implementation has no CSP
* 2 = CSP busy, cannot accept a message
* 3 = Requested operation not implemented by CSP
* 4 = message or reply area wrong physical address or length
5-255 = reserved values, will not be returned

Note: "*" means that the instruction was not accepted and no transmission was initiated.

Traps: INSPRIV

INTERRUPTS AND TRAPS — CHAPTER 7

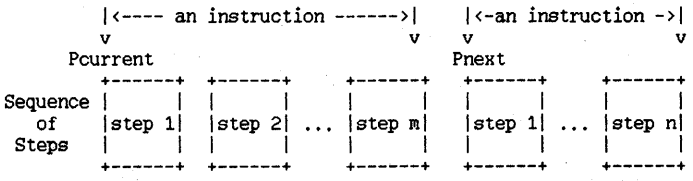
7.1 Introduction

Interrupts and traps are examples of a broad set of conditions called "exceptions" that redirect the normal flow of machine instructions. Generally, instructions are divided into a sequence of smaller actions referred to as "steps". Steps are defined for either of two reasons:

1) The execution of an instruction step results in a change in the machine state (e.g., a byte of memory modified, a register modified). In this case a step may not be repeatable. That is, if the machine state that resulted from a step were used as the input state to the same step, a different output machine state might result.

2) The step represents a large amount of processing. In this case if the instruction were interrupted, too much processing would be lost. Even though the machine state hasn't been modified (as part of the instruction execution so far) it is still desirable to define an intermediate state for the instruction.

Each instruction step is composed of one or more "sub-steps". A sub-step represents an uninterruptible sequence of operations. All steps are architecturally defined. The instruction descriptions define the intermediate state of all instructions that have multiple steps. No other steps or intermediate states are allowed. Instructions which execute very quickly (short in time) have only a single step while other instructions, such as MOVEC (move character), are composed of many steps. The following diagram illustrates this concept of an instruction:



There are multiple conditions that can occur during the execution of an instruction that cause the normal flow of control to be altered. In the previous illustration the normal flow of control is to execute the instruction located at Pcurrent and then to execute the instruction at Pnext. The diagram above shows the steps normally executed for these instructions in the absence of an exception. This diagram is independent of the existence of multiple execution paths that are data dependent. However, if an exception condition(s) is detected during the execution of Pcurrent, the next instruction to be executed by the CPU will be the handler for the condition. Some exception conditions (such as page fault) are specified to be transparent to the current instruction execution. For these conditions the current instruction is resumed after the exception handler completes so that the net effect of the exception is as though it did not occur at all. Other exception conditions (such as overflow) arise as a direct consequence of executing the current instructions. For these exceptions the specific exception defines whether Pcurrent or Pnext is the location to resume instruction execution.

Exceptions are classified into three general categories:

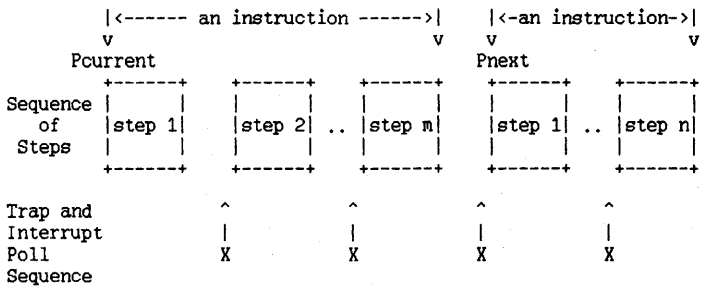
1) external interrupts

2) internal interrupts

3) traps

7.1.1 External Interrupts Overview

External interrupts are generally service requests from I/O devices. External interrupts are polled for between the execution of instruction steps. The following diagram illustrates this:



where X = external interrupt poll

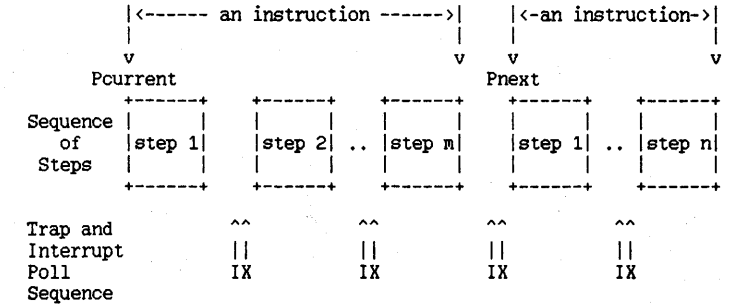
When multiple external interrupts are pending, one is selected (based on the interrupt mask and on priority). The other external interrupts are left pending and are allowed to cause an

external interrupt later (when no longer masked). External interrupts generally are unrelated to the current instruction being executed. The architecture requires that the effect of processing external interrupts be transparent to the current instruction. Therefore, the execution of the current instruction can be suspended between any two steps as long as execution resumes at the next step (technically, execution can resume at any previous step as long as the effects of the intermediate steps can be undone or rolled back). The normal processing sequence for external interrupts is to "cap off" the current stack with an interrupt marker (preserving the current machine context) and to transfer control to the exception handler executing on the interrupt control stack.

7.1.2 Internal Interrupts Overview

Internal interrupts normally originate from some type of abnormal condition occuring within the system not associated with the execution of the current instruction. Some examples of internal interrupts are powerfail, parity error and machine checks. Internal interrupts are polled between the execution of instruction steps. If an internal interrupt is detected, external interrupts are not polled.

The following diagram illustrates this sequence:



where I = internal interrupt poll
X = external interrupt poll

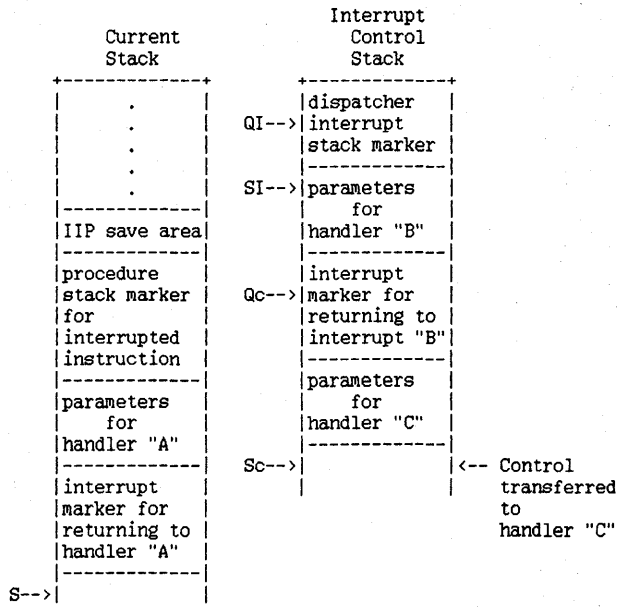
Internal interrupts are handled by pushing a marker (either an interrupt marker or a procedure stack marker depending upon the exception) onto the current stack and then transferring control to the exception handler. The exceptions that push an interrupt stack marker execute on the interrupt control stack (ICS). Those exceptions that push a (procedure) stack marker execute on the current stack.

Multiple internal interrupts are processed by pushing markers onto the stack in increasing priority, then continuing execution with the handler of the latest (highest priority) internal interrupt pushed (interrupts are processed in reverse order -- last-in-first-out). Note, the occurrence of an internal interrupt is remembered by pushing a marker onto the stack so that the handler will execute. This technique contrasts with external interrupts which are remembered with status bits. The following diagram illustrates the stack state(s) following the detection of:

1) an internal interrupt "A" that runs on the current stack,

and

2) two internal interrupts, "B" and "C", respectively, that execute on the ICS.

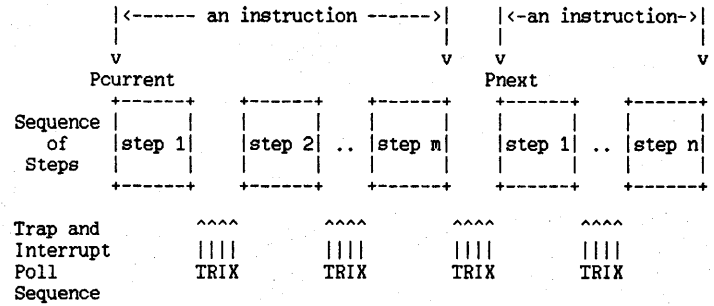


7.1.3 Traps Overview

Traps include all exception conditions which arise as a direct consequence of instruction execution. Examples include traps for arithmetic overflow, ODT access rights violation, page fault and breakpoint (debug). Generally, traps are detected by microcode during the execution of an instruction step. For traps the normal processing sequence is to push an external procedure stack marker onto the current stack, push the parameters on the current stack and then execute the handler on the current stack. An external procedure stack marker is pushed for each different trap detected. Depending upon the type of trap (see the definition of restartable and continuable traps), the stack appears as though an explicit procedure call was made to the trap handler either just before the Pcurrent instruction or after the Pcurrent instruction.

Using the current stack for processing traps allows the maximum degree of concurrency between tasks because the handlers themselves run in a general task environment. This is unlike the interrupt control stack which requires a strict adherence to last-in-first-out processing of events with no option to suspend execution while using the ICS.

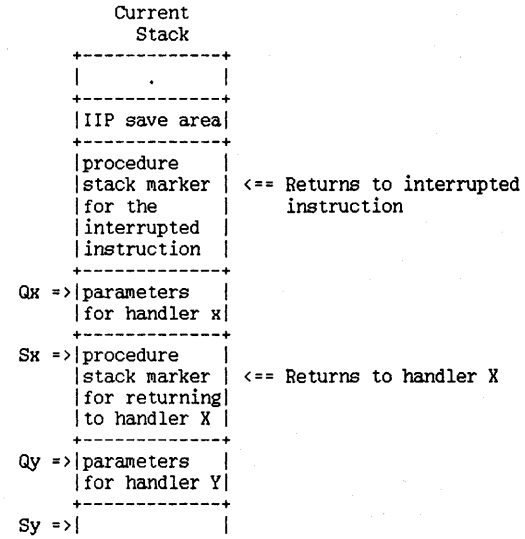
The following diagram illustrates the relationship between the polling of internal and external interrupts and the detection of trap conditions.



where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

The specification of each individual trap condition determines the next instruction step to be executed.

The following diagram illustrates the stack state following the detection of two traps, X and Y, in the same step.



Traps are divided into five categories:

1) non-recoverable traps
2) recoverable traps
   2a) restartable traps
   2b) continuable traps
   2c) step-restartable traps
   2d) step-continuable traps

1) Non-recoverable traps

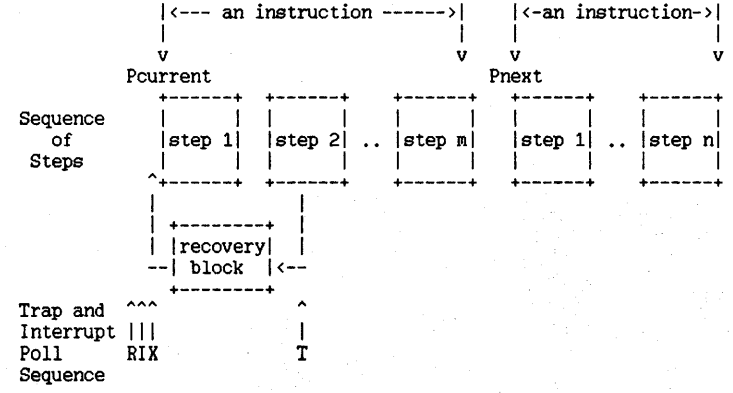
A non-recoverable trap catches the occurrence of a machine state that makes it impossible for the hardware implementation to be able to guarantee correct completion of the instruction even if the trap handler fixes the immediate problem. An example is overflow of the dispatcher disable count or detection of inconsistent Q and S values on IEXIT. Whenever a non-recoverable trap occurs, the simultaneous occurrence of other types of traps is irrelevant.

2) Recoverable Traps

The other categories of traps are part of a set of recoverable exceptions. For these traps it is expected that software can "fix up" the cause of the trap and can re-execute the instruction or software can substitute a "reasonable" result for the instruction.

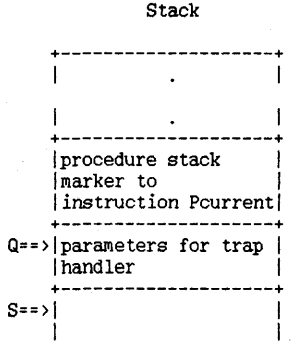
2a) Restartable Traps

A restartable trap is a trap that occurred before the instruction was complete and requires that any changes to the machine state (as part of the current instruction) be undone or "backed out" by the CPU before transferring control to the trap handler. After the trap handler has executed and fixed the problem that caused the trap, the instruction is restarted from the first step. The following diagram illustrates this sequence.



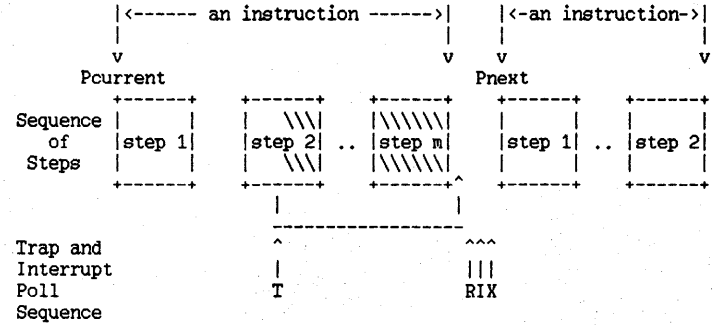
where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

On the previous diagram, a restartable trap could have been detected as part of step 1 through m. On detecting the trap condition, the machine would be restored to its value prior to executing step 1 of the instruction. The program counter value Pcurrent is saved in the stack marker. Then, Pcurrent will be executed again when the trap handler EXITs back to the instruction sequence. At entry to the trap handler the stack state will be:



2b) Continuable Traps

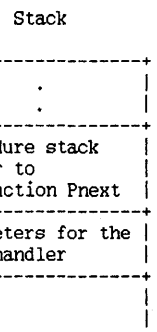
A continuable trap is a trap that serves as an alternate exit point from the instruction. An example is integer overflow. When overflow is detected, the remaining steps in the instruction are skipped. The result of the instruction is the overflow condition. For a continuable trap, the next instruction, after the trap handler, to be executed is Pnext. The following diagram illustrates this:



where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

In the case of a continuable trap, the software trap handler has the option of altering the result of the instruction by modifying (again from software) the result operand. Then execution can be continued from Pnext. This case is very similar to the ordinary external procedure call. The hardware is not required to be able to undo any state changes it has already committed.

At entry to the trap handler, the stack state will be:

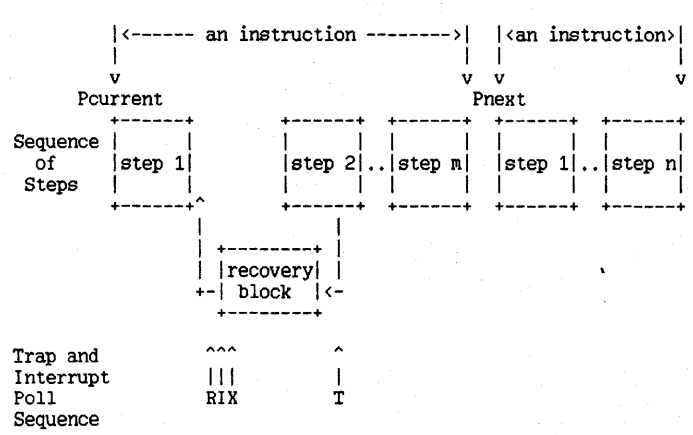


2c and 2d) Step-Restartable and Step-Continuable Traps

The step-restartable and step-continuable traps are very similar to the restartable and continuable traps respectively, but they arise in the context of certain instructions, such as MOVEC, that consist of certain steps repeated a number of times. These instructions have an architecturally defined interrupt state from which they can resume execution safely. A bit in the machine register STATUSA (IIP -- "instruction in progress") allows a decision at instruction fetch time as to whether the instruction has already executed certain steps. If so, the parameters to restart the instruction's execution are popped from the stack and then the instruction is completed. This same mechanism which allows external interrupts to occur in the middle of an instruction also allows internal interrupts and recoverable traps to occur in the middle of the execution of a step without having to back out of more than the last (current) step.

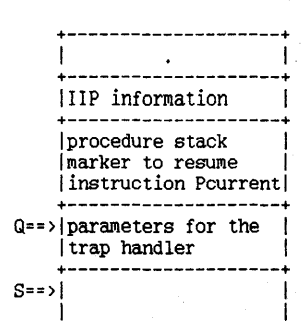
The following diagram illustrates the step-restartable and step-continuable concepts:

Step-Restartable:

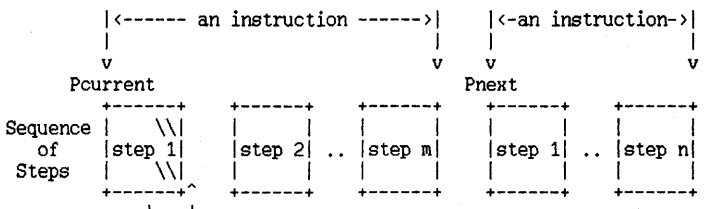


where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

At entry to the trap handler, the stack state will be (except for the top of stack page fault which is handled like an internal interrupt):

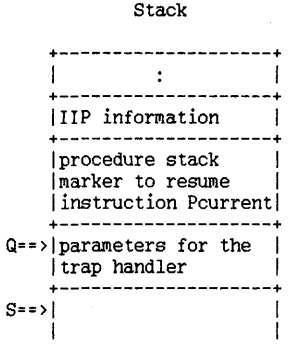


Step-Continuable:



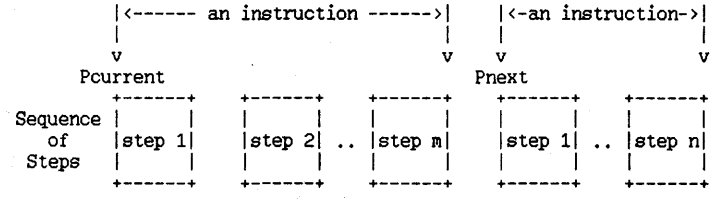
where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

At entry to the trap handler, the stack state will be:



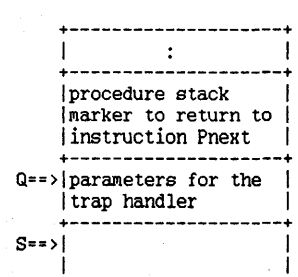
Debug Traps:

The debug traps are a special case of traps. If during the execution of an instruction, microcode detects that the instruction is modifying a location covered by a data breakrange then the DBP (debug breakpoint pending) flag in STATUSA is set to remember that the breakpoint was encountered. Then, the instruction execution is continued with the setting of the breakpoint flag being somewhat transparent. Then, at the end of the instruction execution, the breakpoint handler is activated. By handling the trap at the end of the instruction, the architecture guarantees to report a breakpoint to software only once per instruction. The following diagram illustrates this sequence:



where I = internal interrupt poll
X = external interrupt poll
T = trap condition detected
R = trap condition reported (trap handler activated)
\ = part of the step(s) not executed

At entry to the debug traps, the stack state will be:



7.1.3.1 Special Programming Notes

STATUSB Handling

When control is transferred to exception handlers that execute on the current stack, an external procedure call marker is pushed onto the stack. This marker includes the STATUSA register, but not the STATUSB register. In order not to have any side effects on the suspended instruction, software must save STATUSB at entry to the handler. Then, immediately prior to EXITing back to the suspended instruction, the handler should restore STATUSB to its value when the trap was detected.

Hardware versus Software Recoverability

The classification of the recoverability of a trap condition is based upon whether or not software can remove the condition that caused the trap and can then allow the hardware to proceed with instruction execution. Instruction execution can proceed from either the instruction that caused the trap or the instruction following. This criterion for classification is more permissive than one based strictly on whether machine state has been modified. As an example, if a MOVE8 from location A to location B were to get a bounds violation on B, part of B might have been modified before the bounds violation was detected. If the trap handler increases the upper bound of the object containing B such that the MOVE8 no longer causes a bounds violation, the net effect is that the trap did not occur. This example illustrates a case where the hardware could not restore the contents of location B but software could fix the problem. So from a hardware perspective the input state of the instruction cannot be recreated. From a software perspective the instruction is restartable. Using the classification criterion, this bounds violation is recoverable.

7.2 Detail Description of External Interrupts

7.2.1 Processor context for interrupts

Three items of processor context define the interaction between processor and interrupts. There is an Interrupt Enable/Disable bit in STATUSC, called STATUSC.IE or "IE" for short. This bit controls whether any interrupts are allowed to cause a change in the sequence of execution of macro-instructions. There is a 16-bit Interrupt Mask Register in STATUSC, called STATUSC.IMR or "IMR" for short. This mask controls which interrupts are allowed to cause a change in the sequence of execution and which are not, subject to STATUSC.IE. Finally, there is an Interrupt Pending Register, which is not directly accessible to software; hence implementations have a large amount of freedom in how to implement it. It need not even exist as a separate entity in the machine, as long as the behavior that is here ascribed to it can be reproduced. The Interrupt Pending Register looks like an array as in Pascal:

```
TYPE
   pr_level: 0..15;
   source:    (i_channel, i_processor);
   state:     (clear, set);
VAR
   IPR: ARRAY[pr_level, source] OF state;
```

The IPR is processor-local. Details on multiprocessor aspects of the interrupt system follow in section 7.2.7.

7.2.2 General operation

Interrupts cause bits to get set in IPR (elements of IPR to become 1). The state of the Interrupt Mask Register and the state of the Interrupt Enable/Disable bit control whether the processor is notified or wether the interrupt is held off. Interrupts can be caused by channels or by a processor itself under software control.

7.2.3 Channel Interrupts

Each hardware channel is configured at a specific priority

```
pr : pr_level;
```

If the channel wants to raise an interrupt, it does so by setting the appropriate bit in IPR:

```
IPR[ pr, i_channel ] :=set;
```

If STATUSC.IE = 1 and IMR[pr] = 1, the processor is interrupted at the first convenient opportunity (e.g. between instructions), otherwise the interrupt is held off. The channel must be prepared to inform the processor of details concerning the interrupt when the interrupt is acknowledged. To this end, the channel may use an area of the processor's memory (called "channel overflow area") to store any information needed to avoid overflowing its internal memory capacity. Use of such an interrupt queueing mechanism is an optional hardware implementation and not required by the architecture. Some channels may have restrictions that guarantee that no more than a single interrupt may be outstanding, or the channel may have enough internal buffering so that processor's memory is not needed. Any such use of the processor's memory is transparent except perhaps for initialization of the channel overflow area at configuration time.

7.2.4 Processor-caused Interrupts

The processor raises an interrupt by setting a bit in the IPR through the "INTERRUPT" instruction. The processor will typically hold off this interrupt; when the processor later acknowledges the interrupt, hardware does not report to software any information regarding the interrupt other than the priority level at which the interrupt occurred. Software is responsible for any queueing that is required to entangle the course of events in case of multiple software interrupts. Such queueing must be done before executing the INTERRUPT instruction.
"INTERRUPT pr" sets the appropriate bit in IPR:

```
IPR[ pr, i_processor ] := set;
```


7.2.5 When is the Processor Interrupted?

At the end of an instruction, or at appropriate places in the middle of a long instruction, the processor checks to see if interrupts should be acknowledged. Interrupts are to be acknowledged if the IPR is left "set" at a priority level pr that is currently enabled. The algorithm can be sketched in Pascal as follows:

```
if STATUSC.IE = 1 then
   for pr := 0 to pr_levels-1 do
      if STATUSC.IMR[pr] = 1 then
         for src := i_channel to i_processor do
            if IPR[ pr,src ] = set  then
            begin
               IPR[ pr,src ] := clear;
               GO_ACKNOWLEDGE_INTERRUPT(pr,src);
               end
```

7.2.6 Acknowledging Interrupts

Section 7.2.5 sketched the algorithm that defines which interrupt, if any, must be acknowledged. The algorithm ends either in a GO_ACKNOWLEDGE_INTERRUPT(pr,src) or it indicates that the flow of control should not be changed at all. Detailed below are the steps that must be taken when acknowledging the interrupt.
Note that software arrives at the interrupt handler with values in the registers X0..X15 and B0..B5 that are indeterminate.

```
GO_ACKNOWLEDGE_INTERRUPT(pr,src) :
begin
   STATUSC.IE := 0;
   PUSH_INTERRUPT_MARKER;
   if STATUSC.ICS = 0 then  {a task was interrupted}
   begin
      save S in TCB;
      Q := QI;  S := Q;
      STATUSC.ICS := 1;
      end;
   STATUSA.XL := 0; {go to privileged mode}
   if src = i_channel then
      push channel dependent information identifying
      the interrupt;
   PUSH4  pr;
   {all registers X0..X15, B0..B5 will be indeterminate}
   if src = i_channel  then BRX  2  else  BRX  3
   end;
```

7.2.7 Shared-memory Multiprocessor Considerations

The interrupt mask register is processor-local, as is the interrupt enable/disable bit. More surprising, perhaps, is the fact that the interrupt pending register is processor-local. For this to work, the following notes apply.

Note 1: A channel interrupt causes the pending bit to get set in the IPR of all processors sharing memory.

Note 2: The INTERRUPT instruction must likewise broadcast to all processors in order to get the pending bit set in their own IPR.

Note 3: More than one processor may have a particular priority level enabled at any one time. In this situation, more than one processor will be interrupted. In case of a channel interrupt, the data structure that identifies the interrupt is shared among all processors; this allows only one processor to acknowledge the channel interrupt, all other processors will resume their normal instruction sequence without ever pushing an interrupt marker. In case of a processor interrupt, all enabled processors will run the interrupt handler.

Note 4: When a processor acknowledges an interrupt, it clears the pending bit in its own IPR only. This will not be broadcast.

7.3 Clocks

There are three clocks supported. Each clock is scaled to give results in nanoseconds. However, the actual resolution of the clock is implementation dependent and may be much larger than 1 nanosecond. For example, internally the hardware may count every 100 nanoseconds but when software reads the clock, the count will be scaled to read in nanoseconds. These clocks, when read, return a 64-bit 2's complement count.

7.3.1 Time of Day Clock

This clock will be used by the system for maintaining the current time of day. It runs continuously without interruption and will maintain the correct time even across power failure. Ideally, this clock will be set only once and from that point onwards it will continually count up.

Since 1 January 1972 there has been an internationally accepted time scale based on the International Time Bureau (BIH) standards for atomic clocks. The Vision Time of Day Clock will be based on this standard.

The origin (time zero) of this clock is the same as the origin for the international reference scale of atomic time (TAI), that is, 1 January 1958 at 0 hours GMT (also known as the UTC Reference Zone). The value of this clock is the number of nanoseconds since the TAI origin as defined by Coordinated Universal Time (UTC). Thus, as an example, if it is 4 AM PST then it is 12 Noon UTC as there is an eight hour time difference between California and Greenwich. For details of this time standard see NBS Special Publication 559, "Time and Frequency Users' Manual". Not that it is not intended that all Vision computers be as accurate as atomic clocks but merely that they agree on what time it is.

The following functions are provided to support this clock.

- SET CLOCK (value passed is 64 bits)
- READ CLOCK (return value is 64 bits)

7.3.2 Task Clock

This clock will be used by the system for accounting purposes. This clock counts up and is put in the hold mode whenever control is transferred to the Interrupt Control Stack (ICS). It may also be disabled by software by placing it in hold mode. On return from the ICS, it resumes counting.

The following functions are provided to support this clock.

- SET CLOCK (value passed is 64 bits)
- READ CLOCK (return value is 64 bits)
- HOLD CLOCK
- RESUME CLOCK

7.3.3 Interval Clock

This 64-bit two's-complement clock interrupts the CPU after a programmable interval has elapsed. It is used by the system for device time-outs, time slicing of processes, etc. The interrupt is treated like any other I/O interrupt in the system and is therefore subject to being masked off by software. The clock is set by loading it with the desired interval, in nanoseconds. (It should be a positive interval. A negative interval will load zero into the clock and cause an immediate interrupt.) From there on, it counts down until it becomes negative at which time the interrupt is generated. The interrupt is signalled to all processors in a shared-memory multiprocessor system at a priority level that can be configured by software.

On power-up, the interval clock shall be set to its largest positive value. This should prevent any unexpected interrupts from being generated by this clock for at least 292 years.

The following functions are provided to support this clock.

- SET CLOCK (value passed is 64 bits)
- READ CLOCK (return value is 64 bits)

7.4 Summary of Traps and Internal Interrupts

The following table is a summary of the internal interrupts and traps. Detailed descriptions of each internal interrupt and follows in this chapter. For this table, the following notation will be used:

1. ENV -- Execution environment of handler
   a) CS = current stack
   b) ICS = interrupt control stack
2. E/D Control -- Enable/Disable Control. This indicates the control that software has over the transfer of control to the handler.
   a) PE = permanently enabled
   b) The status word and flag(s) that control the handler (eg. B2.INTOVFE for fixed point overflow)
3. Parameters -- Parameters passed to handler
   a) Pcurrent=Pc = logical address of offending instruction.
   b) Pnext=Pn = logical address of the instruction following the offending instruction.
   c) Preturn=Pr = the return address in the stack marker.
4. Type -- Type of exception
   a) II = Internal interrupt
   b) NR = non-recoverable trap
   c) R = restartable trap (Pr=Pc)
   d) C = continuable trap (Pr=Pn)
   e) SR = step restartable (Pr=Pc)
   f) SC = step continuable (Pr=Pn)

Table of Internal Interrupts and Traps

Mnemonic	Trap#	ENV	E/D Control	Type	Parameters
IIMEMPAR	1	ICS	PE	II	Address,Pc,1
POWERFAIL	2	ICS	PE	II	2
IIPWRRCV	3	ICS	PE	II	3
IICPUCHK	4	ICS	PE	II	Variable,Pc,4
IICSPREPLY	5	ICS	C1.IE	II	Status,5
CODEBNDSV	6	CS	PE	R	Pc,6
CODEODIV	7	CS	PE	R	Pc,7
CODETYPV	8	CS	PE	R	Pc,8
CODERINGV	9	CS	PE	R	Pc,9
INSPRIV	10	CS	PE	R	Pc,10
INSOPSPEC	11	CS	PE	R	Pc,11
INSERROR	12	CS	PE	C	Pc,12
INSCHKLO	13	CS	PE	C	Pc,13
INSCHKHI	14	CS	PE	C	Pc,14
INSUNDEF	15	CS	PE	R	Pc,15
INSXTL	16	CS	PE	C	Pc,16
INSODDP	17	CS	PE	NR	Pc,17
INSPROBE	18	CS	PE	R	Pc,18
INSMOVSPL	19	CS	PE	R	Pc,19
INSSWITCH	20	CS	PE	R	Pc,20
INSVPPERM	21	CS	B1.VPP	R	Pc,21
INSVPICS	22	CS	PE	NR	Pc,22
STKCONSISTV	23	CS	PE	R	Pc,23
STKOVF	24	ICS	PE	R	Pc,24
STKUNF	25	CS	PE	R	Pc,25
STKDEXTV	26	CS	PE	R	Pc,26
DATABNDSV	27	CS	PE	R	Pc,27
DATAODIV	28	CS	PE	NR	Address,Pc,28
DATATYPV	29	CS	PE	R	Address,Pc,29
DATAARV	30	CS	PE	R	Address,Pc,30
FL-INV	31	CS	B2.FLINVE	C	Op1,[Op2],Pc,31
FL-DVDZ	32	CS	B2.FLDVDZE	C	Op1,Op2,Pc,32
FL-OVF	33	CS	B2.FLOVFE	C	Result,Status,Pc,33
FL-UNF	34	CS	B2.FLUNFE	C	Result,Status,Pc,34
FL-INX	35	CS	B2.FLINXE	C	Result,Status,Pc,35
INTDVDZ	36	CS	B2.INTDVDZE	C	Pc,36
INTOVF	37	CS	B2.INTOVFE	C	Pc,37

Mnemonic	Trap#	ENV	E/D Control	Type	Parameters
DECDVDZ	38	CS	B2.DECDVDZE	C	Pc,38
DECOVF	39	CS	B2.DECOVFE	C	Pc,39
DECINVL	40	CS	PE	R	Pc,40
DECINVDG	41	CS	PE	R	Pc,41
DBBREAK	42	CS	PE	C	Operand,Pc,42
DBCALL	43	CS	B1.PTE	SC	Pc,43
DBCHECKA	44	CS	B2.CB.CBA	C	Operand,Pc,44
DBCHECKB	45	CS	B2.CB.CBB	C	Operand,Pc,45
DBSIT	46	CS	STATUSA.SIT	C	Pc,46
SEMAOVF	47	CS	PE	R	Semaphore,Pc,47
SEMADOWN	48	CS	PE	C	Semaphore,Pc,48
SEMAUP	49	CS	PE	C	Semaphore,Pc,49
SWITCHN (*1)	50	CS	PE	C	Pc,50
TRYV	51	CS	PE	R	Trypointer,Pc,51
ADRPDIRBND	52	CS	PE	R	Entry address,Pc,52
ADRPDIR	53	CS	PE	R	Page number,Pc,53
ADRPAGEABS	54	CS	PE	R	Byte offset,VPN, Logical address, Pc,54
ADRPAGETOS	55	ICS	PE	R	User stack data, Byte offset, VPN, Logical address, Pc,55

Notes:
*1: This handler runs on the current stack, but it switches from the compatibility mode part of the stack to the native mode part.

7.5 Detail Description of Internal Interrupts

7.5.1 Architectural Interface

When an internal interrupt is detected, control is transferred to the corresponding internal interrupt service routine. The methods of transferring control and accessing the interrupt service routines are consistent (identical) across all models of the Vision family. The following sections describe the details of the architectural interface between hardware and software (the interrupt service routine).

7.5.2 Execution Environment

All internal interrupt handlers execute on the ICS.

7.5.3 Sequence of Events

When an internal interrupt is detected, hardware performs the following sequence:

1) External interrups are disabled.

2) In case the currently executing instruction is interrupted in an intermediate state, intermediate state information is pushed onto the stack and the IIP bit in STATUSA is set. Otherwise, the step is skipped. (See the description of individual instructions for details on interruptible steps).

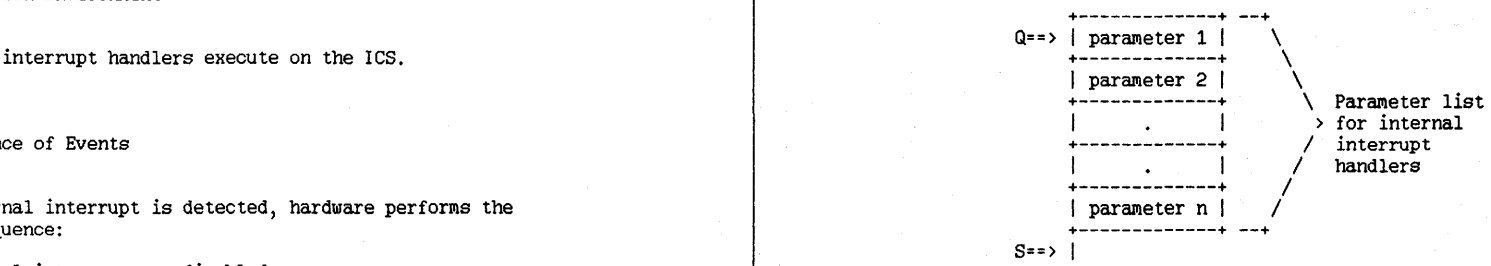
3) The current execution stack is capped with an interrupt stack marker.

4) The following status fields are given standard values:

```
XM    = 0   (native mode)
IIP   = 0
TCE   = 0   (hold task clock)
DISP  = 0
SIT   = 0
DBP   = 0
ICS   = 1
XTL   = 3
```

5) The target location is the entry point in the OD for LOI = 1.

6) A parameter list is pushed onto the interrupt control stack. The description of each internal interrupt describes the parameters. The parameters are pushed onto the stack as shown in the following diagram:



In all cases the last parameter is the 32-bit Trap#.

7) The execution environment is set up for executing the internal interrupt handler. This involves the following:

   a) The environment registers (Q, S, etc.) are set up appropriately for executing code on the ICS. All other registers X0..X15, B0..B5 are left with indeterminate values.

   b) The execution privilege level is set to the minimum execution level described in the OD corresponding to LOI = 1.

   c) A branch (BRX) is performed to the destination defined by the logical object id = 1.

8) Steps 3 through 7 are repeated for each internal interrupt detected.

7.5.4 Multiple Internal Interrupts

Multiple internal interrupts are processed by pushing multiple interrupt stack markers onto the ICS. Interrupt markers are pushed in increasing order of priority. Then execution continues by transferring control to the handler for the latest (highest priority) internal interrupt.

Internal interrupts in order of increasing priority are:

1) any internal interrupt except power fail and power recovery.

2) power recovery

3) power fail

7.5.5 Internal Interrupts Descriptions

7.5.5.1 Memory Parity Error

This internal interrupt is caused when hardware detects a "hard" memory error that it cannot resolve without involving software.

The physical address involved in the access that incurred the error is pushed onto the ICS.

In a shared memory multiprocessor configuration, the parity error may not be uniquely attributable to any particular processor's memory traffic. Therefore, the parity error may be reported to any processor in the configuration.

Mnemonic: IIMEMPAR
Parameters: 1. 32-bit physical byte address of the location with the parity error
2. Pcurrent
3. trap #
Enabling: permanently enabled

7.4.5.2 Power Fail

When the power system detects a power failure, the power fail interrupt is taken. In a shared-memory multi-processor configuration, all processors must receive this interrupt.

Mnemonic: IIPWRFAIL
Parameter: trap #
Enabling: permanently enabled

7.5.5.3 Power Recovery

When power is initially applied to the system, a test of memory contents is performed to determine if it contains valid information and data. If so, the hardware is initialized (including writeable control store) and the power recovery interrupt is taken (warm start). If memory contents are invalid, the machine will perform a cold start. The test for valid memory contents is implementation dependent but there will be a finite probability of mistaking an invalid memory content as being valid. In a shared memory multiprocessor configuration, all processors must receive this interrupt. As much as possible, implementations must save the machine state across power fail/recovery.

Mnemonic: IIPWRRCV
Parameter: trap #
Enabling: permanently enabled

7.5.5.4 CPU Machine Check

This trap is defined for the implementation dependent errors that a CPU implementation can detect about itself. The information reported under this trap classification is specific to each CPU implementation. The first parameter is variable in size. Its third word is the machine check ID number; this defines how much additional information is present.

Mnemonic: IICPUCHK
Parameter: 1. machine check id
2. Pcurrent
3. trap #
Enabling: permanently enabled

7.5.5.5 CSP reply is complete

When hardware has completed receiving the reply from the CSP to the message sent through the MOVEtCSP instruction, this internal interrupt is generated.

Mnemonic: IICSPREPLY
Parameters: 1. 32-bit status (implementation dependent)
2. trap#
Enabling: individually enabled (STATUSC.IE)

7.6 Detail Description of Traps

7.6.1 Architectural Interface

When a trap is detected by the hardware, control is transferred to its corresponding trap service routine. The method of transferring control and accessing the trap service routines is consistent (identical) across all models of the Vision family.

Traps provided are also consistent across all models of the Vision family.

The following sections describe the details of the architectural interface between the hardware (or microcode) and the software (the trap service routines).

7.6.2 Execution Environment

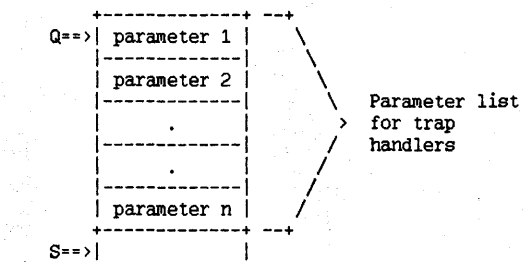
All trap handlers execute on the current stack except the top of stack page fault handler (ADRPAGETOS) which executes on the ICS.

7.6.3 Common Conventions for Traps

7.6.3.1 Parameter Passing to Trap Handlers

All traps push their parameters after pushing the procedure stack

markers. The diagram below shows how parameters are pushed:



In all cases the last two parameters are the 64-bit program counter Pcurrent, and the 32-bit Trap #.

7.6.3.2 Determining Privilege of the Handler

The trap handler is a procedure in the Trap object (object 1 in group 0). The access rights of the trap object indicate the nominal privilege level at which the handler will run. However, privilege is never reduced (will never become numerically greater) in going to a trap handler. This corresponds to the normal procedure calling conventions.

7.6.3.3 Determining the address of a Trap Handler

The address of the trap handler is defined by code object 1 in group 0.

7.6.4 Sequence of Events

The following sections describe the sequence of events involved in transferring control to the trap handler. The descriptions rely on the conventions set out in the previous section. External and internal interrupts are held off during these sequences.

7.6.4.1 A Non-recoverable Trap on the Current Stack

When a non-recoverable trap is detected whose trap handler executes on the current stack, the following events take place.

1) The SIT bit in STATUSA is cleared.

2) An external procedure stack marker is pushed on the stack.

3) The following status fields are given standard values:
   XM = 0 (native mode)
   DBP = 0

4) Q and S are set as expected on a procedure call.

5) Parameters to the trap handler are pushed on the stack.

6) P is set to the entry point address of the trap handler identified by the trap code object.

7) STATUSA.XL is set to the privilege level at which the handler should run.

8) Control is passed to the trap handler at P.

7.6.4.2 A Non-recoverable Trap on the ICS

When a non-recoverable trap is detected whose handler executes on the ICS, the sequence of events is identical to that for an internal interrupt.

7.6.4.3 One Restartable Trap on the Current Stack

(or a step-restartable trap)

When a single restartable trap is detected whose handler runs on the current stack, the sequence of events is the following:

1) For an interrupted instruction, the intermediate state information is pushed, and the IIP bit in STATUSA set. Otherwise, this step is skipped.

2) A stack marker is pushed onto the stack.

3) The following status fields are given standard values:
   IIP = 0
   SIT = 0
   XM = 0 (native mode)

4) Q and S are set as expected on a procedure call.

5) Parameters for the trap handler are pushed onto the stack.

6) P is set to the entry point address of the trap handler identified by trap code object.

7) STATUSA.XL is set to the privilege level at which the handler should execute.

8) Control is passed to the trap handler at P.

7.6.4.4 One Restartable Trap on the ICS

(or a step-restartable trap)

This follows the sequence for an internal interrupt; except the P value reported corresponds to the current instruction, not the next.

7.6.4.5 Top-of-stack Page Fault and Stack Overflow

These follow the sequence for an internal interrupt. Note that these faults can occur at any time when pushing stack markers and parameters for trap handlers. A description of the sequence of events in this case is given in section 7.7.

7.6.4.6 Multiple Restartable Traps

(or step-restartable traps)

When more than one restartable trap is detected, hardware selects one and ignores the others. The sequence followed is therefore given by one of the sections above.

7.6.4.7 Continuable traps

(or step-continuable traps)

Continuable traps can only be detected after restartable traps have already been resolved, so continuable traps occur either alone or in combination with other continuable traps. Note that the breakpoint trap and the single instruction trace trap are classified as continuable traps. In addition to these two, only one continuable trap can occur in an instruction.

Sequence of Events:

1) Remember the state of the SIT bit in STATUSA.

2) Clear the SIT bit in STATUSA (SIT=0).

3) If no other continuable traps except SIT or breakpoint then go to step 11.

4) If the instruction is step continuable and was interrupted at an intermediate step, push intermediate state information onto the current stack and set the IIP bit.

5) Push an external procedure marker.

6) Clear the IIP flag in STATUSA.

7) Set Q and S as expected for a procedure call.

8) Push parameters for the one continuable trap other than SIT or breakpoint.

9) Set P to the entry point address of object 1 in group 0.

10) Set STATUSA.XL to the privilege level of object 1 in group 0.

11) If the DBP bit is clear, go to step (18).

12) Push an external procedure marker.

13) Set Q and S as expected in a procedure call.

14) Push parameters for the breakpoint table trap onto the stack.

15) Set P to the entry point address of object 1 in group 0.

16) Set STATUSA.XL to the privilege level of object 1 in group 0.

17) Reset the DBP bit.

18) If the SIT bit was found in step (1) above, go to step (24).

19) Push an external procedure marker.

20) Set Q and S as expected in a procedure call.

21) Push parameters for the Single Instruction Trace trap.

22) Set P to the entry point address of object 1 in group 0.

23) Set STATUSA.XL at the privilege level of object 1 in group 0.

24) Execute the trap handler at P.

Note: this sequence may give a DBP trap and an SIT trap before a step continuable instruction will have fully completed. These trap handlers can either choose to run at this time or they can set the SIT bit in the stack marker for the interrupted instruction so that the handlers can release control and yet regain control back at the end of the instruction.

7.6.5 System Error

Certain error conditions are non-recoverable and they cause the processor to enter in a special system error state. The following conditions cause the processor to enter the 'system error' state.

1) Any trap, such as ODT Length violation, that occurs while hardware executes the transfer of control to the trap handler.

2) Cases like overflow or underflow of the dispatcher disable count. In these cases, there is a software error in privileged code.

3) Bounds violations on the ICS.

4) TOS page faults when executing on the ICS.

7.6.6 Enabling/Disabling Traps

Traps may be explicitly enabled and disabled individually or in groups. Traps fall in the following categories.

1) Permanently Enabled

   These traps are always enabled when the system is up and the software is running. These traps cannot be explicitly disabled.

2) Individually Enabled

   These traps can be explicitly enabled/disabled individually by setting/resetting a bit in the STATUSB register. Setting of the bit (=1) enables the trap. Resetting the bit (=0) disables the trap.

7.6.7 Transfer of Control Traps

For the descriptions of the transfer of control traps, these notes are applicable:

1) The Lower Bound (LB) of the object is obtained from the OD for the object (third word).

2) The Upper Bound (UB) of the object is obtained from the OD for the object (fourth word).

3) The Object Length is computed from the OD for the object:

   Object Length = UB - LB + 1

4) LB and UB are 32-bit 2's complement signed integers; their values, however, must be positive.

5) The Virtual Address of the target location is calculated according to the description in chapter 3. Generally, the Virtual Offset is computed from the logical offset by the calculation: VOFF = LB + LOFF

6) For instructions BR and CALL, the target code object is always the executing code object because these instructions can only cause internal transfers.

7) For instructions BRX, CALLX and EXIT, the target code object may be either the executing code object of a different code object because these instructions allow both internal and external transfers.

8) For BRX and CALLX, the target location is obtained from the object descriptor of the target code object.

9) For EXIT, the target location is obtained from the procedure stack marker.

7.6.7.1 Code Object Bounds Violation

This trap is caused when P points outside the bounds of the code object. For instructions that change flow of control, such as BR, CALL, CALLX, EXIT, this trap is detected before the next instruction is fetched, so that the Pcurrent of the offending instruction can be reported to software.

Implementations need not detect a Code Object Bounds Violation on sequential instruction execution (instructions other than BR, BRX, CALL, CALLX, SEXIT, EXIT, IEXIT). It is the responsibility of operating system software to guarantee that software cannot "run out of theend of a code object". For example, code objects can be padded with BREAK instructions. If P is incremented so as to become greater than PL on sequential instruction execution, the effects may differ across implementations; however, these effects will remain limited to the currently executing task.

Mnemonic: CODEBNDSV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.7.2 Code ODT Length Violation

This trap is detected for the instructions BRX, CALLX, and EXIT. It occurs when an attempt is made to transfer control to an object that does not exist; i.e., the object number is greater than the number of entries in the ODT of the group selected by the group selector in the target logical address.

Mnemonic: CODEODTV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.7.3 Code Object Type Violation

This trap is detected for instructions BRX, CALLX, EXIT and IEXIT when an attempt is made to transfer control to an object that is not a code object.

Mnemonic: CODETYPV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.7.4 Code Privilege Level (Ring) Violation

This trap is caused for the following cases:

1) This trap is caused in the EXIT instruction when an attempt is made to exit to a privilege level which is more privileged than the processor's current privilege level (contained in the STATUSB register).

2) This trap is detected for BRX and CALLX when an attempt is made to transfer control to a target code object whose `prerequisite privilege level' is more privileged than the current privilege level. The `prerequisite privilege level' of a procedure entry point is contained in the OD of the target object describing the procedure entry point.

Mnemonic: CODERINGV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8 Instruction Traps

7.6.8.1 Privileged Instruction Violation

This trap is caused when an attempt is made to execute an instruction at a privilege level which is less privileged than that required by the instruction.

Mnemonic: INSPRIV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8.2 Error Instruction

This trap is caused by executing the ERROR instruction. This is likely to occur when an error causes P to point to data instead of code, i.e., trying to execute data.

Mnemonic: INSERROR
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.8.3 CHECKLO Violation

This trap is caused when, for the instruction CHECKLO, the first operand is smaller than the second operand.

Mnemonic: INSCHKLO
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.8.4 CHECKHI Violation

This trap is caused when, for the instruction CHECKHI, the first operand is larger than the second operand.

Mnemonic: INSCHKHI
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.8.5 Undefined Instruction

This trap is caused for all opcodes that are not defined as part of the VISION architecture.

Mnemonic: INSUNDEF
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.8.6 Exit Threshold Trap

This trap is caused when the current execution privilege level is reduced to a level that is less privileged than the level in the 'XTL' field in STATUSB.

Mnemonic: INSXTL
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.8.7 Misaligned Program Counter

This trap occurs when the return address in EXIT, SEXIT or IEXIT is not even, so that P would not be on a half-word boundary.

NOTE: This condition may not cause a trap in all implementations; instead, implementations may force P[63]:=0 and continue.

Mnemonic: INSODDP
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8.8 Probe Violation

This trap is caused for instruction PROBE, when the value of the first operand and/or that of the second operand is/are illegal.

Mnemonic: INSPROBE
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8.9 Operand Specifier Violation

This trap is caused when an operand specifier in an instruction is incompatible with the operand attribute expected by the opcode. Example: an operand specifier specifying a literal as a destination in a MOVE instruction.

Mnemonic: INSOPSPEC
Parameters: 1. Pcurrent
2. Trap #
Trap Type: non-recoverable
Enabling: permanently enabled

7.6.8.10 Move Special Violation

This trap is caused for the instructions, MOVEfSP4, MOVEfSP8, MOVEtSP4, and MOVEtSP8, when the value of the selector is illegal and/or when the current privilege level of the processor does not match the required privilege level for that value of the selector.

Mnemonic: INSMOVSPL
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8.11 Switch Violation

This trap is detected by all variants of SWITCH when the execution environment does not allow a task switch.

Mnemonic: INSSWITCH
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.8.12 VP permission control

This trap is detected by all vector instructions when a vector operation is decoded and the vector permission bits (STATUSB.VPP) are zero. That is, the current status does not allow access to the vector instructions because the software environment (vector context save area) has not been initialized.

Mnemonic: INSVPPERM
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: individually enabled
(STATUSB1.VPP)

7.6.8.13 Vector Operation on the ICS

This trap occurs when a vector operation is attempted that uses vector registers while executing on the ICS.

Mnemonic: INSVPICS
Parameters: 1. Pcurrent
2. trap #
Trap Type: non-recoverable
Enabling: permanently enabled

7.6.9 Stack Traps

7.6.9.1 Stack Consistency Violation

The instruction EXIT is used to restore the caller's environment. The registers S, Q are changed to point to new memory locations on executing the EXIT instruction. Prior to executing the EXIT instruction, checks are made to ensure that the registers SB, Q, S, and SL at the end of executing the EXIT instruction would still maintain the following stack consistency relationship:

SB =< Q =< S =< SL

The Stack Consistency Violation trap is taken when this relationship is violated. The IEXIT instruction includes the same checks.

Mnemonic: STKCONSISTV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.9.2 Stack Overflow

This trap is caused when attempting to execute an instruction that will result in S pointing at or beyond SL. Note: processing of the trap condition follows the sequence of events for internal interrupts. The trap handler is executed on the ICS. When this exception is detected, S is set according to the following rules:

1) If the offending instruction is 'restartable' (see below), the is restored to its value prior to the offending instruction.

2) If the offending instruction is one for which this trap is 'step restartable', S is restored to its value prior to the offending instruction step.

In either of the above cases, S is rolled back to the appropriate position so that the offending instruction can be appropriately 'restarted' or 'step restarted'. Then the interrupt marker is pushed onto the stack according to the rules given in section 7.xx (TOS page faults). The overflow part of the marker will go on the ICS. At the end of this sequence, S[32..63] in the TCB will point to where it would have pointed had the entire interrupt marker been pushed onto the user's stack.

Mnemonic: STKOVF
Parameters: 1. Pcurrent
2. trap #
Trap Types: step restartable for instruction DUP
restartable for other offending instructions
Enabling: permanently enabled

7.6.9.3 Stack Underflow

This trap is caused when an attempt is made to move S below Q (i.e. attempt to violate Q[32..63] <= S[32..63])

Mnemonic: STKUNF
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.9.4 Delete/Extend Negative Wordcount

The trap is caused when, for instructions DELETE and EXTEND, the wordcount given is negative.

Mnemonic: STKDEXTV
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.10 Data Object Traps

For data traps the following terminology is used:

1) An explict operand is an operand whose logical address is specified by an operand specifier of the instruction.

2) A data access is non-explicit when its logical address is not directly specified by an operand specifier. The logical address of a non-explicit operand is either specified indirectly by an explict operand (as in VGATHt, VSCATt) or is obtained obtained by modifying/indexing the logical address of an explicit operand (as in MOVEC).

3) The Virtual Address of a byte of any operand is computed according to the algorithms in chapter 3.

7.6.10.1 Data Object Bounds Violation

This trap is caused when the computed (effective) virtual offset for an operand (explicit or implicit) is less than the Lower Bound LB in the OD for the object or the computed virtual offset is greater than the Upper Bound UB minus the size of the data item.

Mnemonic: DATABNDSV
Parameters: 1. Offending logical address (64 bits)
2. Pcurrent
3. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.10.2 Data ODT Length Violation

This trap is caused when a data access uses a logical address with with an object number greater than the number of entries contained in the ODT for the selected group.

Mnemonic: DATAODTV
Parameters: 1. Offending logical address (64 bits)
2. Pcurrent
3. trap #
Trap Types: non-recoverable for instructions that modify the most significant 32 bits of a base register,
restartable for IEXIT
Enabling: permanently enabled

7.6.10.3 Data Object Type Violation

This trap is caused when an attempt is made to access, through a ".w" or ".rw" attribute, an object that is not a native mode data object.

Mnemonic: DATATYPV
Parameters: 1. Offending logical address
2. Pcurrent
3. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.10.4 Data Access Rights Violation

This trap is detected when an attempt is made to access an object while running less privileged than required by the access rights field in the OD for the object.

Mnemonic: DATAARV
Parameters: 1. Offending logical address (64 bits)
2. Pcurrent
3. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.11 Floating Point Traps

These traps are detected for Floating Point operations. Their implementation is in accordance with IEEE Floating Point Standard. (Refer to "A Proposed Standard for Binary Floating Point Arithmetic" draft 9.3.3 of IEEE task P754.) Each trap can be individually enabled or disabled with the appropriate bit in STATUSB. When the trap condition is detected, the destination operand is set according to the following rules:

1) If the trap is enabled, then the contents of the destination operand are not changed (i.e., remain the same as prior to executing the offending instruction).

2) If the trap is disabled, then the contents of the destination operand are set as specified in the IEEE standards.

7.6.11.1 Floating Point Invalid Operation

This trap is caused for Floating Point Invalid Operations as defined in IEEE Floating Point Standard. The operand(s) of the offending instruction is (are) pushed.

Mnemonic: FL-INV
Parameters: 1. Operand1
(2. Operand2)
3. Pcurrent
4. Trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.FLINVE)

7.6.11.2 Floating Point Divide By Zero

This trap is caused when the divisor in a floating divide is zero. The operands are pushed.

Mnemonic: FL-DVDZ
Parameters: 1. Operand1
2. Operand2
3. Pcurrent
4. Trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.FLDVDZE)

7.6.11.3 Floating Point Overflow

This trap is caused when the magnitude of the result of a floating point arithmetic operation is greater than the largest representable floating point value in the indicated precision. The unrounded wrapped result is pushed. The round status is 0 for ROUND=0 and STICKY=0, 1 for ROUND=0 and STICKY=1, 2 for ROUND=1 and STICKY=0, and 3 for ROUND=1 and STICKY=1.

Mnemonic: FL-OVF
Parameters: 1. Wrapped result
2. Round status
3. Pcurrent
4. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.FLOVFE)

7.6.11.4 Floating Point Underflow

This trap is caused for Floating Point Underflow as defined in IEEE Floating Point Standard. The wrapped result and round status are computed as they are in the overflow case.

Mnemonic: FL-UNF
Parameters: 1. Wrapped result
2. Round status
3. Pcurrent
4. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.FLUNFE)

7.6.11.5 Floating Point Inexact Result

This trap is caused when the result of a floating point operation is inexact as defined by the IEEE Floating Point Standard. The result pushed is the rounded or overflowed result. The Round status is as in overflow and underflow.

Mnemonic: FL-INX
Parameters: 1. Rounded or Overflowed result
2. Round status
3. Pcurrent
4. Trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.FLINXE)

7.6.12 Integer Traps

7.6.12.1 Fixed Point Divide By Zero

This trap is caused when an attempt is made to divide an integer by zero. When divide by zero is detected, the destination is unchanged.

Mnemonic: INTDVDZ
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.INTDVDZE)

7.6.12.2 Fixed Point Overflow

This trap is caused when the result value is outside the allowable range of integer values for the destination operand. On overflow in ADDt, SUBt, NEGt, ABSt, ASLt, and MPYt, the lower t bytes of the result is returned. For CONVERT the largest positive integer if the source was positive and the largest negative integer if the source was negative is returned.

Mnemonic: INTOVF
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.INTOVFE)

7.6.13 Decimal Traps

7.6.13.1 Decimal Divide By Zero

This trap is detected when the divisor is zero in a decimal divide. When the divide by zero is detected, the destination operand is not changed.

Mnemonic: DECDVDZ
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.DECDVDZE)

7.6.13.2 Decimal Overflow

This trap is detected for decimal operations when the result is larger than can fit in the destination operand. When the overflow is detected, the destination is affected in the following ways:

1) If the trap is enabled, the destination operand is not changed.

2) If the trap is disabled, the result is stored left truncated into the destination operand.

Mnemonic: DECOVF
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB2.DECOVFE)

7.6.13.3 Decimal Invalid Length

This trap is detected when the value of the length operand is either less than zero or greater than 31.

Mnemonic: DECINVL
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.13.4 Invalid Decimal Digit

This trap is detected for some decimal operations when an invalid decimal digit is found. See the description of each decimal instruction listed below for a list of which characters/digits are invalid for that instruction.

Mnemonic: DECINVDG
Parameters: 1. Pcurrent
2. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.14 Debug Trap Conditions

7.6.14.1 Break Instruction

This trap is caused when executing the BREAK instruction.

Mnemonic: DBBRKINS
Parameters: 1. Operand
2. Pcurrent
3. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.14.2 Procedure Trace Trap

This trap is caused at the start of BRX, CALL, or CALLX instructions when the PTE bit in STATUSB is found set.

Mnemonic: DBCALL
Parameters: 1. Pcurrent
2. trap #
Trap Type: step continuable
Enabling: individually enabled
(STATUSB1.PTE)

7.6.14.3 CHECKA Instruction

This trap is caused when executing the CHECKA instruction if the bit CBA in STATUSB register is set.

Mnemonic: DBCHECKA
Parameters: 1. Instruction operand
2. Pcurrent
3. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB.CB.CBA)

7.6.14.4 CHECKB Instruction

This trap is caused when executing the CHECKB instruction if the bit CBB in STATUSB register is set.

Mnemonic: DBCHECKB
Parameters: 1. Instruction operand
2. Pcurrent
3. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSB.CB.CBB)

7.6.14.5 Single Instruction Trace

This trap is caused at the end of executing an instruction when the single instruction trace bit (SIT) in the STATUSA register is found set.

The SIT bit is always cleared as part of trap initiation. Software must explicitly reenable the single instruction trace by setting the SIT value to one in the stack marker in order to continue single instruction execution.

Mnemonic: DBSIT
Parameters: 1. Pcurrent
2. trap #
Trap Type: continuable
Enabling: individually enabled
(STATUSA.SIT)

7.6.15 Semaphore Traps

7.6.15.1 Semaphore Overflow

This trap is caused for the instructions UP, DOWN, and TESTDOWN when incrementing or decrementing the 31-bit semaphore value causes a 31-bit overflow.

Mnemonic: SEMAOVF
Parameters: 1. Logical address of the first operand (semaphore)
2. Pcurrent
3. Trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.15.2 Down Semaphore

This trap is caused for the instruction DOWN, when decrementing the 31-bit 2's complement semaphore value of the operand causes it to drop below zero.

Mnemonic: SEMADOWN
Parameters: 1. Logical address of the operand (semaphore)
2. Pcurrent
3. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.15.3 Up Semaphore

This trap is caused for the instruction UP, when incrementing the 31-bit 2's complement semaphore value of the operand leaves it at or below zero.

Mnemonic: SEMAUP
Parameters: 1. Logical address of the operand (semaphore)
2. Pcurrent
3. trap #
Trap Type: continuable
Enabling: permanently enabled

7.6.16 Vision Mode Switch

This trap is the entry point for a switch from HP3000 mode to Vision mode. See section 10.5.1.2 for details.

Mnemonic: SWITCHN
Parameters: 1. trap #
2. Pcurrent
Trap Type: continuable
Enabling: permanently enabled

7.6.17 TRY/UNTRY Traps

7.6.17.1 TRY or UNTRY Violation

This trap is caused for an illegal TRY or UNTRY instruction. This will happen if TRY or UNTRY is used on the ICS.

Mnemonic: TRYV
Parameters: 1. TRYoffset
2. Pcurrent
3. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.18 Virtual Addressing Traps

7.6.18.1 PDINSERT Inconsistent Page Number

This trap is caused for the instruction PDINSERT when the physical page number provided by the instruction does not equal the physical page number contained in the corresponding PDIR entry.

Mnemonic: ADRPDIR
Parameters: 1. Physical Page number in PDIR
2. Pcurrent
3. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.19 Page Absent Traps

7.6.19.1 Page Absent

This trap is caused when a page containing the byte being accessed is not present in physical memory. This trap is used for all absent pages except the page on top of the stack; the ADRPAGETOS fault is used for that. ADRPAGEABS is in all respects, including handling of parameters, a normal trap.

Mnemonic: ADRPAGEABS
Parameters: 1. Byte Offset (POFF)
2. Virtual Page Number (VPN)
3. Logical Address (LA)
4. Pcurrent
5. trap #
Trap Type: restartable
Enabling: permanently enabled

7.6.19.2 Top of Stack Page Absent

This trap is caused when the top of stack page is referenced and is not present in physical memory.
This trap is very special in that all other traps use the current stack to push a marker. The sequence of events for internal interrupts is therefore used. The top of stack page absent handler executes on the interrupt control stack.
More information can be found in section 7.7.

Mnemonic: ADRPAGETOS
Parameters: {0. Overflow Information}
1. Byte Offset (POFF)
2. Virtual Page Number (VPN)
3. Logical Address of S (LA)
4. Pcurrent
5. trap #
Trap Type: restartable
Enabling: permanently enabled

7.7 Top of Stack Page Faults

All stack objects, except the interrupt control stack, are paged objects. Some activities involve pushing information onto the stack, including:

1) an instruction explicitly references the stack as an operand; e.g. PUSH, CALL and SWITCH for the Vision mode stack and many instructions for the HP3000 mode stack.

2) an instruction encounters a page absent condition and the intermediate state information for the instruction must be put on the stack. The instruction in progress (IIP) bit described in chapter 4 refers to this case.

3) an instruction execution results in some conditions that are handled as user traps. In this case the instruction pushes stack markers as well as parameters for these conditions onto the stack.

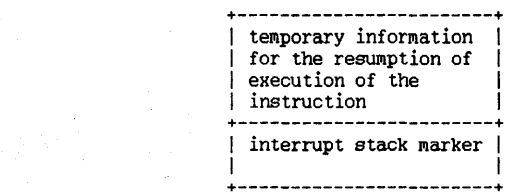
4) a condition such as an external interrupt causes a transfer of control from the user's stack onto the interrupt control stack. In this case an interrupt marker is pushed onto the stack.

However, two obstacles could prevent information from being pushed onto the stack:

1) the page containing the byte pointed to by S is not present in physical memory (ADRPAGETOS)

2) the logical offset S[32..63] attains the length of the stack object (UB-LB). This is the stack overflow condition (STKOVF).

In either case the information normally saved on the stack must be saved in a different location. The VISION architecture specifies the Interrupt Control Stack of the executing processor as the location to store the context when the stack page absent condition is detected. In general, the information to be saved can be divided into two parts. The illustration on the next page shows this:



The VISION architecture does not define at which point during an instruction a top of stack page absent condition is detected. That is, if during pushing any information onto the stack the page absent condition is detected, implementations are free to place any part of the above information onto the user's stack at S or onto the Interrupt Control Stack. In particular, implementations are free to push all the above information onto the ICS when it detects that not all of it fits onto the user's stack.

VISION specifies the following to be the same for all hardware implementations:

1) The value of S stored in the TCB is the same independent of where the information is actually saved. In all cases the S value is updated as though the information were placed on the user's stack.

2) The amount of overflow information pushed onto the Interrupt Control Stack can be computed as follows: subtract from the value of S (pointing into the Interrupt Control Stack) the value of QI, and further subtract the length of the argument list of the page fault trap handler.

3) This information must be moved immediately by software from the Interrupt Control Stack to some memory resident area so that the handler can IEXIT from the ICS. The move can be accomplished by a MOVEC instruction using the following operands:

   ARGLEN: length of argument list for trap handler (32 bytes)
   SRC: starting address of source information (=QI)
   RES: starting address of some resident destination area big enough to receive the information
   L: length of move, computed as: S-QI-ARGLEN

   MOVEC L, SRC, RES

   After the page(s) missing from the user's stack have been

brought into physical memory, the information can be moved from the memory resident page to the user's stack by MOVEC using these operands:

RES: address of the memory resdient page
DST: destination area, computed as the S value in the TCB minus (Si-QI-LA) where Si is the value of S (pointing into the ICS) on entry to the trap handler
L: same as above

```
MOVEC  L, RES, DST
```

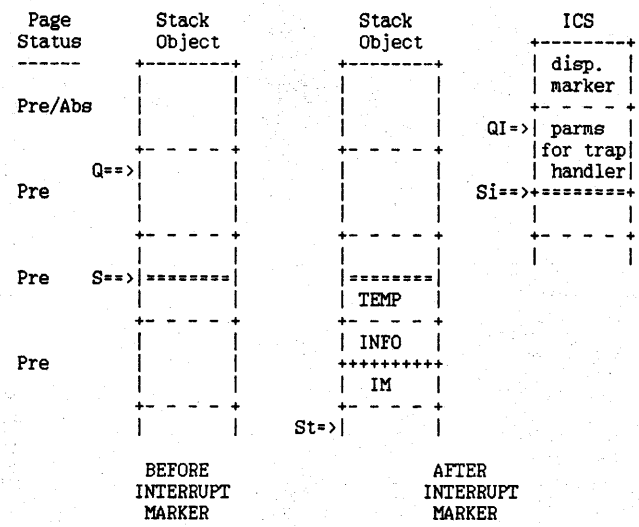
After this move has completed, the user's stack will appear as though the absent page condition had never occurred.

4) For all of the user's stack markers that were pushed onto the ICS, the value of the Qold[32..63] in the stack marker is relative to the user's stack and not the ICS. In other words, those markers are treated as raw data; they must never be used in EXIT or IEXIT when still on the ICS.

Three cases are sketched with respect to the saving of information. The following notation is used in these examples:

- - -	indicates a page boundary
=====	indicates the boundary value of the base register
Pre	indicates a virtual page present in physical memory
Abs	indicates a virtual page absent from physical memory
St	indicates the value of S stored in the TCB after the interrupt marker is pushed
TEMP	temporary information left on the stack as part of the execution of the previous instruction
IM	denotes the Interrupt Marker
Si	indicates the value of S after entry to the trap handler (points into the ICS)

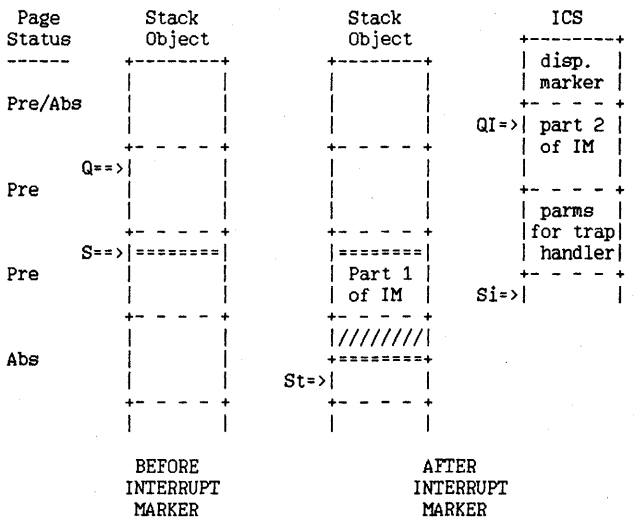
Case 1 -- The stack pages are present in memory.



Note that in this case the information to restart the interrupted instruction and the task's interrupt marker fit onto resident pages in the task's stack and the only information pushed onto the ICS is parameters for the page fault handler.

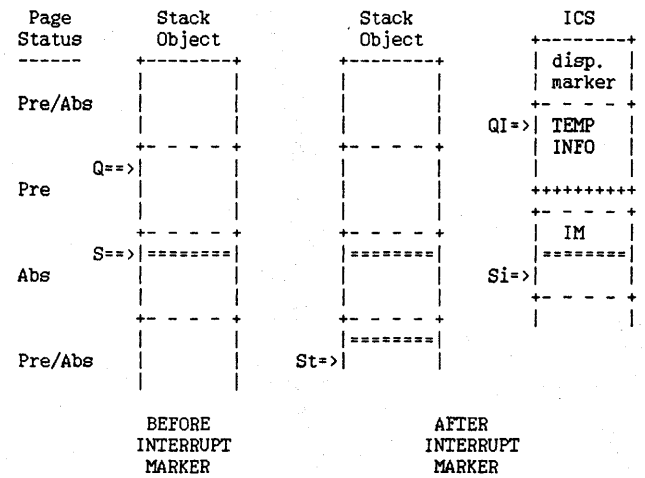
Case 2 -- The current S page is present but the next virtual page is absent. In this case the architecture does not define how much information is pushed onto the current S page before information is pushed onto the ICS. This results in the possibility of the information being saved in two parts as shown below.

Case 2A -- part of the information is pushed onto the user's stack



The stack page fault handler gets control after the interrupt marker part 2 has been pushed onto the ICS. The size of part 2 can be determined from calculating Si-QI-LA at the entry to the page fault handler.

Case 2B -- The current S page is absent. This case includes bothwhen the following page is present or absent.



In this case all of the information is pushed onto the ICS. As in Case 2A the amount of information on the ICS can be computed from (Si-Qi-LA) at entry to the page fault handler on the ICS.

7.8 ICS Mechanism

The Interrupt Control Stack (ICS) is a fixed size, memory resident structure. The location of the ICS is kept in a processor register. This location can only be changed through the MOVEtSP8 instruction.

All Vision mode external and internal interrupts execute on the ICS. A few Vision mode traps, such as Page Absent, Top of Stack Page Absent and Stack overflow, also execute on the ICS. The remaining traps are handled on the current task's stack.

All HP3000 mode internal interrupts as well as HP3000 mode page fault traps and stack overflow traps are directed to the Vision mode environment to execute on the ICS. The rest of the HP3000 mode traps are handled on the task's HP3000 mode stack. (See the Architectural Control Document for HP3000 Mode for a list of the traps supported on VISION.)

The dispatcher also executes on the ICS. There is a special stack marker permanently located at the bottom of the ICS, known as the dispatcher marker. It contains the information necessary to locate the dispatcher code and begin execution of the dispatcher.

While executing on the ICS, the ICS flag in STATUSC is set. The flag is set when the ICS environment is established for executing the dispatcher or an interrupt service routine. It is cleared by the Interrupt Exit Instruction (IEXIT) when it determines that the exit is to a procedure that does not execute on the ICS. The STATUSC.ICS flag is not directly accessible by any instruction.

There is a separate ICS for each processor in a shared-memory multi-processor configuration.

INPUT/OUTPUT DATA STRUCTURES

CHAPTER 8

This chapter will eventually describe the data structures that must be understood jointly by processor hardware and by I/O hardware.

VISION ARCHITECTURE CONTROL DOCUMENT 07/31
DO NOT COPY -- HP PRIVATE INFORMATION

SYSTEM INITIALIZATION — CHAPTER 9

9.1 Virtual Object Initialization

Virtual address space is organized as 2^32 virtual objects of 2^32 bytes each (see section 2.2).

Virtual Objects 1 through 5 are reserved for special areas which are allocated in physical memory and mapped into virtual space during system initialization.

Virtual Object		
0		(reserved)
1	SYSCOM	The System Communications Area
2		(reserved)
3	HASH	The Hash Table
4	PDIR	The Physical Page Directory
5	PMEBUF	The Primary Macro Environment Buffer

9.2 The System Communications Area

The System Communications Area (SYSCOM) is a memory resident buffer used by hardware and for communications with the Control Support Processor (CSP), if available.

SYSCOM is page aligned in virtual space as virtual object 1.

The SYSCOM.LENGTH field (+!00) records the total length in bytes of SYSCOM. The SYSCOM buffer is organized into sections. The number of sections is recorded in the SYSCOM.NUMBER_OF_SECTIONS field (+!04). Each section is a physically and virtually contiguous subset of SYSCOM, and can be located through a descriptor which defines the offset within SYSCOM to the start of the section, and the length in bytes of the section.

Section descriptors are located by fixed section numbers. The section number * 8 is the offset in SYSCOM to the section descriptor. Once a section is defined in SYSCOM a fixed section number is assigned. New implementations may add sections to SYSCOM, but they cannot remove sections.

The System Communications Area is partitioned into at least six main sections identified below:

1	SYSCOM.ENV_SECTION	Environmental Section
2	SYSCOM.ID_SECTION	Identification Section
3	SYSCOM.DIAG_SECTION	Diagnostics Section
4	SYSCOM.HARD_SECTION	Harware Reserved Section
5	SYSCOM.LOAD_SECTION	Load Section
6	SYSCOM.DUMP_SECTION	Dump Section

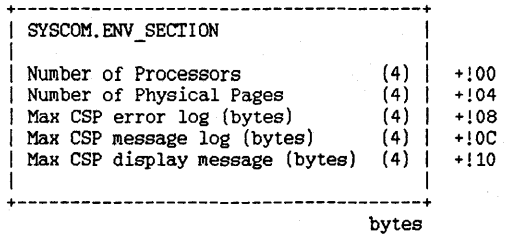
To locate the hardware reserved section of SYSCOM, for example, multiply section number 3 * 8 bytes = !18 bytes offset to the section descriptor.

The System Communications Area

		SYSCOM.LENGTH	(4)	+!00
		SYSCOM.NUMBER_OF_SECTIONS	(4)	+!04
	1	SYSCOM.ENV_SECTION.OFFSET	(4)	+!08
		SYSCOM.ENV_SECTION.LENGTH	(4)	+!0C
	2	SYSCOM.ID_SECTION.OFFSET	(4)	+!10
		SYSCOM.ID_SECTION.LENGTH	(4)	+!14
+->	3	SYSCOM.HARD_SECTION.OFFSET	(4)	+!18
		SYSCOM.HARD_SECTION.LENGTH	(4)	+!1C
	4	SYSCOM.DIAG_SECTION.OFFSET	(4)	+!20
		SYSCOM.DIAG_SECTION.LENGTH	(4)	+!24
	5	SYSCOM.LOAD_SECTION.OFFSET	(4)	+!28
		SYSCOM.LOAD_SECTION.LENGTH	(4)	+!2C
	6	SYSCOM.DUMP_SECTION.OFFSET	(4)	+!30
		SYSCOM.DUMP_SECTION.LENGTH	(4)	+!34

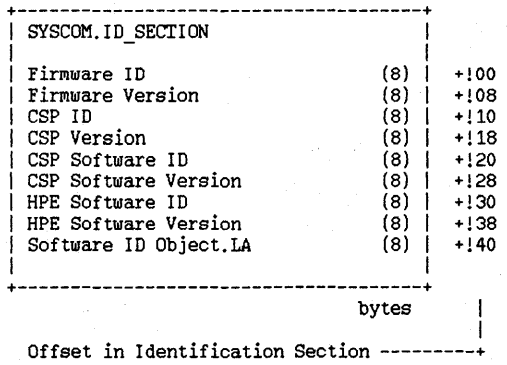
9.2.1 The Environment Section of SYSCOM

The Environment Section of SYSCOM is defined as section number 1 of SYSCOM and can be located through the section descriptor found at an offset of +!08 bytes into SYSCOM.



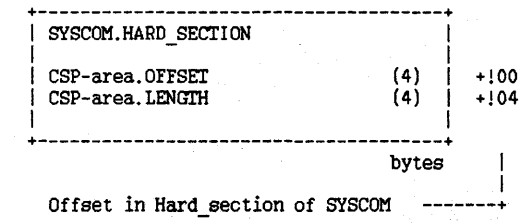
9.2.2 The Identification Section of SYSCOM

The Identification Section of SYSCOM is defined as section number 2 of SYSCOM and can be located through the section descriptor found at an offset of +!10 bytes into SYSCOM.



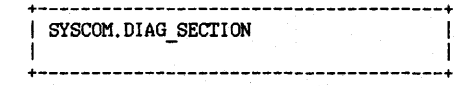
9.2.3 The Hardware Reserved Section of SYSCOM

The Hardware Reserved Section of SYSCOM is defined as section number 3 of SYSCOM and can be located through the section descriptor found at an offset of +!18 bytes into SYSCOM.



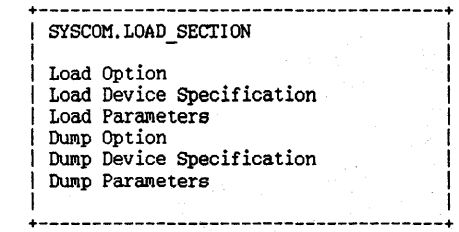
9.2.4 The Diagnostics Section

The Diagnostics Section of SYSCOM is defined as section number 4 of SYSCOM and can be located through the section descriptor found at an offset of +!20 bytes into SYSCOM.



9.2.5 The Load Section of SYSCOM

The Load Section of SYSCOM is defined as section number 5 of SYSCOM and can be located through the section descriptor found at an offset of +!28 bytes into SYSCOM.



9.2.6 The Dump Section of SYSCOM

The Dump Section of SYSCOM is defined as section number 6 of SYSCOM and can be located through the section descriptor found at an offset of +!30 bytes into SYSCOM.

All Vision processors, when not running, can be made to save their current register state into the Dump Section of SYSCOM by means not defined in this document.
Global computer context is deposited into fixed locations.

SYSCOM.DUMP_SECTION		
HASH.PA	(4)	+!00
HASH.LENGTH	(4)	+!04
PDIR.PA	(4)	+!08
PDIR.LENGTH	(4)	+!0C
Group 0 Descriptor (GD0)	(16)	+!10
STATUSD	(4)	+!20
System Breakrange Descriptor	(16)	+!24
Time of Century Clock	(8)	+!34
SYSCOM.PA	(4)	+!3C
Implementation Dependent.OFFSET	(4)	+!40
Implementation Dependent.LENGTH	(4)	+!44
Processor Arch Record.OFFSET	(4)	+!48
Processor Arch Record.LENGTH	(4)	+!4C
	bytes	

Offset in Dump Section of SYSCOM

The Dump Section also contains space for a processor architectural dump record for each processor in the computer. The first processor record can be located through the offset and length pair located in the dump section (+!44). Additional processor records are linked together through the next processor field in the processor record (+!D4). A length of 0 bytes is used to indicate that no further records follow.

Processor Architectural Record		
STATUSA	(4)	+!00
STATUSB1	(4)	+!04
STATUSB2	(4)	+!08
STATUSC1	(4)	+!0C
STATUSC2	(4)	+!10
QI	(8)	+!14
TCB.LA	(8)	+!1C
TCBX.LA	(8)	+!24
X0 .. X15	(64)	+!30
B0 .. B5	(48)	+!70
Q	(8)	+!A0
S	(8)	+!A8
Program Counter	(8)	+!B0
Task Clock	(8)	+!B8
Interval Timer	(8)	+!C0
Processor Serial Number (*)	(8)	+!C8
Processor Dependent Record.OFFSET	(4)	+!CC
Processor Dependent Record.LENGTH	(4)	+!D0
Next Processor Record.OFFSET	(4)	+!D4
Next Processor Record.LENGTH	(4)	+!D8
	bytes	

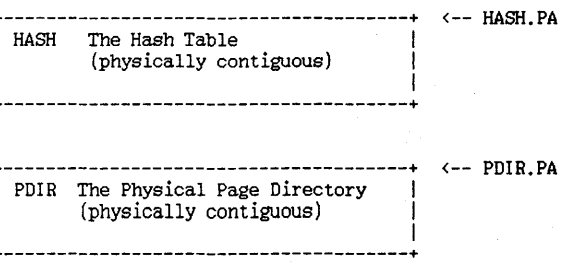
Offset in Processor Architectural Record

(*): if supported

Optional resident contiguous buffers for dumping implementation dependent information can be allocated and linked to either the global record or to any processor architectural record. A length of 0 bytes can be used to skip this option.

9.3 The Hash Table and Physical Page Directory

Virtual Object



During system initialization all software addressable memory is mapped into virtual space by hardware. The size and physical location of the hash table and the physical page directory are committed at this point.

The hash table (HASH) must be contiguous in physical memory and is initially mapped as virtual object 3. The size of HASH is a function of memory size and load options.

The physical page directory (PDIR) must be contiguous in physical memory and is initially mapped as virtual object 4. The size of PDIR is a function of memory size.

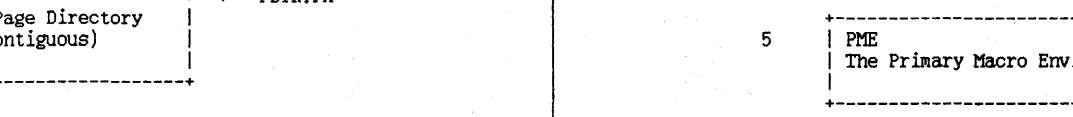
Hardware may choose to not associate certain phyical pages with virtual pages. The virtual page number (VPN) field in PDIR entries will be set to 0 by convention to indicate that no virtual page association has been made.

9.4 The Primary Macro Environment Buffer

A Primary Macro Environment (PME) is a pre-built, bootable, macro code image.

The Primary Macro Environment Buffer (PMEBUF) is a pre-mapped memory resident buffer which will be loaded with a bootable macro code image.

Virtual Object



PMEBUF is contiguous in virtual space and is initially mapped as virtual object 5.

During system initialization hardware allocates a few physical pages for SYSCOM, PDIR, and HASH as described in sections 9.2 and 9.3. Then the remaining physical pages are mapped into the PME buffer.

In contrast to SYSCOM, PDIR and HASH, the PME buffer need not remain resident in physical memory once software executes.

9.4.1 Loading the Primary Macro Environment Buffer

Virtual Object

5	PME The Primary Macro Environment Buffer		
	PME.LENGTH	(4)	+!00
	PME Checksum	(4)	+!04
	Group 0 Descriptor (GD0)	(16)	+!08
	TCB.LA	(8)	+!18
	TCB.VA	(8)	+!20
	QI.LA	(8)	+!28
	QI.VA	(8)	+!30
	CST descriptor	(8)	+!38
	DST descriptor	(8)	+!40
	reserved for expansion		+!48
	Macro Code Image		+!100

The first 256 bytes of each Primary Macro Environment serve as a descriptor of the PME.

The PME.LENGTH field (!0) defines the length of the image in bytes and can be used to ensure that the entire image will fit into the pre-mapped buffer. The PME Checksum (!4) can be used to insure that the image has been properly loaded.

The group 0 descriptor (GD0 +!08) defines the location of ODT0 within the PME. Since the PME is constructed to be loaded into PMEBUF, GD0.VON will always be virtual object 5. GD0.LB must be page aligned in virtual space. GD0.UB is equal to PME.LENGTH 1. GD0.LON will vary from PME to PME.

The TCB.LA field (!18) contains the logical address of a pre-built Task Control Block within the PME. TCB.VA (+!20) contains the virtual address of the TCB.

The QI.LA field (+!28) contains a logical pointer to the dispatcher marker on the Interrupt Control Stack. The QI.VA field (+!30) contains the virtual address of the dispatcher marker.

The CST descriptor (+!38) defines the object number in group 0 where the CST starts, and the length in bytes of the CST.

The DST descriptor (+!40) defines the object number in group 0 where the DST starts, and the length in bytes of the DST.

9.5 The Macro Code Launch

The following sequence of steps are taken for macro code launch.

1) Allocate physical pages for PDIR, HASH, SYSCOM, and PMEBUF, and map these virtual objects.

2) Load the PME into the PMEBUF in memory.

3) Using the PME descriptor do:

   Set the ODT0 registers. Now logical addressing is defined.

   Find the logical and physical address of the TCB.

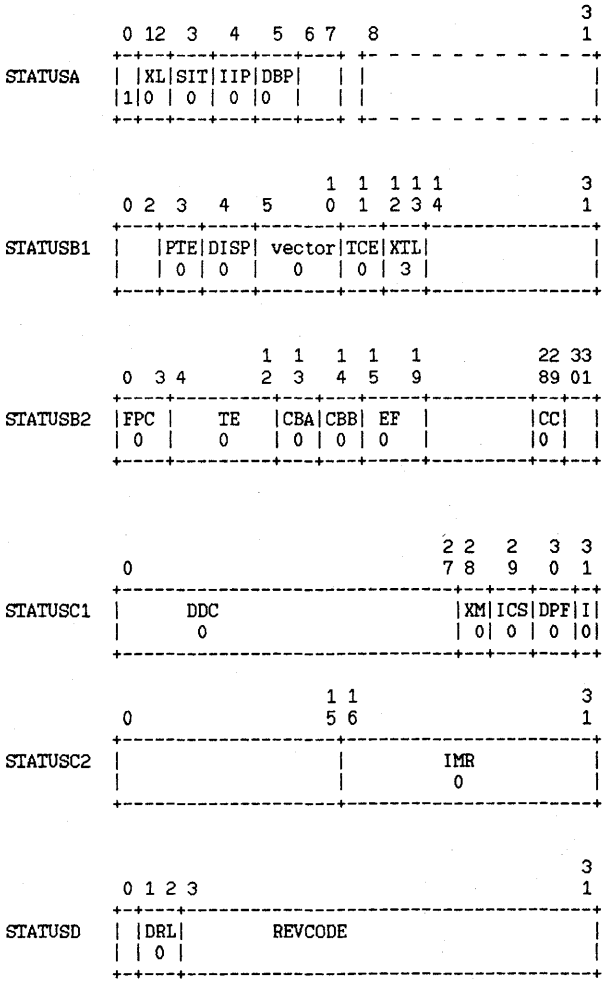
   Set QI to point into the ICS object.

   Locate the CSTs and the DSTs.

4) Set the initial state of the processor such that it will run uninterrupted at the highest privilege level. See section 9.7 for a summary of the initial state.

The cold load hardware then executes the algorithm described under the LAUNCH instruction to initiate the launching of software. The task pointed to by the TCB is launched.

9.6 Initial State Summary



HP/3000 MODE | CHAPTER 10

10.1 INTRODUCTION

A mode of execution is available which provides the software architectural environment of the HP/3000 system. This is called (HP/3000) COMPATIBILITY mode to distinguish it from the normal NATIVE mode of the VISION architecture.

The complete architectural definition of Compatibility mode is divided into two parts:

First, Chapter 10 describes the relationship between Compatibility and Native mode architectures. The purpose is to identify the specific features required of the VISION architecture to allow the existence of Compatibility mode in a manner which does not affect the inherent integrity of Native mode operations. Discussions progress from a generalized overview of the Compatibility and Native mode environments to the actual detail descriptions of the manner by which System and Task control structures of Compatibility mode are implemented and managed using the VISION architecture.

Second, the addendum to the ACD titled 'HP/3000 Compatibility Mode' continues the description of Compatibility Mode but from a different viewpoint. It provides the complete details of Compatibility mode from the perspective of both User and Privileged mode programmers. The instruction sets, data structure formats, addressing modes, traps, and environmental concepts are described.

10.2 ENVIRONMENTAL OVERVIEW

The two modes, Native and Compatibility, are very distinct even though they coexist and share access to physical resources. Instruction formats, data formats, and addressing modes are different. In particular, the Native mode architecture supports arbitrary byte alignment, a very large address space, and a nominal four-byte word size, while the Compatibility mode architecture requires word alignment, has amoderate size address space, and uses a two-byte word size.

The differences are so extensive that each mode is considered to be an independent architectural model designed to support and execute User programs in a particular manner. This results in the task (process) model being different in each mode. To switch execution from one mode to the other is conceptually equivalent to a process switch.

The primary objective of Compatibility mode is to provide an execution environment for User mode programs identical to that on the HP/3000 system. A secondary objective is to provide an execution environment for Privileged mode code subject to the condition that there is guaranteed protection against Native mode structures being accessed directly from Compatability mode code. To achieve this level of security could mean that the privileged mode set of instructions available in Compatibility mode are a subset of that in the HP/3000 system. These objectives are accommodated as follows:

On HP/3000 system two types of addressing are provided:

* Addressing into segmented code and data structures is the most common form. In User mode it is the only type and is fully bounds checked. In Privileged mode it is not always bounds checked.

* Absolute addressing is allowed only in Privileged mode with absolutely (!) no checks.

Compatibility mode provides both types of addressing but does so with full protection against unwarranted access into Native mode by encapsulating the Compatibility mode environment (address space) using the Native mode ODT structures. The formats of Compatibility mode ODT descriptors are identical to Native mode ODT formats. Consider the two addressing types:

* Segment adddressing - all code and data segments are Native objects. The ODT entry contains a type field which specifies certain Compatibility segment types. The management of these ODTs (CST and DST) is done by Native mode code and (trusted) microcode only. Compatibility code accesses the CST and DST only through microcode, never directly.

* Absolute addressing - emulated using a special Native object accessible through microcode. Viewed by Native mode it is a logical address space. To Compatibility privileged users it still looks like the `real' absolute memory. There is no correspondence between the absolute addresses used by 3000 Compatibility mode and the real main memory addresses.

So now, instructions can be executed safely, but how are the Native task and Compatibility process environments related? Within a logical task domain, there may exist the need to execute in both execution modes(in a serial manner, not in parallel). In such a case, two physical tasks/processes are apparent, one for each mode having unique code and data (stack included) structures. The common shareable element is the single Hardware Task Control Block (TCB). Switch mode instructions are provided in both modes to allow an environment switch to occur to the other mode. Even though execution switches back and forth between modes, each mode in execution is still an instance of executing a single logical task. There is one Dispatcher and one Interrupt Control Stack (ICS) in the architecture which exist only in Native mode and it is capable of launching either task into the appropriate mode.

Launching a task/process into Compatibility mode means establishing the Registers which are specifically used by the Compatibility instruction sets. The precise mode of execution is determined at any time by the XM field of the STATUSC register.

```
STATUSC.XM = 0   Native mode
STATUSC.XM = 1   Compatibility mode
```

In summary, Compatibility mode is completely and safely emulated under Native architectural control to provide an environment for Compatibility mode Users which is almost an exact replica of the HP/3000 environment. Certainly, normal (User mode) users do not notice any difference.

10.3 SYSTEM CONTROL STRUCTURES

The following Native mode data structures are required to manage and control Code segments, Data segments, and Absolute memory for Compatibility mode operations:

* CST - Code Segment Table

* DST - Data Segment Table

* ABS - Absolute Memory Object

These basic tables cannot be accessed directly by Compatibility mode Users, they are only accessed by hardware to execute the appropriate instructions.

10.3.1 CST - Code Segment Table

The CST is a contiguous block of entries in the ODT for group 0. The ODT entries are of type 4 or 5 'HP3000 mode code object'.

A CST number from Compatibility mode is converted into the appropriate ODT entry by locating the base of the CST block in ODT (group 0) and indexing through the ODT entries using the CST number.

The Base and Length of the CST are defined at system initialization time and passed to the microcode using the MOVEtSP8 instruction.

Base - 29 bit object number pointing to the entry in the ODT for group 0 corresponding to CST 0.

Length - 32 bit integer specifying the length of the CST in bytes (0<=Length<=192*16). A zero Length implies the absence of a CST.

They are now protected in dedicated memory from unwarranted software access. Microcode uses them to locate the CST and perform bounds checking on the CST index. The legal range of the CST index is:

```
1 <= CST index <= 191
```

An explicit reference to CST 0 will cause a 'CST Violation' trap to occur.

10.3.2 DST - Data Segment Table

The DST is a contiguous block of entries in the ODT for group 0. The ODT entries are of type 3 'Data' object.

A DST number from Compatibility mode is converted into the appropriate ODT entry by locating the base of the DST block in ODT (group 0) and indexing through the ODT entries using the DST number.

The Base and Length of the DST are defined at system initialization time and passed to hardware using the MOVEtSP8 instruction.

Base - 29 bit object number pointing to the entry in the ODT for group 0 corresponding to DST 0.

Length - 32 bit integer specifying the length of the CST in bytes. A zero length implies the absence of a DST.

They are now protected in dedicated memory from unwarranted software access. Hardware uses them to locate the DST and perform bounds checking on the DST index. An explicit reference to DST 0 will cause a 'DST Violation' to occur.

10.3.3 ABS - Absolute Memory Object

The ABS is a special object in group 0 which provides a logical representation of Absolute memory to Compatibility mode instructions.

The ABS is defined at system initialization time and the ODT entry used is the ODT entry equivalent to DST 0 which is inaccessible to instructions but readily available to hardware. The absence of a DST will cause all absolute addressing to fail and generate an `Absolute Address Violation' trap.

It is now protected in dedicated memory from unwarranted software access and used only by hardware for all absolute memory references. The legal size of the ABS is defined to be:

$$0 <= \text{ABS size} < 128KB$$

Several instructions require System Global Region type of access i.e. through Absolute address 1000 octal. As for all absolute addressing, the ABS is used by hardware for such accesses.

10.4 TASK CONTROL STRUCTURES

10.4.1 CSTX - Code Segment Table Extension

The local code domain defined by the CSTX concept in HP3000 Compatibility mode is emulated in Native mode as follows:

The CSTX is a contiguous block of entries in the ODT for group 0 which have been assigned to a given task. The TCB contains a descriptor of the CSTX to define the base of the CSTX and the length of CSTX, to allow conversion of the CST index to the corresponding ODT entry (see Section 4.10).

The CSTX contains the CST indices in the range

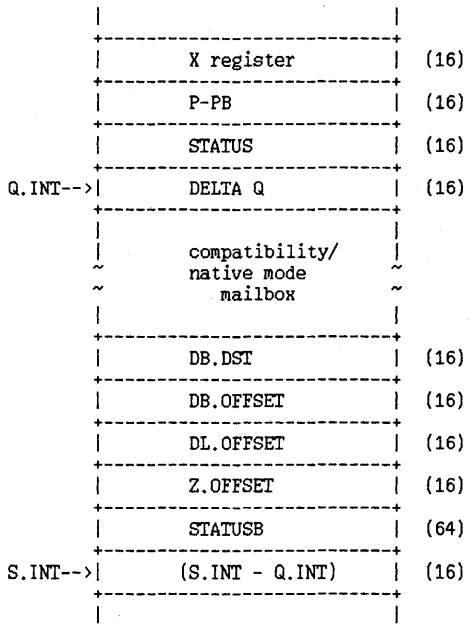
$$192 <= \text{CST index} <= 255$$

where the first legal entry in the CSTX is CST 193.
An explicit reference to CST 192 will cause a `Not Code Segment' trap to occur.

10.4.2. Interrupt Stack Marker

The interrupt stack marker is used to mark the upper limit of the stack on external interrupts, traps, transfers to the Dispatcher, and the Switch operation.

The interrupt marker generated in Compatibility mode is presented below. The one for Native mode is presented in Section 5.1.2.



Notes:

1) The number in parenthesis following each box reflects the appropriate number of bits of specification.

2) X register, P-PB, STATUS, DELTA Q are the normal contents of a Compatibility mode procedural stack marker.

3) DB.DST = 0 if DB set to ABS (absolute memory)
   <> 0 if DB set to stack or data segment
   DB.OFFSET defines the displacement (in units of 16 bits) into the corresponding object.

4) DL.OFFSET, Z.OFFSET are the current values of the DL and Z registers given as displacements into the stack object.

5) STATUSB is the current STATUSB register contents.

6) S.INT is the interrupted S value stored into TCB.SC. The value of Q can be calculated from the contents (S.INT-Q.INT).

10.4.3 TCB contents known to Hardware

The additional information required in the TCB by the hardware to support Compatibility mode instructions and the special instructions in Native mode to interface with Compatibility mode are specified below.
See Section 5.8 for complete TCB details.

CSTX - CSTX descriptor (see 10.4.1)

XM - mode of execution of the task
= 0 Native mode
= 1 Compatibility mode
(1 bit)

SN - logical address of top-of-stack of Native stack when capped by an interrupt stack marker - it points to the next byte following the interrupt marker.
(64 bits)

SC - logical address of top-of-stack of Compatibility stack when capped by an interrupt stack marker - it points to the last 16-bit word of the interrupt stack marker.
(64 bits)

SWIP - switch in progress flag.
(1 bit)

10.5 MODE SWITCHING

Mode switching refers to the operations which affect the execution mode flag XM in the STATUSC register.

STATUSC.XM = 0 Native mode
STATUSC.XM = 1 Compatibility mode

Native mode instructions and/or operations which can initiate a switch to Compatibility mode are:

IEXIT
SWITCH
RSWITCH

Compatibility mode instructions and/or operations which can cause a switch to Native mode are:

SWT
RSWT
DISP
External Interrupts
ICS Internal Interrupts

The following operations cause a transfer of execution to the ICS, in Native mode, from both Native and Compatibility modes.

DISP
External Interrupts
ICS Internal Interrupts

The impact of the two modes, Native and Compatibility, on the above declared instructions is discussed below.

10.5.1 Compatibility Mode Instructions

10.5.1.1 DISP

This instruction is used to enter the Dispatcher directly from the Compatibility mode process environment. If external interrupts are disabled then the Dispatcher pending flag is set and execution continues with no switch taking place.

This is a privileged instruction.

```
if STATUSC.IE = 0
then STATUSC.DRF := 1
else
   begin
   `PUSH2' X;
   `PUSH2' P-PB;
   `PUSH2' STATUS;
   `PUSH2' (S-Q+2) [0..14]
   Q := S;
   `PUSH2' DB.DST;
   `PUSH2' DB.OFFSET;
   `PUSH2' DL.OFFSET;
   `PUSH2' Z.OFFSET;
    PUSH8  STATUSB;
   `PUSH2' S-Q+2;
   TCB.SC := S;
   STATUSC.ICS := 1;
   STATUSC.DPF := 0;
   execute_case_2_of_IEXIT;
   end;
```


10.5.1.2 SWT

The SWT instruction provides a switch of the execution environment of a process from Compatibility mode directly to Native mode. The Compatibility mode stack is capped with an Interrupt Stack Marker, the appropriate mode flags changed, and control passed to the Native SWITCH trap routine on the Native mode stack which executes above the previous interrupt stack marker. Any interferences, such as Page Faults, aborts the operation after setting the `switch in progress' flag which then takes effect on the subsequent IEXIT to the process.

This is a privileged instruction.

```
if STATUSC.IE = 0
then Trap"INSSWITCH"
else
   begin
   `PUSH2' X;
   `PUSH2' P-PB;
   `PUSH2' STATUS;
   `PUSH2' (S-Q+2) [0..14];
   Q := S;
   `PUSH2' DB.DST;
   `PUSH2' DB.OFFSET;
   `PUSH2' DL.OFFSET;
   `PUSH2' Z.OFFSET;
    PUSH8  STATUSB;
   `PUSH2' S-Q+2;
   TCB.SC := S;
   TCB.XM := 0;
   TCB.SWIP := 1;
   execute_case_1_of_IEXIT;
   end;
```

10.5.1.3 RSWT

The RSWT is the reverse operation to a corresponding SWITCH instruction which occurred from Native mode and basically returns execution control back onto the Native mode stack environment. The Compatibility mode stack is capped with a register save to build the interrupt stack marker, the process mode flag is set to Native mode, and a relaunch of the Native mode process occurs.

This is a privileged instruction.

```
if STATUSC.IE = 0
then Trap"INSSWITCH"
else
   begin
   `PUSH2' DB.DST;
   `PUSH2' DB.OFFSET;
   `PUSH2' DL.OFFSET;
   `PUSH2' Z.OFFSET;
    PUSH8  STATUSB;
   `PUSH2' S-Q+2;
   TCB.SC := S;
   TCB.XM := 0;
   execute_case_1_of_IEXIT;
   end;
```


10.5.2 Native Mode Instructions

10.5.2.1 DISP

The DISP instruction is described in Section 6.2.9.6.

10.5.2.2 IEXIT

The IEXIT instruction is described in Section 6.2.9.8. The execution environment of a process is determined first by the PM flag, indicating Native or Compatibility mode, and then by the SWIP `switch in progress' flag to either trap to the SWITCH Trap routine or just perform a normal launch of the process by by reestablishing the registers from the interrupt stack marker.

10.5.2.3 SWITCH

The SWITCH instruction provides a switch of the execution environment of a process from Native mode directly to Compatibility mode. The Native mode stack is capped with an Interrupt Stack Marker, the appropriate mode flags changed, and control passed to the Compatibility SWITCH trap routine on the Compatibility mode stack which executes above the previous interrupt stack marker. Any interference, such as Page Faults, aborts the operation after setting the `switch in progress' flag which then takes effect on the subsequent IEXIT to the process.

This instruction requires Ring level 1.

```
if STATUSC.ICS = 1 or STATUSC.IE = 0
then Trap"INSSWITCH"
else
   begin
   PUSH_INTERRUPT_MARKER;
   TCB.SN := S;
   TCB.XM := 1;
   TCB.SWIP := 1;
   execute_case_1_of_IEXIT;
   end;
```


10.5.2.4 RSWITCH

The RSWITCH is the reverse operation to a corresponding SWT instruction which occured from Compatibility mode and basically returns execution control back onto the Compatibility mode stack environment. The Native mode stack is flushed to leave the old interrupt stack marker, the process mode flag set to Compatibility mode, and a relaunch of the Compatibility mode process occurs.

This instruction requires Ring level 1.

```
if STATUSC.ICS = 1 or STATUSC.IE = 0
then Trap"INSSWITCH"
else
   begin
   S := Q+120;
   TCB.SN := S;
   TCB.XM := 1;
   execute_case_1_of_IEXIT;
   end;
```

10.6 PROTECTION

The details of protection are integrated with those of Native mode objects in Chapter 2. In particular, refer to the discussion on object types and object access rights.

10.7 IMPLEMENTATION NOTES

1. All Compatibility mode objects, code and data segments, are assumed by hardware to be aligned on an even byte boundary.

VISION ARCHITECTURE CONTROL DOCUMENT 07/31
DO NOT COPY -- HP PRIVATE INFORMATION

SORTED LIST OF INSTRUCTIONS — APPENDIX

Section	Instruction
6.2.2.6	ABSt source.r, destination.w
6.2.2.1	ADDt term.r, sum.rw
6.3.3.1	ADDtD term.r, sum.rw
6.2.3.1	AND4 mask.r4, operand.rw4
6.2.3.7	ASLt shiftcount.r1, operand.rw
6.2.3.9	ASRt shiftcount.r1, operand.rw
6.2.5.9	BADD4 term.r4, dest.b
6.2.5.11	BCMP4 sourcea.b, sourceb.r4
6.2.5.12	BCMP8 sourcea.b, sourceb.r8
6.2.5.5	BGET4 source.b, dest.w4
6.2.5.1	BGET8 source.b, destination.w8
6.2.5.4	BMOVE8 source.b, dest.b
6.2.5.3	BMOVEADR source.m, dest.b
6.2.5.8	BPOP8 dest.b
6.2.5.7	BPUSH8 source.b
6.2.6.7	BREAK parameter.r4
6.2.6.4	BRX loi.r4
6.2.6.1	BR{GLEU} target.r4
6.2.5.6	BSET4 source.r4, dest.b
6.2.5.2	BSET8 source.r8, dest.b
6.2.5.10	BSUB4 term.r4, dest.b
6.2.5.13	BTEST8 source.b
6.2.6.2	CALL target.r4
6.2.6.3	CALLX loi.r4
6.2.6.10	CHECKA parameter.r4
6.2.6.11	CHECKB parameter.r4
6.2.6.13	CHECKHI source.r4, hibound.r4
6.2.6.12	CHECKLO source.r4, lobound.r4
6.5.1.2.1	CHNOP
6.5.1.2.9	CIS channel.r1, status.r1
6.4.6.1	CLRMR mrselect.r1
6.2.4.10	CMPB fillchar, lgtha, srca, lgthb, srcb, index
6.2.4.3	CMPC length.r4, stringa.m, stringb.m, index.w4
6.2.4.11	CMPT table, fillchar, lgtha, srca, lgthb, srcb, index
6.2.4.1	CMPt source1.r, source2.r
6.3.3.5	CMPtD sourcea.r, sourceb.r
6.3.3.17	CVAD length.r1, source.r, dest.w
6.3.3.18	CVDA length.r1, source.r, dest.w
6.3.3.11	CVDI length.r1, source.r, dest.w8
6.3.3.12	CVID length.r1, source.r8, dest.w

6.2.8.5 CVLAtVA operand.m1, virtaddr.w8
6.2.8.7 CVVAtPP virtaddr.r8, ppn.w4
6.2.1.15 DELETE wordcount.r4
6.2.9.1 DISABLE oldi.w1
6.2.9.6 DISP
6.2.2.4 DIVt divisor.r, dividend.rw
6.3.3.4 DIVtD divisor.r, quotient.rw
6.2.9.18 DOWN sema.mrw4
6.2.1.6 DPF value.r4, shiftcount.r1, mask.r4, target.rw4
6.2.1.12 DUP wordcount.r4, value.r4
6.2.9.2 ENABLE oldi.r1
6.2.6.8 ERROR
6.2.6.5 EXIT
6.2.1.14 EXTEND wordcount.r4
6.3.3.14 GETSIGN operand.r1, sign.w1
6.2.8.8 GrowGDO newlength.r4
6.2.8.6 HASH virtaddr.r8, hashindex.w4
6.2.9.11 IDLE
6.2.9.8 IEXIT
6.5.1.1.1 IFC
6.2.9.3 INTERRUPT pr.r4
6.5.2.1.3 IOC channel.r4, control.r4
6.5.2.1.2 IOR channel.r4, control.r4, data.w4
6.5.2.1.1 IOW channel.r4, control.r4, data.r4
6.4.5.7 IVB tcb.mr
6.2.9.7 LAUNCH tcbla.r8, tcbva.r8
6.4.6.3 LDMR mrselect.r1, source.r16
6.4.5.2 LDVLR source.r4
6.2.3.5 LSLt shiftcount.r1, bitfield.rw
6.2.3.6 LSRt shiftcount.r1, bitfield.rw
6.4.5.8 LVB tcb.mr
6.2.2.8 MODt divisor.r, dividend.rw
6.2.1.2 MOVEADR operand.m, destination.w8
6.2.1.8 MOVEBIT bitindex.r4, source.r1, bitarray.mrw
6.2.1.9 MOVEBLR fillchar, srcl, src, destl, dest
6.2.1.10 MOVEBRL fillchar, srcl, src, destl, dest
6.2.1.7 MOVEC length.r4, source.mr, destination.mw
6.3.3.9 MOVED length.r1, source.r, dest.w
6.2.9.17 MOVESEMA source.r4, sema.mw4
6.2.7.1 MOVEfSP4 selector.r1, destination.w4
6.2.7.3 MOVEfSP8 selector.r1, destination.w8
6.2.7.2 MOVEtSP4 selector.r1, source.r4
6.2.7.4 MOVEtSP8 selector.r1, source.r8
6.2.2.3 MPYt factor.r, product.rw
6.3.3.3 MPYtD factor.r, product.rw
6.4.6.5 MRAND mrasleect.r1, mrbselect.r1
6.4.6.4 MRNOT mrselect.r1
6.4.6.6 MROR mraselect.r1, mrbselect.r1
6.4.6.7 MRXOR mraselect.r1, mrbselect.r1
6.2.2.5 NEGt source.r, destination.w

6.2.6.9 NOP
6.2.3.2 NOT4 source.r4, destination.w4
6.2.3.3 OR4 mask.r4, operand.rw4
6.3.3.15 OVPUNCH sign.r1, operand.rw1
6.5.1.2.5 PAR response.w1
6.5.1.2.4 PDA response.w1
6.2.8.4 PDDEL ppn.r4
6.2.8.3 PDINS ppn.r4
6.2.2.9 POLYt degree.r1, polyn.mr, operand.rw
6.2.1.5 POPt destination.w
6.5.1.2.3 PRD response.w1
6.2.8.1 PROBE ring.r1, access.r1, address.r8, length.r4
6.2.9.4 PSDB
6.2.9.5 PSEB
6.2.1.4 PUSHADR operand.m
6.2.1.3 PUSHt source.r
6.4.5.6 PUVCSA tcb.mr
6.2.3.8 QUAD4 source.r4, destination.w4
6.5.1.1.4 RBYTE data.w1
6.5.1.2.2 RCL response.w1
6.5.1.2.6 RDP channel.r1, dest,w16, length.w1
6.2.2.7 REMt divisor.r, dividend.rw
6.2.1.13 REP wordcount.r4, value.r4, operand.mw
6.5.1.2.8 RIS channel.r1, status.w1
6.2.9.10 RSWITCH
6.4.5.1 RVLR
6.2.4.9 SCANUNTIL charset.mr, string.mr, index.rw4
6.2.6.6 SEXIT
6.5.1.2.10 SIS channel.r1, status.r1
6.3.3.7 SLD count.r1, length.r1, source.r, dest.w
6.3.3.8 SRD count.r1, lenght.r1, source.r, dest.w
6.4.6.2 STMR mrselect.r1, destination.w16
6.2.9.12 STOP
6.4.5.3 STVLR dest.w4
6.2.2.2 SUBt term.r, difference.rw
6.3.3.2 SUBtD term.r, difference.rw
6.2.9.9 SWITCH
6.2.9.15 SYNCIB operand.mc, length.r4
6.2.9.13 SYNCOD loi.r4
6.2.9.14 SYNCTCB tcb.r8
6.2.4.6 TESTA
6.2.4.7 TESTB
6.2.4.8 TESTBIT bitindex.r4, bitarray.mr
6.2.9.19 TESTDOWN sema.mrw4
6.2.4.4 TESTLSB source.r1
6.2.4.5 TESTOV
6.2.8.2 TESTREF va.r8
6.2.9.16 TESTSEMA sema.mrw4, result.w4
6.3.3.13 TESTSTRIP operand.rw1
6.2.4.2 TESTt source.r
6.3.3.6 TESTtD source.r

```
6.2.1.11    TRANSL  table.mr, length.r4, source.mr, dest.mw
6.2.7.5     TRY
6.2.7.6     UNTRY destination.w4
6.2.9.20    UP sema.mrw4
6.4.5.5     UVCSA
6.4.2.7     VABSt  vqual.r1, source.vr, abs.vw
6.4.4.3     VACCDt  vqual.r1, terms.vr, sumrw
6.4.4.2     VACCt  vqual.r1, terms.vr, sum.rw
6.4.2.2     VADDt  vqual.r1, terma.vr, termb.vr, sum.vw
6.3.3.10    VALD  length.r1, operand.rw
6.3.3.16    VALN  length.r1, operand.rw
6.4.3.1     VAND4  vqual.r1, facta.vr, factb.vr, and.vw
6.4.2.12    VASLt  vqual.r1, shiftcount.vr, target.vw
6.4.2.13    VASRt  vqual.r1, shiftcount.vr, target.vw
6.4.4.1     VCMPt  vqual.r1, field.r1, srca.vr, srcb.vr, mrsel.r1
6.4.4.8     VCOMPRSt  vqual.r1, terms.vr, compressed.vw
6.4.7.1     VCONVERT  vqual.r1, typer.r1, source.vr, dest.vw
6.4.2.5     VDIVt  vqual.r1, divd.vr, divsr.vr, quot.vw
6.4.4.9     VEXPNDt  vqual.r1, terms.vr, expanded.vw
6.4.4.6     VEXTt  vqual.r1, terms.vr, index.r, value.w
6.4.4.10    VGATHt  vqual.r1, source.vr, index.vr, destination.vw
6.4.4.7     VINSt  vqual.r1, terms.vw, index.r, newval.r
6.4.5.4     VINVAL  vrmask.r1
6.4.2.10    VLSLt  vqual.r1, shiftcount.vr, target.vrw
6.4.2.11    VLSRt  vqual.r1, shiftcount.vr, target.vrw
6.4.4.4     VMAXELt  vqual.r1, terms.vr, maxind.w4
6.4.4.5     VMINELt  vqual.r1, terms.vr, minind.w4
6.4.2.9     VMODt  vqual.r1, divd.vr, divsr.vr, mod.vw
6.4.2.1     VMOVEt  vqual.r1, source.vr, dest.vw
6.4.2.4     VMPYt  vqual.r1, facta.vr, factb.vr, prod.vw
6.4.2.6     VNEGt  vqual.r1, source.vr, neg.vw
6.4.3.2     VOR4  vqual.r1, terma.vr, termb.vr, or.vw
6.4.2.8     VREMt  vqual.r1, divd.vr, divsr.vr, rem.vw
6.4.4.11    VSCATt vqual.r1, source.vr, index.vr, destination.vw
6.4.2.3     VSUBt  vqual.r1, terma.vr, termb.vr, diff.vw
6.4.3.3     VXOR4  vqual.r1, terma.vr, termb.vr, xor.vw
6.5.1.1.3   WBYTE  data.r1, end.r1
6.5.1.1.2   WCMD  command.r1
6.5.1.2.7   WDP  channel.r1, data.r16, length.rw1
6.2.3.4     XOR4  mask.r4, operand.rw4
```