HEULETT PACKARD

COMPUTER SYSTEMS - 19447 Pruneridge Ave. Cupertino CA 95014

From: Bert Speelpenning X4133

Date:

February 8, 1983

To: Dick Anderson Alan Christensen Subject: VISION architecture

status (CPU)

Shane Dickey Bob Erickson Bob Frankenberg Bill Gimple

Larry Goldman (IND) Rich Hammons (TCG)

Rich Hammons (IC Carson Kan Leon Leong (IND) Jim Nissen Ed Olander

Elik Porat Howard Smith

Ken Spalding

Dave Salomaki

Cc: Alan Hewer Jim Miller

An updated description of the HP3000 mode of the Vision architecture has now been released, through the efforts of Terry Jackson. For questions or comments, please refer to Terry.

The following issues have been resolved, clarified or addressed since the previous Vision CPU architecture memo.

1. Mode switch

We have identified several ways to shave time from the mode switch operations between Vision mode and HP3000 mode. This is clearly important for the performance of HPE.

a) switch marker

Switching modes is not a truly asynchronous event like an external interrupt. This allows us to get by with saving only a subset of the register values, under software control. This means that fewer pushes and pops are required to do the mode switch. In order to accomplish this we need to distinguish between a switch marker and a full blown interrupt marker. We also need to give IEXIT the means to distinguish between the two markers.

Basically, a switch marker will be an external procedure marker with STATUSB added; an interrupt marker is then a switch marker with X0-X15 and B0-B5 added. An additional bit in the TCB, called RSWIP (return-switch in progress) will keep the IEXII logic straight. Updated ACD pages are provided.

b) switch entry point

A dedicated object in group zero (see under 11) will provide the entry point for the switch software in native mode. The switch operation is no longer regarded as a trap. It will have no parameters. This will save some execution time at the expense of some replicated code.

2. Nil object specifications

The Nil object is further defined (relative to the ACD version 5) to guarantee that all implementations cause consistent traps to occur when attempting to access memory through the nil pointer. The full statement is that operating system software shall set the OD of object zero in group zero to values that correspond to an object type of data, access rights R3W3, a lower bound of 1 and an upper bound of 0. The virtual object number of the nil object shall also be fixed at zero.

3. SIT trap

The SIT trap (DBSIT) will report the pointer to the next instruction as its parameter rather than the previous instruction address. The obvious hardware implementation for SIT uses similar logic as for external interrupts and it is unnecessarily expensive for hardware to hang on to the previous instruction counter. We don't expect this to create any difficulty for software.

4. Access rights checking

Both VCF50 and VCF50 teams have requested reconsideration of certain aspects of the access rights checking rules.

a) write access to imply read access

Hardware simplifications accrue (and redesign can be avoided) if write access to a data object always implies read access. The access right fields in an Object Descriptor keep their original meaning; we plan to merely add a statement that operating system software shall not create objects that have write access without also granting them read access.

b) read access to the current code object

The ACD makes a distinction between read access to the current code object and read access to the same object when it does not happen to be the current code object. It does this by stating that read access to the current code object is always granted regardless of the contents of the Object Descriptor for the code object.

Currently, this can be implemented on the VCF60 and the VCF50 only by doing extra work in CALLX and EXIT, which will slow these important instructions down.

We therefore feel that very strong reasons are needed to retain this exceptional treatment of the current code object in the Vision architecture. If you feel you have such strong reasons, we would like to hear them by Feb 15. On the HP3000, P-relative addressing of data is a basic addressing mode and the only one available to offer protection to third-party software. On Vision, no performance benefits derive from keeping data in your code segment rather than in some data segment, and third-party software can be protected by separate privilege level and by exploiting the group structure.

5. Interruptible instructions

Questions have been raised regarding the expected behavior when an interruptible instruction is resumed and finds that its data on the stack has been corrupted. In particular, what should happen when the IIP bit is set but the word popped from the stack (which represents how many times around the loop have already been performed) is found to be negative? The expected behavior in this and similar cases is allowed to be implementation dependent, as long as the "damage" does not extend to another task. For example, it is acceptable to immediately continue to the next instruction when this happens; it is not acceptable to hang in an infinite microcode loop.

6. "Overlap"

The notion of overlap between source and destination of an instruction needs some revision to get around some nasty microcode implications. An instruction such as

MOVE8 source, destination

is only guaranteed to obtain the expected result when the destination does not "overlap" the source. This is to allow hardware to do the move in either one 64-bit gulp or two 32-bit gulp or (probably in case of misaligments) in some number of odd-sized gulps, and yet be able to recover from a page fault in the middle of the move. The exclusion of overlap makes it permissible to restart the instruction even if the destination had been partially modified.

For this to <u>really</u> work, we need to extend the notion of overlap to cover the case exemplified by MOVES [B5+X6], X6. If MOVES encounters a page fault at [B5+X6+1], the value of X6 may already have been modified to the value at [B5+X6]. To avoid this, the definition of "overlap" must incorporate the components of an address calculation for the source operands.

7. Code object size

A Vision mode code object is limited in size to 2^24 bytes. This is not currently stated explicitly in the ACD but could be assumed from the format of the external procedure marker. We now make this assumption explicit.

8. DST and CST descriptors

We will extend the MOVEfSP8 instruction to allow software to get access to the current values of the CST and DST descriptors.

9. Bounds checking on variable length instructions

Some instructions such as MOVEC and CMPC involve a sequence of byte operations over a length given in the instruction. The way bounds checking is performed optimally in such an instruction depends on the organization of the hardware. On the VCF60, bounds checking is performed in parallel with an actual access. On the VCF50, bounds checking is done explicitly in microcode As a consequence, on the VCF60 it is fastest to start up the loop of MOVEC or CMPC and trap out when the end of the object is reached before the loop counter is exhausted. In contrast, on the VCF50 it is fastest to check whether both first and last byte are within bounds and not do any bounds checking for intermediate bytes once the loop starts.

The issue then arises when and how a bounds violation must be

reported and how much of the instruction should be presumed to have been completed when this occurs.

We have decided that hardware should be left free to choose the sequence that is optimal for it. The definition for MOVEC will now state that if MOVEC cannot be completed due to a bounds violation, the effect of MOVEC is that a contiguous but unspecified number of bytes has been moved, all within the object's bounds.

A similar modification will serve for CMPC.

10. Page fault trap

The parameters for the page fault trap are currently listed as including an 8-byte Virtual Page Number (left justified) and a 4-byte Page Offset (right justified).

We have collapsed these now to a single 8-byte Virtual Address.

11. Architecturally fixed object numbers

We have received a request from HPE-I to dedicate certain objects in group zero for certain uses and to fix these objects architecturally. In the version 5 ACD, four objects are fixed by their logical address (the NIL object, trap code object, channel interrupt code object and processor interrupt code object) and four are fixed by their virtual address (SYSCOM area, hash table, page directory and PME). We have been asked to extend this list and also to move the logical object numbers for trap code object, etc. downward so that SYSCOM area, etc. can be given a logical object number that is the same as its virtual object number.

We believe that the only object numbers (logical or virtual) that need to be fixed architecturally are those that must be known to both software and microcode. Any other object numbers can be fixed by software convention, not architectural mandate. We are willing to move the logical object numbers for trap code object, etc. downward in order to make it possible for HPE-I to implement the scheme they proposed as a software convention. The revised numbering is shown below. More object numbers will be fixed only after it has been demonstrated that both software and hardware (microcode) are affected.

logical address

NIL object	group 0,	object	0
trap object	group 0,	object	10
channel interrupt object	group 0,	object	11
processor interrupt object	group 0,	object	12
switch handler (nm) object	group 0,	object	13

12. Decimal instructions

We have decided to allow conversions from 64-bit integers to both 8-byte decimal and 16-byte decimal and vice versa. All these instructions will be moved to the CONVERT escape group.

We have also decided to combine the ZEXT3 and TRUNC3 instructions of the previous status memo into a single MOVE3 instruction. We will include a fuller description of these later. An updated opcode chart, though still tentative, is included.

13. IEEE floating point

The Proposed Standard for floating point arithmetic has come one important step closer to becoming the Standard for floating point arithmetic.

In the latest round of balloting some small amendments were passed. We will include a description of this later. In the mean time, please consult Bill Ames.

14. "MEMSAC" instructions

We have decided in principle to adopt the memory diagnostic capabilities proposed by Jim Wichelman and Jim Chiochios. We are in the process of refining all the encodings to assist the hardware in implementing these. The latest iteration is reflected in a memo by Brian Button dated Feb 1.

15. STATUSC and STATUSD

The current definition of STATUSC and STATUSD is based on the difference in behavior of changes in status in a shared-memory multiprocessor system. Items in STATUSC, when changed, do not affect any other processor in the system; whereas changes to STATUSD must be propagated to all other processors in the sharedmemory multiprocessor system.

We are currently investigating whether the responsibility for notifying other processors can be relegated to system software; this would make the multiprocessor implementation potentially simpler, faster and more reliable.

Until this investigation is complete, we will hold off on other changes to STATUSC and STATUSD that have been proposed, such as removing the mode bit from STATUSC.

It is quite possible that the eventual result will be to move all items of STATUSC and STATUSD into the SYSCOM area.

16. PROBE and BPROBE.

PROBE is intended for use by system intrinsics to allow them to test whether the caller has passed a legal address (range) to the intrinsic.

Jim Miller has made a proposal, with Alan Hewer, for a change in the definition of PROBE and for a new variant (BPROBE) of PROBE that takes the address to be probed from a base register rather than from memory. The change in PROBE closes a protection hole having to do with the fact that the value of S in the environment of the procedure doing the PROBE is larger than the value of S that applied in the environment of the caller. The variant BPROBE would help make passing address parameters in base registers more effective. Updated ACD pages for these two instructions are provided.

Note that the encodings for "ring" have been changed as well.

17. CHECKA and CHECKB.

Currently, the definition of CHECKA, B includes a special way of treating the operand of the instruction. If the bit CBA or CBB is not set, the specification prohibits trapping on an illegal operand. This was done so that implementations could implement CHECKA, B without having to do an operand fetch; this could speed up the (frequent) case where CBA, CBB is clear at the expense of the case where the bit is set. However, it turns out that in many situations the operand fetch does not slow down execution and the special prohibition on operand traps incurs a cost simply because it involves a special case.

In retrospect, it is therefore clear that CHECKA and CHECKB were overspecified. A better statement is that CHECKA and CHECKB are not required to trap an operand violation if CBA,CBB are clear.

18. Ring level for code running on the ICS

We are looking into the possibility of relegating more code that must run on the ICS to ring level 1 rather than level 0. This would allow better granularity on protection in system code. It would also allow the trap object to run at level 1. Changes to IEXIT would be required to allow it to exit into code that runs at a higher level.

We hope to have a proposal by next month.

ERRATA

- all references to "time of day" should be replaced with "time of century"
- MOVEtSP4: setting CBA and CBB does not require special privilege. Privilege level 3 is sufficient.
 MOVEfSP4: CBA and CBB do not warrant their own selector.
 Accessing CBA and CBB must now be done using MOVEfSP STATUSB1.
- The breakrange trap does <u>not</u> return the operand responsible for tripping the breakrange. It is up to software to determine what values changed within the breakrange. It is precisely because it is not very feasible for hardware to keep track of the operand responsible that the VISION architecture has a breakrange for write but not for read and not for execute.
- section 7.1.2 still mentions SI. This is an unintended carry-over from the version 3 ACD. Because of the redefinition of the dispatcher marker, in version 5 QI and SI are one and the same. Hence any reference to SI should be deleted.
- TCBX is no longer architecturally defined. In version 5 the TCB is accessible to software, so a MOVEf/tSP is no longer needed to manipulate a TCBX pointer. The TCB-extension is a software concept only.
- clarification: MOVETSP4 task clock enable has no effect when executed on the ICS.
- corrections and clarifications prove necessary in the definitions of CALLX, EXIT and PDDEL. Updated ACD pages are provided.

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.

OPCOL	DE							
	+!00 _	+!01	+!02	+!03	+!04	+105	+!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	TESTSTRIE	> *	*	*	*	BRX	*	*
!20	*	*	QUAD4	*	POP8	*	*	POP16
!28	PUSH1	PUSH2	PUSH8	*	TESTDOWN	N UP	DOWN	PUSH16
. ! 30	POP1	POP2	*	*	*	TESTREF	*	TEST16D
138	*	TEST2	TEST8	TEST4F	TEST4D	TEST8D	TEST8F	TEST16F
!40	AND4	*	*	MPY4F	MPY8	*	MPY8F	MPY16F
!48	NOT4	*	DIV4	DIV4F	DIV8	×	DIVSF	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	BMOVEADI		*	*	*	*	*
!88	*	*	MOVEfSP4	MOVEfSP8	TESTSEM		SL8D	SL16D
! 90	*	*	MOVEtSP4		MOVESEMA		SR8D	SR16D
! 98	CHECKLO	CHECKHI	DUP	OVPUNCH	WOAE3	CMP4D	CMP8D	CMP16D
! A0	LSL4	ASL4	BCMP4	GETSIGN	ZEXT2	ADD4D	ADD8D	ADD16D
	LSR4	ASR4	BADD4	VALN	*	SUB4D	SUB8D	SUB16D
	LSL8	ASL8	BSUB4	VALD	¥	MPY4D	MPY8D	MPY16D
•	LSR8	ASR8	*	*		DIV4D	DIASD	DIV16D
: C0	PROBE	BPROBE *	MOVEBIT	MOVEC	*	*	MOVEBLR	
!C8	DPF		REP	CMPC	*	TRANSL	MOVEBRL *	CMPT *
! D0	POLY4F	POLY8F	POLY16F	SCANUNTI:	L* *			
!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	ν.	~	π	ERROR

- Note 1: the rows marked with "@" contain the instructions that can be packed two per word.
- Note 2: the instructions VECTOR , SYS and CONVERT are escapes to a secondary set of opcodes.

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";
IIP := 0;
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 Préturn;
(S-4)[0..2] := STATUSA[0..2];
PUSH4 Q[32..63];
Q := S;
if loi non-existent-object then Trap"CODEODTV";
if OD(loi).TYP \leftrightarrow VisionCode then Trap"CODETYPV";
if STATUSA.XL > OD(loi).PR then Trap"CODERINGV";
STATUSA.XL := OD(loi).XL;
Ptarget[ 0..31] := loi;
Ptarget[32..63] := OD(10i).EPWO * 4;
P := Ptarget;
```

Traps: STKOVF CODEODTV CODETYPV CODEBNDSV CODERNGV DBCALL

02/08

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 > ST return.XL then Trap"CODERINGV";
   if Pobject non-existent then Trap"CODEODTV";
   if OD(Pobject).TYPE <> VisionCode then Trap"CODETYPV";
   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;
   enā;
Q offset := (Q-4)[0..31];
i\bar{f} \ Q \ offset < 0 \ or \ Q \ offset > Q[32..63] - 12
then Trap"STKCONSISTV";
if Poffset[31] = 1 and {implementation choice}
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;
            := ST return; {SIT and DBP bits not to take
STATUSA
                           effect until next instruction}
```

Status: restored from marker on external exit

Traps: INSXTL
CODEODTV
CODERINGV
CODETYPV
STKCONSISTV
CODEBNDSV
INSODDP

6.2.4.9 SCANUNTIL limit.r4, 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. Scanning starts at the byte index "index" into the string and will not go beyond "limit". SCANUNTIL sets "index" to the value of the first byte scanned that satisfies the condition if such a byte exists; it leaves "index" at the value of "limit", otherwise. 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 within 32 bytes of the object's upper bound, an addressing violation trap is raised. This instruction must be interruptible.

```
MOVEADR string, St;
if IIP = 0 then C := index
else POP4 C;
IIP := 0:
CC := CCĹ;
Notvetdoné := C <= limit:
while Notvetdone do begin
   Char := (St + C) [0..7]; {zero-extend}
   TESTBIT Char, charset;
   if CC = CCG then begin
      Notyetdone := false;
      index := C;
   end
   else begin
      C := C + 1:
      Notyetdone := C <= limit;
      { if implementation chooses to acknowledge
        an external interrupt here, then
        PUSH4 C; set IIP := 1; Notyetdone := false;
   end;
end;
```

Status: CC

Status: CC

Traps: AddressingV

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.

> Searchpa := PDIR.PA + 16 * ppn + 12; VPN := (PDIR.PA + 16 * ppn + 4)[0..51];
> Linkpa := HASH.PA + 4 * hash(VPN); repeat Oldlinkpa := Linkpa; if (Linkpa)[0..31] = 0 then Trap"ADRPDIR"; Linkpa := (Linkpa)[0..31] + 12; until Linkpa = Searchpa; (01dlinkpa)[0..31] := (Searchpa)[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

6,2,6,10 CHECKA parameter.r4

Conditional break. If the "CBA" enable bit is set, a trap is taken. The value of "parameter" is passed to the trap handler. It is permissible for hardware to not trap on an illegal operand if CBA is clear.

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

Traps: DBCHECKA

6.2.6.11 CHECKB parameter.r4

Conditional break. If the "CBB" enable bit is set, a trap is taken. The value of "parameter" is passed to the trap handler. It is permissible for hardware to not trap on an illegal operand if CBB is clear.

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

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.

/TCB.LA \TCB.VA		0 1 2 3 31 XM SWIP RSWIP reserved
	+4	for hardware
	+12	reserved for system software
	+16 +20 +24 +28	GD1 group descriptor for group 1
	+32 : : +108	
	+112 +116 +120 +124	 GD7 group descriptor for group 7
	+128 +132 +136	Task Breakrange Descriptor

+140 | +144 |

TCB. VA +144 SC - HP3000 mode +148 Stack Pointer > HP3000 mode +152 CSTX information +156 descriptor SN - Vision mode +160 +164 Stack Pointer | logobjid of VCSA > Vision mode | information +172 TRYOFFSET +176

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

SWIP -- switch in progress. Used by IEXIT.

RSWIP -- return switch in progress. Used by IEXIT.

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.

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, a switch marker or 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 <u>stack</u> 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 3 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 left pending, or the dispatcher.

The IEXIT description uses these uninterruptible sequences:

```
RESTORE RETURN(Bregs):
                        begin
                            if (TCB.RSWIP = 0) or Bregs then begin
                               BPOP8 B5; .. BPOP8 B0;
                               POP4 X15; .. POP4 X0;
                               end;
                            POP8 STATUSB; TCB.RSWIP := 0;
                            Q := S; EXIT;
                         begin 'POP2' DelQ; Q := S - DelQ;
RESTORE HP3000:
                                             `POP2' Z.OFFSÉT:
                            POP8' STATUSB:
                            'POP2' DL.OFFSÉT; 'POP2' DB.OFFSET;
                            'POP2' DB.DST;
                            end
                            6-65
```

```
IEXIT: if STATUSC.ICS = 0 then Trap"INSPRIV";
        if Q = QI and not(todispatch) then begin
           {return to task}
case 1:
           STATUSC.ICS := 0; XM := TCB.XM;
           if XM = 0 then begin
              {return to Vision mode}
              S := TCB.SN[0..63];
              if TCB.SWIP = 0 then RESTORE RETURN(false)
              else begin
                 TCB.SWIP := 0;
                 BRX switch handler; {object 13}
           else begin
              {return to HP3000 mode}
              S := TCB.SC[0..63];
              RESTORE HP3000:
                                                \ don't allow
              if TCB. \overline{SWIP} = 0 then `EXIT 0'
                                               / interrupts
              else P := "SWITCHC" trap label;
              TCB.SWIP := 0;
              end
           end
        else if Q=QI or (todispatch and (Q)[4]=1) then begin
           {start dispatcher}
case 2:
           Q := QI; DRF := 0;
           STATUSB := DispatcherStatusBInit;
           EXIT <<but leave S at Q>> {Q doesn't change}
        else
case 3:
           {resume code running before interrupted}
           S := Q + 120;
           RESTORE RETURN(true);
```

- Note 1: implementations may substitute for the test Q = QI the test (Q-4)[0..31] = QI[32..63].
- Note 2: "todispatch" summarizes the condition that dispatching is both desired (DRF=1) and possible (DDC=0, IE=1).

Status: restored from marker
Traps: INSPRIV
STKUNF
STKCONSISTV
SWITCHC

AddressingV on all base register loads

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 a Switch 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.

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 switch 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 STATUSB.IE = 0
then Trap"INSSWITCH"
else
  begin
  S := Q+8;
  TCB.SN := S;
  TCB.XM := 1;
  TCB.RSWIP := 1;
  execute_case_1_of_IEXIT;
  end;
```

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 task clock enable	1	1	GetTCE
13 STATUSC	32	1	GetSTATC
14 Interrupt Mask	16	1	GetIMR
15 STATUSD	32	1	GetSTATD
16 HASH.PA	32	1	
17 HASH, LENGTH	32	1	
18 PDIR.PA	32	. 1	
19 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	3	SetCBA
11 cond break B	1	3	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

SELECTORV INSPRIV

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 "SELECTORV" to be raised.

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

Traps: SELECTORV INSPRIV

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 interval timer	64	0	
1 task clock	64	Ō	
2 time of century	64	0	
3 QI.LA	64	0	
4 DST descriptor	64	0	
5 CST descriptor	64	0	

Traps: dependent on selector

SELECTORV

INSPRIV

6-54

6.2.8 Instructions that interact with the address space

6.2.8.1 PROBE ring_access.r1, address.m, length.r4, s_upper.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. A negative "length" is considered illegal; a zero length represents the case where the address range will not be used, yet the address may have to be loaded into a base register. If the object is the stack object, then the ending address is compared against "s upper" instead of the present value of S or SL. PROBE requires ring 1 privilege.

> ring := ring access[0..3]: access := ring access[4..7];

Encodings:	ring	access
========	====	=====
0	0	instruction fetch
1	1	memory read
2	2	memory write
3	3	
4	caller's	

Values not in this list will cause a SELECTORV trap.

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

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; or: the object exists, the access is legal and the length is zero.

Status: CC Traps: INSPRIV

SELECTORY

VISION ARCHITECTURE CONTROL DOCUMENT DO NOT COPY -- HP PRIVATE INFORMATION 02/08

6.2.8.2 BPROBE ring_access.r1, address.b, length.r4, s_upper.r4

Probe access rights. This instruction sets condition codes dependent on the legality of accessing the address range given by "address" and "length".

This instruction differs from PROBE only in that the address is already loaded into a base register. This implies that the object is already known to exist.

BPROBE requires level 1 privilege.

Status: CC

Traps: SELECTORV

VCF 60 SPU BLOCK DIAGRAM

