Intel® Itanium® Architecture Software Developer's Manual Specification Update

July 2005

Notice: Intel® Itanium® architecture processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are documented in processor specification updates.

Document Number: 248699-010

Contents

Revision History

Version Number	Description	Date
-010	Added Specification Changes 10-17; added Specification Clarifications 23-30; added Document Change 14; updated Document Changes 1 and 13.	July 2005
-009	Added Specification Changes 5-9; added Specification Clarifications 16-22; added Documentation Changes 8-13.	August 2004
-008	Added Specification Changes 2-4; added Specification Clarifications 5-15; added Documentation Changes 1-7.	October 2003
-007	Added Specification Change 1; added Specification Clarification 1-4.	December 2002
-001- -006	Changes from previous Software Developer's Manual Specification Updates were incorporated into version 2.1 of the *Intel® Itanium® Architecture Software Developer's Manual* October 2002.	June 2002

Preface

This document is an update to the specifications contained in the Affected Documents/Related Documents table below. This document is a compilation of device and documentation errata, specification clarifications, and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Information types defined in Nomenclature are consolidated into the specification update and are no longer published in other documents.

This document may also contain information that was not previously published.

Affected Documents/Related Documents

Title	Document #
IIntel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture	245317-004
Intel® Itanium® Architecture Software Developer's Manual, Volume 2: System Architecture	245318-004
Intel® Itanium® Architecture Software Developer's Manual, Volume 3: Instruction Set Reference	245319-004

Nomenclature

Specification Changes are modifications to the current published specifications for Intel® Itanium® architecture processors. These changes will be incorporated in the next release of the specifications.

Specification Clarifications describe a specification in greater detail or further explain a specification's interpretation. These clarifications will be incorporated in the next release of the specification.

Documentation Changes include typos, errors, or omissions from the current published specifications. These changes will be incorporated in the next release of the *Intel® Itanium® Architecture Software Developer's Manual*.

Summary Table of Changes

The following tables indicate the specification changes, specification clarifications, or documentation changes that apply to the *Intel® Itanium® Architecture Software Developer's Manual*.

Specification Changes

No.	Page	SPECIFICATION CHANGES
1	9	Volume 1: ao bit added to CPUID Register 4
2	9	MCA architecture extensions for supporting data-poisoning events
3	12	LID enhancements
4	12	Extend PALE_CHECK exit options
5	13	Addition of tf instruction
6	17	Removal of requirement for externally connected pins
7	20	Architecture extensions for processor Power/Performance states
8	34	Allow undefined behavior for all must-be-last instructions
9	35	Addition of PAL_BRAND_INFO
10	37	Addition of PAL_MC_ERROR_INJECT
11	48	Addition of PAL_GET_HW_POLICY and PAL_SET_HW_POLICY
12	52	PAL_GET_PSTATE modification
13	53	PAL calling convention change to preserve floating-point registers
14	53	Addition of shared error (se) bit to the processor state parameter (PSP)
15	54	Enhancement to PAL_LOGICAL_TO_PHYSICAL
16	54	Change of maximum alignment requirements for PAL_COPY_INFO
17	54	Indication of variable P-states

Specification Clarifications (Sheet 1 of 2)

No.	Page	SPECIFICATION CLARIFICATIONS
1	56	Volume 2: PSR.dt serialization clarification
2	9	Volume 2: Unaligned debug fault clarification
3	12	Volume 3: Clarification on PSR requirements for br.ia/rfi instructions during PSR.is transition
4	12	Volume 3: Added Illegal Operation fault to fnma I-page
5	13	Clarify INTA/XTP definition
6	17	Clarify VHPT insert rules
7	20	Adding FP-readers to support table
8	34	cmpxchg clarifications
9	35	Add Illegal Operation fault
10	37	Non-speculative reference for WBL attribute clarification
11	48	Dirty-bit fault ISR.code clarification
12	52	FC data dependency ordering clarification
13	53	PAL_MC_DRAIN clarification
14	53	Add hint instructions to support table

Specification Clarifications (Sheet 2 of 2)

No.	Page	SPECIFICATION CLARIFICATIONS
15	54	Clarify speculative operation fault handler requirements
16	54	Clarify role of PMC.ev bit as implementation-specific
17	54	Relax IA-32 Application Registers Reserved/Ignored checking
18	67	Relax ordering constraints for VHPT walks
19	67	Clarify illegal operation fault behavior for predicated off reserved ops
20	68	Clarify opcode hint fields in encodings
21	69	Clarify speculative operation fault handler requirements
22	70	PAL_CACHE_FLUSH clarification
23	70	Interruption serialization clarification
24	71	Clarification of Local ID fields
25	71	PSR.ri field clarification
26	71	Missing instructions in the Ordering Semantics table
27	71	ttag clarification
28	71	Missing fault for mov cr
29	71	Clarifications to tak instruction
30	72	Clarification for the dirty bit during TLB insertion

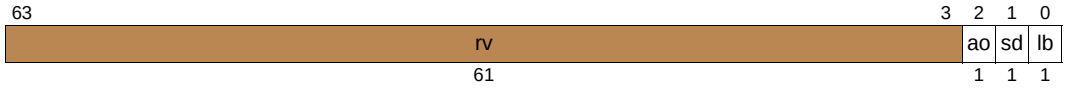
Documentation Changes

No.	Page	DOCUMENTATION CHANGES
1	73	Update IA-32 CPUID I-Page
2	79	PAL_BUS_GET/SET_FEATURES fix
3	79	PAL_COPY_PAL update
4	80	Fixing X-Unit text correction
5	80	PAL_CACHE_SHARED_INFO text correction
6	80	PAL_CACHE_FLUSH clarification and minor code sequence fix
7	80	PAL_GET_PROC_FEATURES table fix
8	80	Correct the role of X-resources during MCA
9	81	Clarification on the short format VHPT
10	81	Floating-point correction
11	81	Clarify effect of sending IPI to non-existent processor
12	81	Add a new instruction class
13	81	Updated RAW Dependence Table
14	90	ISR.ir typo

Specification Changes

1. Volume 1: ao bit added to CPUID Register 4

1. New Figure 3-12 (page 1:30) - added a new bit for ao:



2. Table 3-8 (page 1:30) has a new entry for ao:

Field	Bits	Description
lb	0	Processor implements the long branch (brl) instructions.
sd	1	Processor implements spontaneous deferral (see Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:88).
ao	2	Processor implements 16-byte atomic operations (see “ld — Load”, “st — Store” and “cmpxchg — Compare and Exchange” instructions in Volume 3).
rv	63:3	Reserved.

2. MCA architecture extensions for supporting data-poisoning events

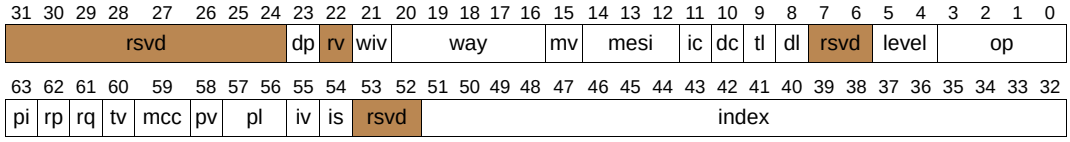
1. Volume 2: Added the following row to Table 11-54 (page 2:360) of PAL_PROC_GET_FEATURES:

Table 11-54. Processor Features

Bit	Class	Control	Description
53	Opt.	Req.	Enable MCA signaling on data-poisoning event detection. When 0, a CMCI will be signaled on error detection. When 1, an MCA will be signaled on error detection. If this feature is not supported, then the corresponding argument is ignored when calling PAL_PROC_SET_FEATURES. Note that the functionality of this bit is independent of the setting in bit 60 (Enable CMCI promotion), and that the bit 60 setting does not affect CMCI signaling for data-poisoning related events.

2. Volume 2: dp bit added to PAL_MC_ERROR_INFO Cache_Check and Bus_Check:
   a. Figure 11-37 (page 2:345) – added new bit for dp:

Figure 11-37. cache_check Layout



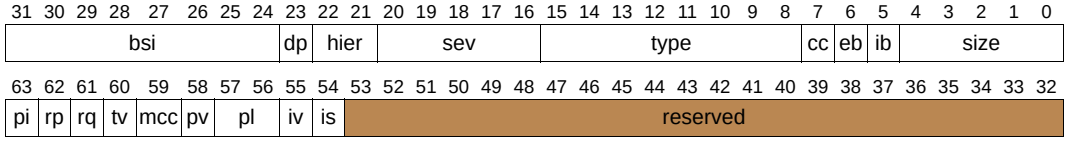
b. Table 11-47 (page 2:345) – added new entry for dp:

Table 11-47. cache_check Fields

Field	Bits	Description
op	3:0	Type of cache operation that caused the machine check: 0 – unknown or internal error 1 – load 2 – store 3 – instruction fetch or instruction prefetch 4 – data prefetch (both hardware and software) 5 – snoop (coherency check) 6 – cast out (explicit or implicit write-back of a cache line) 7 – move in (cache line fill) All other values are reserved.
level	5:4	Level of cache where the error occurred. A value of 0 indicates the first level of cache.
rsvd	7:6	Reserved
dl	8	Failure located in the data part of the cache line.
tl	9	Failure located in the tag part of the cache line.
dc	10	Failure located in the data cache
ic	11	Failure located in the instruction cache
mesi	14:12	0 – cache line is invalid. 1 – cache line is held shared. 2 – cache line is held exclusive. 3 – cache line is modified. All other values are reserved.
mv	15	The *mesi* field in the cache_check parameter is valid.
way	20:16	Failure located in the way of the cache indicated by this value.
wiv	21	The *way* and *index* field in the cache_check parameter is valid.
rsvd	22	Reserved
dp	23	A multiple-bit error was detected, and data was poisoned for the corresponding cache line during castout.
rsvd	31:24	Reserved
index	51:32	Index of the cache line where the error occurred.
rsvd	53:52	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the cache_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the cache_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

c. Figure 11-39 (page 2:347) – added new bit for dp:

Figure 11-39. bus_check Layout



d. Table 11-49 (page 2:347) – added new entry at bit 23 for dp:

Table 11-49. bus_check Fields

Field	Bits	Description
size	4:0	Size in bytes of the transaction that caused the machine check abort.
ib	5	Internal bus error
eb	6	External bus error
cc	7	Error occurred during a cache to cache transfer.
type	15:8	Type of transaction that caused the machine check abort. 0 – unknown 1 – partial read 2 – partial write 3 – full line read 4 – full line write 5 – implicit or explicit write-back operation 6 – snoop probe 7 – incoming or outgoing ptc.g 8 – write coalescing transactions 9 – I/O space read 10 – I/O space write 11 – inter-processor interrupt message (IPI) 12 – interrupt acknowledge or external task priority cycle All other values are reserved
sev	20:16	Bus error severity. The encodings of error severity are platform specific.
hier	22:21	This value indicates which level or bus hierarchy the error occurred in. A value of 0 indicates the first level of hierarchy.
dp	23	A multiple-bit error was detected, and data was poisoned for the incoming cache line.
bsi	31:24	Bus error status information. It describes the type of bus error. This field is processor bus specific.
reserved	53:32	Reserved
is	54	Instruction set. If this value is set to zero, the instruction that generated the machine check was an Intel Itanium instruction. If this bit is set to one, the instruction that generated the machine check was IA-32 instruction.
iv	55	The *is* field in the bus_check parameter is valid.
pl	57:56	Privilege level. The privilege level of the instruction bundle responsible for generating the machine check.
pv	58	The *pl* field of the bus_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
tv	60	Target address is valid: This bit is set to one to indicate that a valid target address has been logged.
rq	61	Requester identifier: This bit is set to one to indicate that a valid requester identifier has been logged.

Table 11-49. bus_check Fields (Continued)

Field	Bits	Description
rp	62	Responder identifier: This bit is set to one to indicate that a valid responder identifier has been logged.
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

3. LID enhancements

1. Volume 2, Part I, Section 5.8.3.1, page 2:104, first paragraph should now read:

   "The LID register contains the processor's local interrupt identifier. Two fields (id and eid) serve as the processor's physical name for all interrupt messages (external interrupts, INITs, and PMIs). LID is loaded by firmware during platform initialization based on the processor's physical location within the system. Processors receiving an interrupt message on the system interconnect may or may not compare their id/eid fields with the target address for the interrupt message, depending on the type of system interconnect. If this comparison is performed, then a match would indicate that the interrupt received was intended for this processor. In case of no comparison, processors use other system topology mechanisms to determine the correct target of the interrupt message."

2. Volume 2, Part I, Section 5.8.3.1, page 2:104, second paragraph, change from:

   "LID is a read-write register."

   to:

   "The LID register fields are read-only or read-write. Details of the programmability of these fields is communicated by PAL at PALE_RESET handoff (see Section 11.2.2: 'PALE_RESET Exit State' for details). Read-only LID bits always return a value of 0. Writes to read-only bits are ignored."

3. Volume 2, Part I, Section 11.2.2, page 2:259, change the GR33 bullet from:

   "GR33 contains the geographically significant unique processor ID. The value is the same as that returned by PAL_FIXED_ADDR"

   to:

   "GR33 contains information about the geographically significant unique processor ID, and a mask that indicates which bits in the LID register (CR64) are read-only. Firmware should write the processor's local interrupt identifier in the programmable portion of the LID register. Writes to the read-only bits are ignored.

   [63:48] Reserved

   [47:40] Mask indicating which bits in eid are programmable

   0 = programmable, 1 = read-only

   [39:32] Mask indicating which bits in id are programmable

   0 = programmable, 1 = read-only

   [31:16] Reserved

   [15:0] Geographically significant processor ID

   The value returned in bits [15:0] is the same as that returned by PAL_FIXED_ADDR."

4. Extend PALE_CHECK exit options

1. Volume 2, Part I, Section 11.3.1:

   a. On page 2:265, first paragraph, change the following sentence from:

      "PALE_CHECK terminates by branching to SALE_ENTRY, passing the state of the processor at the time of the error."

      to:

"PALE_CHECK terminates either by returning to the interrupted context or by branching to SALE_ENTRY, passing the state of the processor at the time of the error."

b. In the fifth paragraph change the following from:

"PSR.mc is set to 1 by the hardware when PALE_CHECK is entered. PSR.mc will remain set for the duration of PALE_CHECK, and PALE_CHECK will exit with psr.mc set."

to:

"PSR.mc is set to 1 by the hardware when PALE_CHECK is entered. When PALE_CHECK branches to SALE_ENTRY, PSR.mc remains set (PSR.mc is restored to its original value if PALE_CHECK terminates by returning to the interrupted context)."

And delete: "PALE_CHECK must attempt to branch to SALE_ENTRY unless code execution is not possible."

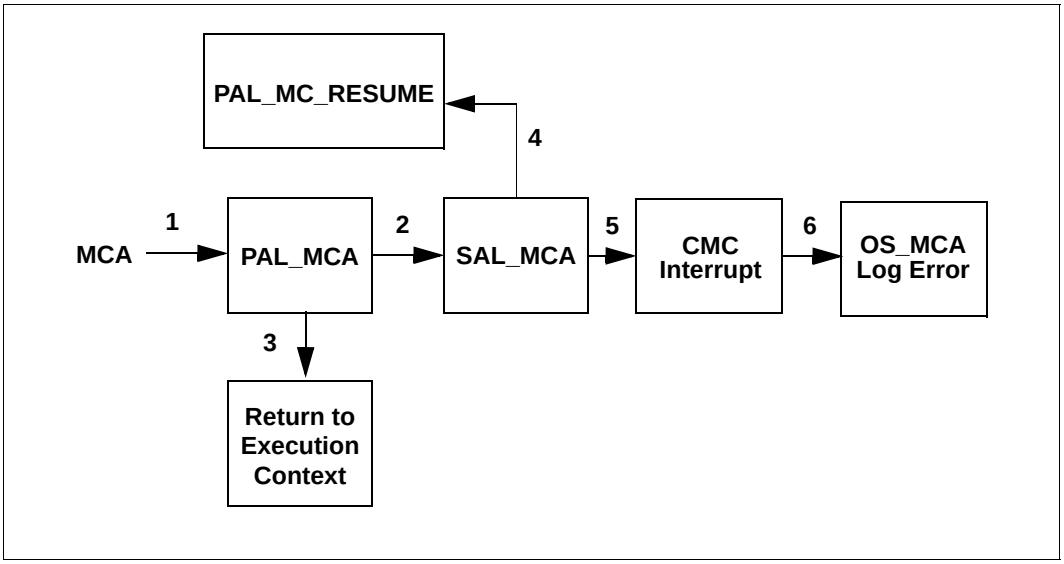
2. Volume 2, Part II, Section 13.3.1, page 2:493:

a. The second paragraph should read:

When the processor detects an error, control is transferred to the PAL_MCA entrypoint. PAL_MCA will perform error analysis and processor error correction where possible. Subsequently, PAL either returns to the interrupted context or hands off control to the SAL_MCA component. The level of recovery provided by PAL_MCA is implementation dependent and is beyond the scope of this specification. SAL_MCA will perform error logging and platform error correction where possible. Errors that are corrected by PAL and SAL firmware are logged and control is transferred back to the interrupted process/context. For corrected errors, no OS intervention is required for error handling, but the OS is notified of the event for logging purposes through a low priority asynchronous corrected machine check interrupt (CMCI). See Section 5.8.3.8, "Corrected Machine Check Vector (CMCV – CR74)" for more information on the CMCI. If the error was not corrected by firmware, SAL hands off control to the OS_MCA handler.

b. Added correctable machine check flow:

Figure 13-3. Correctable Machine Check Code Flow



5. Addition of `tf` instruction

1. Volume 3: Added `tf` I-page.

tf — Test Feature

Format: (*qp*) tf.*trel*.*ctype* $p_1, p_2 = imm_5$ I30

Description: The imm_5 value (in the range of 32-63) selects the feature bit defined in Table 2-56 to be tested from the features vector in CPUID[4]. See Section 3.1.11, "Processor Identification Registers" on page 33 for details on CPUID registers. The selected bit forms a single-bit result either complemented or not depending on the *trel* completer. This result is written to the two predicate register destinations p_1 and p_2. The way the result is written to the destinations is determined by the compare type specified by *ctype*. See the Compare instruction and Table 2-15 on page 3:38.

The *trel* completer values.nz and.z indicate non-zero and zero sense of the test. For normal and unc types, only the.z value is directly implemented in hardware; the.nz value is actually a pseudo-op. For it, the assembler simply switches the predicate target specifiers and uses the implemented relation. For the parallel types, both relations are implemented in hardware.

Table 2-54. Test Feature Relations for Normal and unc tf

trel	Test Relation	Pseudo-op of	
nz	selected feature available	z	$p_1 \leftrightarrow p_2$
z	selected feature unavailable		

Table 2-55. Test Feature Relations for Parallel tf

trel	Test Relation
nz	selected feature available
z	selected feature unavailable

If the two predicate register destinations are the same (p_1 and p_2 specify the same predicate register), the instruction will take an Illegal Operation fault, if the qualifying predicate is set or the compare type is unc.

Table 2-56. Test Feature Features Assignment

imm_5	Feature Symbol	Feature
32 - 63	*none*	Not currently defined

Operation:

```
if (PR[qp]) {
    if (p1 == p2)
        illegal_operation_fault();

    tmp_rel = cpuid[4]{imm5};

    if (trel == 'z')                    // 'z' - test for 0, not 1
        tmp_rel =!tmp_rel;

    switch (ctype) {
        case 'and':                     // and-type compare
            if (!tmp_rel) {
                PR[p1] = 0;
                PR[p2] = 0;
            }
            break;
        case 'or':                      // or-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 1;
            }
            break;
        case 'or.andcm':                // or.andcm-type compare
            if (tmp_rel) {
                PR[p1] = 1;
                PR[p2] = 0;
            }
            break;
        case 'unc':                     // unc-type compare
        default:                        // normal compare
            PR[p1] = tmp_rel;
            PR[p2] =!tmp_rel;
            break;
    }
} else {
    if (ctype == 'unc') {
        if (p1 == p2)
            illegal_operation_fault();
        PR[p1] = 0;
        PR[p2] = 0;
    }
}
```

Interruptions: Illegal Operation fault

2. Update Table 4-4 on page 3:258 with the new format:

Test NaT	I17	5	t_b	x_2	t_a	p_2	r_3	x		y	c	p_1	qp
Test Feature	I30	5	t_b	x_2	t_a	p_2	0	x	imm_{5b}	y	c	p_1	qp

3. Updated Section 4.3.3 of Volume 3 on page 3:279:

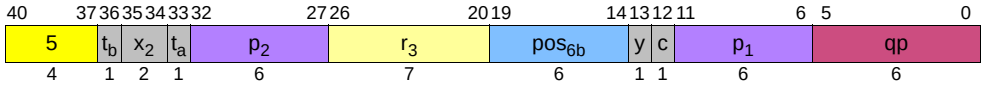
4.3.3 Test Bit

All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) plus five 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), 12 (c), 13 (y) and 19 (x). Table 4-23 summarizes these assignments.

Table 4-23. Test Bit Opcode Extensions

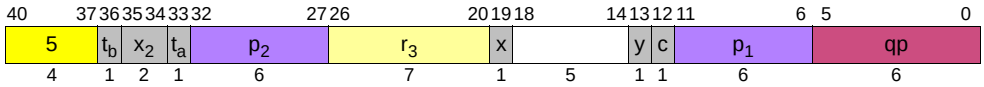
Opcode Bits 40:37	x_2 Bits 35:34	t_a Bit 33	t_b Bit 36	c Bit 12	y Bit 13	x Bit 19	
						0	1
5	0	0	0	0	0	tbit.z l16	
					1	tnat.z l1	tf.z l30
				1	0	tbit.z.unc l16	
					1	tnat.z.unc l1	tf.z.unc l30
			1	0	0	tbit.z.and l16	
					1	tnat.z.and l1	tf.z.and l30
				1	0	tbit.nz.and l16	
					1	tnat.nz.and l1	tf.nz.and l30
		1	0	0	0	tbit.z.or l16	
					1	tnat.z.or l1	tf.z.or l30
				1	0	tbit.nz.or l16	
					1	tnat.nz.or l1	tf.nz.or l30
			1	0	0	tbit.z.or.andcm l16	
					1	tnat.z.or.andcm l1	tf.z.or.andcm l30
				1	0	tbit.nz.or.andcm l16	
					1	tnat.nz.or.andcm l1	tf.nz.or.andcm l30

4.3.3.1 Test Bit



Instruction	Operands	Opcode	Extension				
			x_2	t_a	t_b	y	c
tbit.z	$p_1, p_2 = r_3, pos_6$	5	0	0	0	0	0
tbit.z.unc							1
tbit.z.and					1		0
tbit.nz.and							1
tbit.z.or				1	0		0
tbit.nz.or							1
tbit.z.or.andcm					1		0
tbit.nz.or.andcm							1

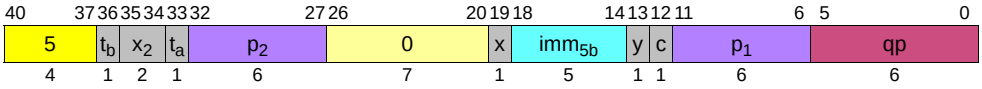
4.3.3.2 Test NaT



Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tnat.z	$p_1, p_2 = r_3$	5	0	0	0	1	0	0
tnat.z.unc								1
tnat.z.and					1			0
tnat.nz.and								1
tnat.z.or				1	0			0
tnat.nz.or								1
tnat.z.or.andcm					1			0
tnat.nz.or.andcm								1

4. New Section 4.3.9, “Test Feature.”

4.3.9 Test Feature



Instruction	Operands	Opcode	Extension					
			x_2	t_a	t_b	y	x	c
tf.z	$p_1, p_2 = imm_5$	5	0	0	0	1	1	0
tf.z.unc								1
tf.z.and					1			0
tf.nz.and								1
tf.z.or				1	0			0
tf.nz.or								1
tf.z.or.andcm					1			0
tf.nz.or.andcm								1

5. In Table 5-5, Volume 3, Section 5.4; add `tf` to “pr-gen-writers-int” and “pr-readers-nobr-nomovpr” entries.
6. Volume 3, Section 4.8; add new `tf` entry after I24 entry in Table 4-74, “Immediate Formation.”

I30	$imm_5 = imm_{5b} + 32$

7. Volume 3, Section 4.3.2, on page 3-283, update Tables 4-21 and 4-22 to reflect new `tf` instruction.
   a. In Table 4-21, change 5:0:0:1 entry from “Test NaT” to “Test Nat/Test Feature.”
   b. In Table 4-22, change 5:0:1:- entry from “Test Bit/Test NaT” to “Test Bit/Test Nat/Test Feature.”

6. Removal of requirement for externally connected pins

1. All references to INIT, PMI, and LINT pins restated to allow for their absence.
   a. Volume 2, Part I, Section 5.8 “Interrupts” on page 2:97, first paragraph, change the following from:

"As shown in Figure 5-3, interrupts are managed by the processor and by one or more intelligent external interrupt controllers or devices in the I/O subsystem."

to:

"Interrupts are managed by the processor and by one or more intelligent external interrupt controllers or devices in the I/O subsystem. Figure 5-3 shows just one example of a high performance interrupt architecture subsystem; other topologies are possible."

b. Volume 2, Part I, Section 5.8, add reference and footnote to LINT, INIT, PMI pins in the locally connected devices bullet.

"Locally connected devices - These interrupts originate on the processor's interrupt pins (LINT, INIT, PMI)1, and are always directed to the local processor. "

In footnote:

"1. Processors are not required to support externally connected interrupt pins. Software can query the presence of the INIT, PMI, and LINT pins via the PAL_PROC_GET_FEATURES procedure call."

c. Volume 2, Part I, Section 5.8.1 "Interrupt Vectors and Priorities", change the second sentence in the second paragraph from:

"Assertion of the processor's PMI pin results in PMI vector number 0."

to:

"Assertion of the processor's PMI pin, when present, results in PMI vector number 0."

d. Volume 2, Part I, Section 5.8.3.9, modify references to LINT pins in the first paragraph from:

"Local Redirection Registers (LRR0-1) steer external signal based interrupts that are directly connected to the local processor to a specific external interrupt vector. All processors support two direct external interrupt pins. These External interrupt signals (pins) are referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1)."

to:

"Local Redirection Registers (LRR0-1) steer external signal based interrupts that are directly connected to the local processor to a specific external interrupt vector. Processors may optionally support two direct external interrupt pins. When supported these external interrupt signals (pins) are referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1). Software can query the presence of these pins via the PAL_PROC_GET_FEATURES procedure call."

e. Volume 2, Part I, Section 5.8.4.2 "Interrupt and IPI Ordering", change the first paragraph from:

"Interrupt messages from external device(s), or external interrupts routed to the processor's LINT pins, may arrive at one or more processors and become pending in any order. No ordering is enforced by the processor or the platform."

to:

"Interrupt messages from external device(s), or external interrupts routed to the processor's LINT pins, when present, may arrive at one or more processors and become pending in any order. No ordering is enforced by the processor or the platform."

f. Volume 2, Part I, Section 5.8.5 "Edge- and Level-sensitive Interrupts", modify the first sentence from:

"The processor's LINT pins directly support edge and level sensitive interrupts, however all other interrupt sources are edge sensitive."

to:

"The processor's LINT pins, when present, directly support edge and level sensitive interrupts, however all other interrupt sources are edge sensitive."

g. Volume 2, Part I, Section 11.5.1 “PMI Overview”, Table 11-10 “PMI Events and Priorities.” Add a footnote reference to the PMI pin row:

PMI Events	Priority
PMI pin (a) (vector 0)	Low

a. PMI pin is not required to be present on all systems.

Also modify this sentence in the fourth paragraph from:

“Vector 0 is used to indicate the PMI pin event.”

to:

“A PMI pin event, when the PMI pin (1) is present, is indicated by vector 0."

Add footnote:

“1. PMI pin is not required to be present. Software can query the presence of PMI pin via the PAL_PROC_GET_FEATURES procedure call.”

h. Volume 2, Part II, Section 10.2 “Configuration of External Interrupt Vectors”, add a footnote reference to the second bullet:

“From the processor's LINT0 or LINT1 pins(1) (typically connected to an Intel 8259A compatible interrupt controller), or”

Add footnote:

“1. Processors optionally support two external interrupt pins. Software can query for the presence of LINT pins via the PAL_PROC_GET_FEATURES procedure call.”

i. Volume 2, Part II, Section 10.5.6 “Local Redirection Example”, add the following note at the end of the section:

“The Local Redirection Registers (LRR0-1) serves to steer external signal based interrupts that are directly connected to the processor. LRR0 and LRR1 control the external interrupt signals (pins) referred to as Local Interrupt 0 (LINT0) and Local Interrupt 1 (LINT1) respectively. The example below shows how to mask interrupt delivery on LINT0.

```
movl r18=(1<<16)
;;
mov cr.lrr0=r18
;;
srlz.d // srlz.d is required after LRR write to ensure write effect.
```

Note: LINT0 and LINT1 pins are not required to be supported. Writes to LRR0-1 control registers would have no effect, and reads from LRR0-1 control registers would return 0."

2. Effects of writes to and reads from control registers LRR 0, 1 defined in the absence of LINT pins:

a. Volume 2, Part 1, Section 5.8.3.9, change the second paragraph from:

“To ensure that subsequent interrupts from LINT0 and LINT1 reflect the new state of LRR prior to a given point in program execution, software must perform a data serialization operation after an LRR write and prior to that point.”

to:

“To ensure that subsequent interrupts from LINT0 and LINT1 reflect the new state of LRR prior to a given point in program execution, software must perform a data serialization operation after an LRR write and prior to that point. In the case when LINT0 and LINT1 pins are absent, writes to LRR would have no effect, and reads from LRR would return 0. Software can query the presence of the LINT pins via the PAL_PROC_GET_FEATURES procedure call.”

3. Detection of INIT, PMI, and LINT pins presence via the PAL_PROC_GET_FEATURES procedure.

   a. Volume 2, Part 1, Chapter 11, Table 11-54, insert new row:

Bit	Class	Control	Description
37	Opt	No	INIT, PMI, and LINT pins present. Denotes the absence of INIT, PMI, LINT0, and LINT1 pins on the processor. When 1, the pins are absent. When 0, the pins are present. This feature may only be interrogated by PAL_PROC_GET_FEATURES. It may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.

   b. Update the last row to:

Bit	Class	Control	Description
36-0	N/A	N/A	Reserved

7. Architecture extensions for processor Power/Performance states

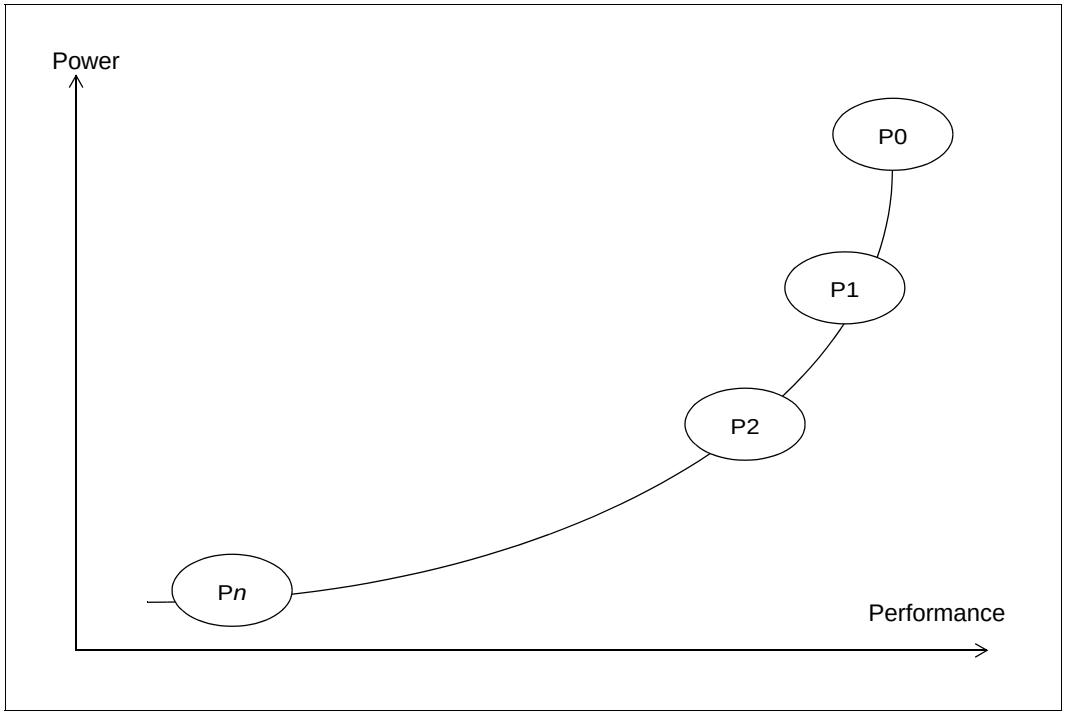
1. New Section 11.6.1 in Volume 2, Part I

11.6.1 Power/Performance States (P-states)

This section describes the power/performance states (hence to be referred as P-states) supported by the Itanium architecture. P-states enable the caller to adjust the power/performance characteristics of the processor in response to changing workload requirements. This allows for implementation of a processor-level power management policy which is driven by system demand and response time requirements.

The P-states are defined within the context of the active/executing processor state. At the highest performing P-state (referred to as the P0 state), the processor uses its maximum performance capability and may consume maximum power. In the next P-state (P1), the processor performance capability is limited below the maximum performance, and it consumes less than the maximum power. Successive P-states continue to have reduced performance capabilities and reduced power consumption than the corresponding lower state. The Itanium architecture supports a maximum of 16 P-states, with the highest numbered P-state that is available on an implementation providing the least possible performance capability and minimal power consumption while remaining in a non-HALT state.

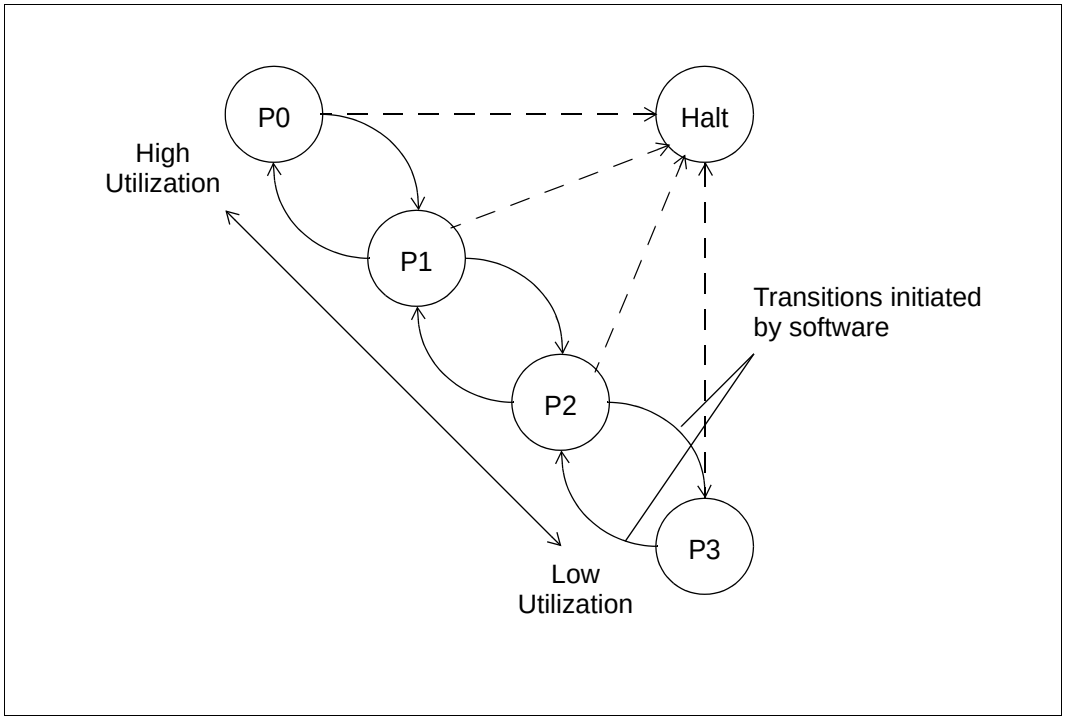
Figure 11-19. Power and Performance Characteristics for P-states



P-states can be utilized by software to implement a demand-based dynamic power management policy where it would continuously try to adapt the processor performance to the current workload characteristics. This allows software to achieve power savings at the system level, while allowing it to quickly respond to changing workload requirements.

The example in Figure 11-20 assumes four P-states (P0, P1, P2 and P3), and a software policy that transitions between the states depending on the current system utilization. During times of high utilization, the software migrates the processor towards lower-numbered P-states, which increases processor performance and increases the dissipated power. When system utilization is low, the software policy migrates the processor towards higher-numbered P-states, thereby reducing the processor performance and reducing dissipated power. The figure also shows the HALT state, which the software can transition to at any time from a given P-state.

Figure 11-20. Example of a P-state Transition Policy



The concept of P-states applies to each logical processor, and this gives software the required granularity to individually control the power/performance characteristics for each available thread of execution in the system. In the most simplistic case, the processor package has only one thread of execution, and this allows software to apply the same P-state policy at the package-level as well as at the logical processor level. However, with implementations that support multithreading and multiple cores, a single package can have multiple logical processors (threads of execution). These may have P-state dependencies among them, which may not allow for individual P-state control flexibility at the software level. For example, these logical processors may be sharing the same clock and power delivery network. In such circumstances, software would need to know which logical processors have dependencies and what the nature of the dependencies is, so that appropriate coordination techniques can be applied. To allow the architecture definition to comprehend for multi-threaded/multi-core designs, we define the concept of dependency domain and coordination mechanisms.

A **dependency domain** is comprised of logical processors that share a common set of implementation-dependant domain parameters that affect power consumption and performance for all logical processors in that domain. As an example, a processor package comprising of two cores controlled by the same clock and power distribution network are part of the same dependency domain, since changing either the operating frequency or voltage will affect power consumption and performance for both cores. Alternatively, if these two cores on the processor package had independent distribution networks for clocks and power, then a change in the parameters for one core would not have any effect on the other core, and in that case, the cores would not belong to the same dependency domain. Software can utilize P-states to affect changes in the domain parameters. Each

P-state maps to a set of values for the domain parameters, and hence a P-state transition results in a change in the underlying power/performance characteristics for the logical processor.

The Itanium architecture supports different types of dependency domains, which enables software to have different degrees of control for P-state changes affecting logical processors in the domain.

A **software-coordinated dependency domain** relies on the software to coordinate P-state changes among the processors in that dependency domain. Software will have knowledge about logical processors belonging to that domain, and will decide when it is appropriate to request the P-state transition. The software policy has to be aware that a P-state change on any logical processor will change the P-state for all logical processors in that domain. As an example, let us assume that the software-coordinated dependency domain consisted of two cores with the same clock and power distribution networks and the intent of the software policy was to lower power/performance only when the workload utilization was low on both cores. Software could then monitor utilization on both cores, and when both cores were under-utilized (i.e., were running at a higher performance P-state than required by the current system demand), it could migrate one of the cores to a lower performance P-state. This transition would simultaneously reduce performance and power dissipation for both cores, and would result in both cores operating at the same P-state.

A **hardware-coordinated dependency domain** relies on hardware-based mechanisms to synchronize P-state changes. Software can make independent P-state change requests on individual processors, recognizing that hardware is responsible for the required coordination with other processors in the same hardware-coordinated dependency domain. Hardware-based coordination mechanisms would be implemented to allow for changes to the logical processor's power and performance local parameters (which are implementation-dependant), in addition to the existing domain parameters. Hardware would use a combination of changes to both of these parameters to satisfy the software-initiated P-state change request. This type of coordination mechanism is effective when it is desired to have individual control over all logical processors, and when the hardware has local parameters for power/performance at the logical processor level. The local parameters allow for fine-grained control (affecting only the logical processor power/performance), whereas the domain parameters allow for coarse-grained control (affecting all logical processors). As an example, let us assume that the hardware-coordinated dependency domain consisted of two cores with the same clock and power distribution networks, and that there were also some other techniques to affect power and performance which were local to each logical processor. When software initiates a P-state transition on the first core, hardware would use only the local parameters to carry out the request. When software requests the same P-state change on the second core, then hardware can undo the changes to the local parameters for the first core, and then initiate changes to the domain parameters, which would allow both cores to operate at the same P-state.

A **hardware-independent dependency domain** is a self-contained domain that typically means that every logical processor is the only logical processor in that domain, and its domain parameters are individually controllable. Since there are no dependencies with any other logical processors, there is no P-state coordination needed for such domains. Software can make P-state change requests independently on that logical processor.

The PAL procedure PAL_PROC_GET_FEATURES returns whether an implementation supports P-states. If an implementation supports P-states then the PAL_PROC_SET_FEATURE procedure will allow the caller to enable or disable this feature.

The Itanium architecture provides three new PAL procedures to enable P-state functionality.

PAL_PSTATE_INFO: This procedure returns information about the P-states implemented on a particular processor. For details on the information returned by this procedure, please refer to the procedure description on page 2:361. The Itanium architecture supports a maximum of 16 P-states.

PAL_SET_PSTATE: This procedure allows the caller to request the transition of the processor to a new P-state. The procedure can either return with transition success (request was accepted) or transition failure (request was not accepted) depending on hardware capabilities, implementation-specific event conditions, and the spacing between successive PAL_SET_PSTATE procedure calls.

If hardware has the ability to either preempt a previous in-progress P-state transition, or to queue successive P-state requests while the first request is in transition, then the implementation has a preemptive policy for P-state request handling. The architecture also allows for a non-preemptive policy for P-state request handling, whereby a new PAL_SET_PSTATE request is not accepted if a previous P-state transition is already in progress. The PAL_SET_PSTATE procedure returns different status values corresponding to the accepted and not accepted cases for P-state requests. If the transition is not accepted, no P-state transition is initiated by the PAL_SET_PSTATE procedure, and the caller is expected to make another PAL_SET_PSTATE request to transition to the desired P-state. The *transition_latency_2* field in the *pstate_buffer* returned by PAL_PSTATE_INFO indicates the time interval the caller needs to wait to have a reasonable chance of success when initiating another PAL_SET_PSTATE call.

If the logical processor belongs to a software-coordinated dependency domain, the PAL_SET_PSTATE procedure will change the domain parameters, which will result in all logical processors in that domain to transition to the requested P-state. If the logical processor belongs to a hardware-coordinated dependency domain, the PAL_SET_PSTATE procedure will attempt to change the power/performance characteristics only for that logical processor, which will result in either partial or complete transition to the requested P-state. In case of partial transition (see Figure 11-21, “Computation of performance_index” on page 2:310 for an example, where the logical processor transitions from state P0 to state P3 in partial increments), the logical processor may attempt to perform changes at a later time to the local parameters and/or domain parameters to transition to the originally requested P-state. If the logical processor belongs to a hardware-independent dependency domain, the PAL_SET_PSTATE procedure will attempt to change the domain parameters, which will transition the logical processor in that domain to the requested P-state.

PAL_GET_PSTATE: This procedure returns the performance index of the logical processor, relative to the highest available P-state – P0 – which has an index value of 100. For example, if the value returned by the procedure is 80, it indicates that the performance of the logical processor over the last time period was 20% lower than the P0 performance capability of the logical processor. The performance index is measured over the time interval since the last PAL_GET_PSTATE call. Every invocation of the PAL_GET_PSTATE procedure resets the internal performance measurement logic, and initiates a new *performance_index* count, which is reported when the next PAL_GET_PSTATE procedure call is made.

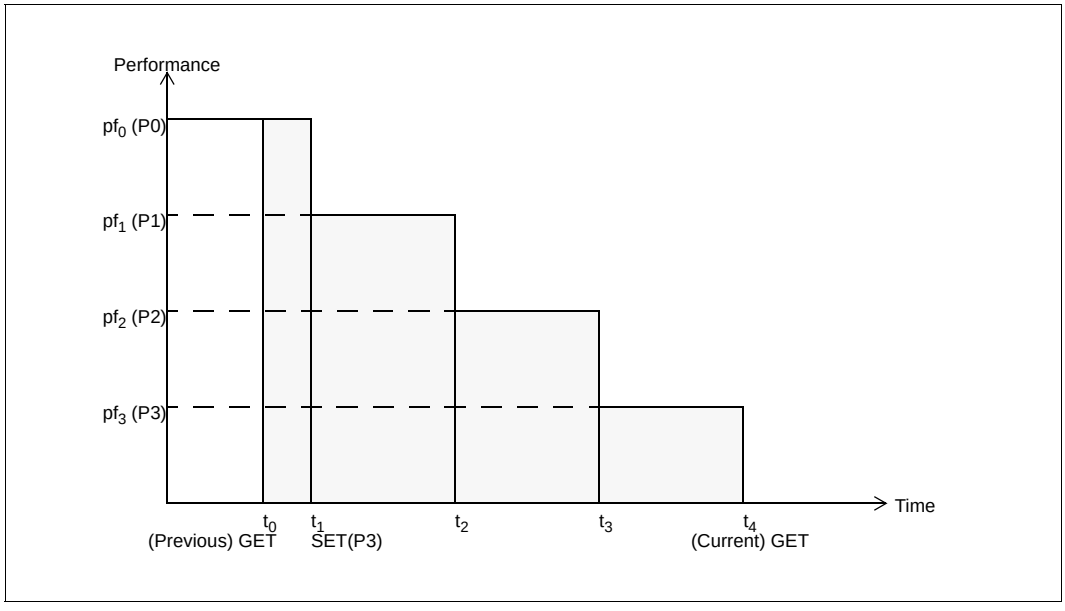
If the logical processor belongs to a software-coordinated dependency domain or a hardware-independent dependency domain, the performance index returned corresponds to the target P-state requested by the most recent successful PAL_SET_PSTATE procedure call.

If the logical processor belongs to a hardware-coordinated dependency domain, the performance index returned will be a weighted-average sum of the *perf_index* values corresponding to the different P-states that the logical processor was operating in before the PAL_GET_PSTATE procedure was called. Note that this return value may not necessarily correspond to the performance index of the target P-state requested by the most recent PAL_SET_PSTATE procedure call. For example, let's assume that the previous PAL_GET_PSTATE procedure was called at time t_0, when the processor was operating in state P0. The previous PAL_SET_PSTATE procedure requested a transition from P0 to P3. The transition happened over a period of time, such that the logical processor went through states P1 at time t_1, P2 at time t_2 and P3 at time t_3, and was in state P3 at time t_4 when the current PAL_GET_PSTATE procedure was called. The *performance_index* returned is calculated as:

performance_index =
((time spent in P0 after the previous PAL_GET_PSTATE) * (*performance_index* for P0) +
(time spent in P1) * (*performance_index* for P1) +
(time spent in P2) * (*performance_index* for P2) +
(time spent in P3 up to the current PAL_GET_PSTATE) * (*performance_index* for P3)) /
(time interval between previous and current PAL_GET_PSTATE) =

$$\frac{(t_1 - t_0) \times pf_0 + (t_2 - t_1) \times pf_1 + (t_3 - t_2) \times pf_2 + (t_4 - t_3) \times pf_3}{t_4 - t_0}$$

Figure 11-21. Computation of *performance_index*



As seen above, for a hardware-coordinated dependency domain, the PAL_GET_PSTATE procedure allows the caller to get feedback on the dynamic performance of the processor over the last time period. The caller can use this information to get better system utilization over the next time period by changing the P-state in correlation with the current workload demand.

11.6.1.1 Interaction of P-states with HALT State

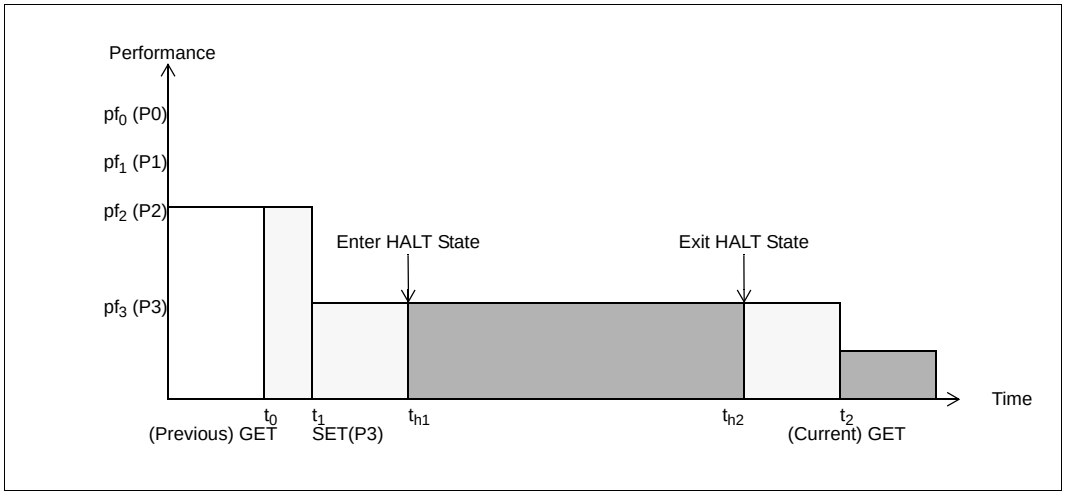
It is possible for a logical processor to enter and exit a HALT state between two consecutive calls to PAL_GET_PSTATE. Since the logical processor is not executing any instructions while in the HALT state, the performance index contribution during this period is essentially 0, and will not be accounted for in the *performance_index* value returned when the next PAL_GET_PSTATE procedure call is made.

For example, let us assume that the previous PAL_GET_PSTATE procedure was called at time t_0, when the processor was operating in state P2. The previous PAL_SET_PSTATE procedure initiated a transition from P2 to P3 at time t_1. The processor entered HALT state at time t_{h1}, and exited the HALT state at time t_{h2}, and was in state P3 at time t_2 when the current PAL_GET_PSTATE procedure was called. The *performance_index* returned is calculated as:

performance_index =
((time in P2 after the previous PAL_GET_PSTATE) * (*performance_index* for P2) +
(time in P3 before entering HALT state) * (*performance_index* for P3) +
(time in P3 after exiting HALT up to current PAL_GET_PSTATE))) *
(*performance_index* for P3)) /
(time interval between previous and current GET, excluding time spent in HALT) =

$$\frac{(t_1 - t_0) \times pf_2 + (t_{h1} - t_1) \times pf_3 + (t_2 - t_{h2}) \times pf_3}{(t_2 - t_0) - (t_{h2} - t_{h1})}$$

Figure 11-22. Interaction of P-states with HALT State



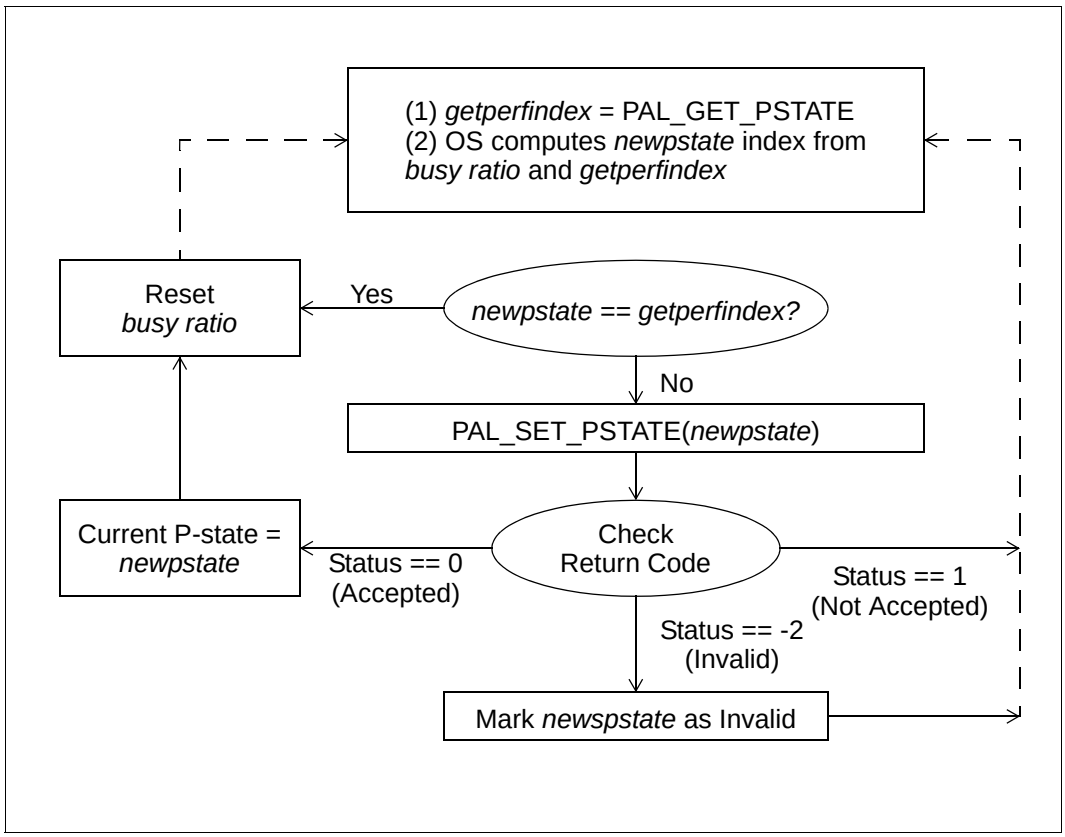
As shown above, the value returned for *performance_index* does not account for the performance during the time spent by the logical processor in the HALT state. This provides for better accuracy in the value reported for *performance_index*, allowing the caller to make optimal adjustments to the system utilization even in scenarios where we have interactions between P-states and HALT state.

2. New Section 13.3.4 in Volume 2, Part II

13.3.4 P-state Feedback Mechanism Flow Diagram

The example flowchart shown below illustrates how the caller can utilize the PAL_SET_PSTATE and the PAL_GET_PSTATE procedures to manage system utilization and power consumption, for a processor implementation that belongs to either a hardware-coordinated dependency domain or a hardware-independent dependency domain. At the beginning of the loop, PAL_GET_PSTATE gives the performance characteristics of the processor over the last time period. It is assumed that the caller maintains an internal count for determining the busy ratio of the logical processor (busy ratio can be defined as the percentage of time the processor was busy executing instructions and not idle). The caller then seeks to adjust the P-state for the next time period to match the busy ratio from the previous time period. For example, if the busy ratio for a given period was 100%, and the *performance_index* returned by PAL_GET_PSTATE was 60, then this indicates that the P-state for the next time period should be P0 (which has performance index of 100). The caller would then call the PAL_SET_PSTATE procedure to transition the processor to the P0 state. In essence, if the busy ratio is greater than the *performance_index* returned by PAL_GET_PSTATE, the caller responds to the increased demand requirement of the workload by transitioning the processor to a higher-performance P-state. Alternatively, if the busy ratio is lower than the *performance_index* returned by PAL_GET_PSTATE, the caller responds by transitioning the processor to a lower performance P-state, which consumes less power and operates at reduced performance.

Figure 13-6. Flowchart Showing P-state Feedback Policy



Such an adaptive policy implemented by the caller to dynamically respond to system workload characteristics using P-states allows for efficient power utilization – the processor consumes additional power by operating at a higher performance level only when the current workload requires it to do so.

3. New PAL Power Management Procedures

Table 11-16. PAL Power Information and Management Procedures

Procedure	Idx	Class	Conv.	Mode	Description
PAL_GET_PSTATE	262	Opt.	Stacked	Both	Returns information on the performance index of the processor.
PAL_PSTATE_INFO	44	Opt.	Static	Both	Returns information about the P-states supported by the processor.
PAL_SET_PSTATE	263	Opt.	Stacked	Both	Request processor to enter power/performance state.

PAL_GET_PSTATE

Return Information on the Performance Index of the Processor

Purpose: Returns the performance index of the processor.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_GET_PSTATE within the list of PAL procedures.
type	Type of *performance_index* value to be returned by this procedure.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_GET_PSTATE procedure.
performance_index	Unsigned integer denoting the processor performance for the time duration since the last PAL_GET_PSTATE procedure call was made. The value returned is between 0 and 100, and is relative to the performance index of the highest available P-state.
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error, but accuracy of performance index has been impacted by a thermal throttling event, or a hardware-initiated event.
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure returns the performance index of the processor over the time period between the previous and the current invocations of PAL_GET_PSTATE, and is relative to the highest available P-state. For processors that belong to a software-coordinated dependency domain or a hardware-independent dependency domain, the *performance_index* value returned will correspond to the target P-state requested by the most recent PAL_SET_PSTATE procedure call.

For processors that belong to a hardware-coordinated dependency domain, the *type* argument allows the caller to select the *performance_index* value that will be returned. See Table 11-48 below for details.

Table 11-74. PAL_GET_PSTATE *type* Argument

type	Description
0	The *performance_index* returned will correspond to the target P-state requested by the most recent PAL_SET_PSTATE procedure call.
1	The *performance_index* is a weighted-average value of the different P-states that the processor was operating in for the time duration between the current PAL_GET_PSTATE procedure call, and the previous invocation of PAL_GET_PSTATE with type=1. This allows the caller to establish a new starting point for subsequent computation of the weighted-average *performance_index*. See Section 11.6.1, “Power/Performance States (P-states)” on page 20 for more details on how the weighted average value is derived.

Table 11-74. PAL_GET_PSTATE *type* Argument

type	Description
2	The *performance_index* is a weighted-average value of the different P-states that the processor was operating in for the time duration between the current PAL_GET_PSTATE procedure call, and the previous invocation of PAL_GET_PSTATE with type=1. This allows the caller to sample the current value of the *performance_index*, without affecting the starting point used for computing the weighted-average performance_index.
3	The *performance_index* returned will correspond to the current instantaneous P-state of the logical processor, at the time of the procedure call.
All Other Values	Reserved

For processors that belong to a software-coordinated dependency domain or a hardware-independent dependency domain, the PAL_GET_PSTATE procedure should always be called with type argument value of 0 or 3. Note that the *performance_index* returned for *type*=0 and *type*=3 will have identical values for these coordination domains. This is because the most recent PAL_SET_PSTATE procedure call will always succeed in transitioning to the requested performance state for these coordination domains (see PAL_SET_PSTATE procedure description for additional details).

If there was a thermal-throttling event or any hardware-initiated event, which affected the processor power/performance for the current time period and the accuracy of the *performance_index* value has been impacted by the event, then the procedure will return with *status*=1. The *performance_index* returned in this case will still have a value between 0 and 100.

The procedure returns with a *performance_index* value of 100 when invoked for the first time. For subsequent invocations, the procedure will return the *performance_index* value corresponding to the processor performance in the time duration between the previous and current calls to PAL_GET_PSTATE.

If the processor had transitioned to a HALT state (see Section 11.6.1, “Power/Performance States (P-states)” on page 20) in between successive invocations to the PAL_GET_PSTATE procedure, the performance index computation returned will not take into account the performance of the processor during the time spent in HALT state (see Section 11.6.1.1, “Interaction of P-states with HALT State” on page 26 for details).

PAL_PSTATE_INFO

Get Information for Power/Performance States

Purpose: Returns information about the P-states supported by the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_PSTATE_INFO within the list of PAL procedures.
pstate_buffer	64-bit pointer to a 256-byte buffer aligned on an 8-byte boundary.
Reserved	0
Reserved	0

Returns:

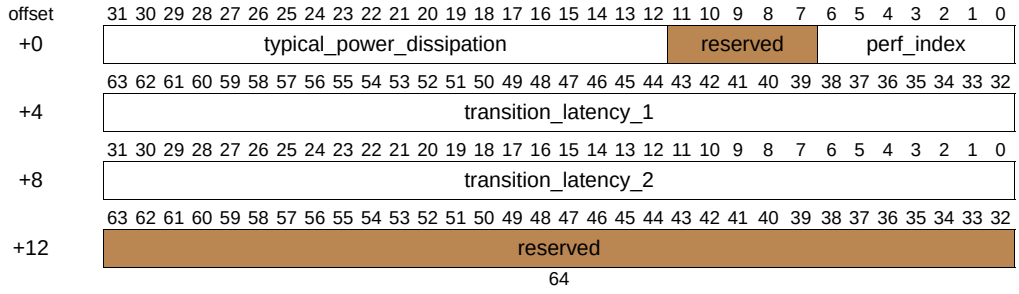
Return Value	Description
status	Return status of the PAL_PSTATE_INFO procedure.
pstate_num	Unsigned integer denoting the number of P-states supported. The maximum value of this field is 16.
dd_info	Dependency domain information
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: Information about available P-states is returned in the data buffer referenced by *pstate_buffer*. Entries in the buffer are organized in an ascending order. For example, P0 (the highest performance P-state) state information is index 0 in the buffer, P1 state is index 1 in the buffer, and so on. The return argument *pstate_num* indicates the number of P-states supported on the given implementation. For example, if *pstate_num* is 4, it indicates that P-states P0-P3 are available for that implementation. Information in *pstate_buffer* is returned only for entries corresponding to the available P-states. Entries corresponding to unimplemented P-states must be ignored. Figure 11-49 illustrates the format of the *pstate_buffer*.

Figure 11-49. Layout of *pstate_buffer* Entry



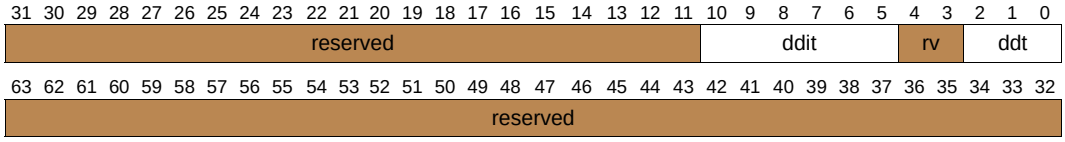
- *typical_power_dissipation* is a 20-bit field denoting the typical processor package power dissipation if all logical processors on the package are placed in this P-state, measured in milliwatts.
- *perf_index* is a 7-bit field denoting the performance index of this P-state, relative to the highest available P-state (P0). This field is enumerated on a scale of 0…100, with the value of 100 corresponding to the P0 state. For example, if the P1-state has a value of 75, and the next P-state (P2) has a value

of 50, it implies that P1 performance is 25% lower than P0 performance, and P2 performance is 50% lower than P0 performance.

- *transition_latency_1* is a 32-bit field indicating the minimum number of processor cycles required to initiate a transition to this P-state from any other P-state.
- *transition_latency_2* is a 32-bit field indicating the minimum recommended number of processor cycles that the caller should wait, before initiating a new P-state transition with a reasonable chance of acceptance. This field is intended to give the caller an estimation of the frequency with which PAL_SET_PSTATE procedure calls should be made, without having the transition request be not accepted.

Dependency domain details for the logical processor are returned in *dd_info*. See Figure 11-50 for *dd_info* layout.

Figure 11-50. Layout of *dd_info* Parameter



- *ddt* (Dependency Domain Type) is a 3-bit unsigned integer denoting the type of dependency domains that exist on the processor package. The possible values are shown in Table 11-65. See Section 11.6.1, “Power/Performance States (P-states)” on page 19 for details of the values in this field.

Table 11-65. Values for *ddt* Field

Value	Description
0	Hardware independent
1	Hardware coordinated
2	Software coordinated
3-7	Reserved

- *ddid* (Dependency Domain Identifier) is a 6-bit unsigned integer denoting this logical processor's dependency domain. The *ddid* values are unique only for a given processor package. Software can use the *ddid* field to determine which logical processors belong to the same dependency domain within the package.

For more information on performance states and power management, refer to Section 11.6.1, “Power/Performance States (P-states)” on page 19.

PAL_SET_PSTATE

Request Processor to Enter Power/Performance State

Purpose: To request a processor transition to a given P-state.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_SET_PSTATE within the list of PAL procedures.
p_state	Unsigned integer denoting the processor P-state being requested.
force_pstate	Unsigned integer denoting whether the P-state change should be forced for the logical processor.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_SET_PSTATEprocedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed without error, but transition request was not accepted
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: PAL_SET_PSTATE is used to request the transition of the processor to the P-state specified by the *p_state* input parameter. The PAL_SET_PSTATE procedure does not wait for the transition to complete before returning back to the caller. The request may either be accepted (*status* = 0) or not accepted (*status* = 1), depending on hardware capabilities and implementation-specific event conditions. If the request is not accepted, then no transition is performed, and it is up to the caller to make another PAL_SET_PSTATE procedure call to transition to the desired P-state. When the request is accepted, it will attempt to initiate a transition to the requested performance state. For processors that belong to a software-coordinated dependency domain or a hardware-independent dependency domain, the procedure will always succeed in transitioning to the requested performance state. If the processor belongs to a hardware-coordinated dependency domain, the procedure will make a best-case attempt at fulfilling the transition request, based on the nature of the dependencies that exist between the logical processors in the domain. In such circumstances, the procedure may initiate no transition, partial transition or full transition to the requested P-state. Since there is the possibility that the procedure may initiate no processor transition, there are implementation-specific forward progress requirements.

The *force_pstate* argument may be used for a hardware-coordinated dependency domain when it is necessary to get a deterministic response for the P-state transition at the expense of compromising the power/performance of other logical processors in same domain. If the *force_pstate* argument is non-zero, and if the request is accepted, the procedure will initiate the P-state transition on the logical processor regardless of any dependencies that exist in the dependency domain at the time the procedure is called. The *force_pstate* argument is ignored for software-coordinated and hardware-independent dependency domain.

4. Add a new row to Processor Features, Table 11-54 on page 2:360:

Table 11-54. Processor Features

Bit	Class	Control	Description
52	Opt.	Req.	Disable P-states. When 1, the PAL P-state procedures (PAL_PSTATE_INFO, PAL_SET_PSTATE, PAL_GET_PSTATE) will return with a status of -1 (Unimplemented procedure).

8. Allow undefined behavior for all must-be-last instructions

1. Volume 1, Section 4.1.2, 4th paragraph, change from:

   A `cover` instruction must be the last instruction in an instruction group otherwise an Illegal Operation fault is taken.

   to:

   A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

2. Volume 2, Section 6.5.4, 2nd paragraph, change from:

   The `cover` instruction must be specified as the last instruction in a bundle group otherwise an Illegal Operation fault is taken.

   to:

   A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

3. Volume 3, `bsw` instruction page:

   a. Description, 2nd paragraph, change from:

      A `bsw` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken.

      to:

      A `bsw` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

   b. Operation. Change from:

```
if (!followed_by_stop())
  illegal_operation_fault();
```

      to:

```
if (!followed_by_stop())
  undefined_behavior();
```

   c. Interruptions. Remove “Illegal Operation fault”.

4. Volume 3, `clrrrb` instruction page.

   a. Description, 2nd paragraph, change from:

      This instruction must be the last instruction in an instruction group, or an Illegal Operation fault is taken.

      to:

      This instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

   b. Operation. Change from:

```
if (!followed_by_stop())
  illegal_operation_fault();
```

to:

```
if (!followed_by_stop())
  undefined_behavior();
```

c. Interruptions. Change “Illegal Operation fault” to “None”.

5. Volume 3, `cover` instruction page.

a. Description, 2nd paragraph, change from:

A `cover` instruction must be the last instruction in an instruction group. Otherwise, an Illegal Operation fault is taken.

to:

A `cover` instruction must be the last instruction in an instruction group; otherwise, operation is undefined.

b. Operation. Change from:

```
if (!followed_by_stop())
  illegal_operation_fault();
```

to:

```
if (!followed_by_stop())
  undefined_behavior();
```

c. Interruptions. Remove “Illegal Operation fault”.

6. Volume 3, `rfi` instruction page.

a. Description, 2nd paragraph, change from:

This instruction must be immediately followed by a stop. Otherwise, an Illegal Operation fault is taken.

to:

This instruction must be immediately followed by a stop; otherwise, operation is undefined.

b. Operation. Change from:

```
if (!followed_by_stop())
  illegal_operation_fault();
```

to:

```
if (!followed_by_stop())
  undefined_behavior();
```

c. Interruptions. Remove “Illegal Operation fault”.

9. Addition of PAL_BRAND_INFO

1. Add a new row to Table 11-14, Volume 2, Part I:

Procedure - PAL_BRAND_INFO

Idx - 274

Class - Opt.

Conv. - Stacked

Mode - Both

Description - Provides processor branding information.

2. New PAL Procedure:

PAL_BRAND_INFO

Provides Processor Branding Information

Purpose: Provides processor branding information.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_BRAND_INFO within the list of PAL procedures.
info_request	Unsigned 64-bit integer specifying the information that is being requested. (See Table 11-25)
address	Unsigned 64-bit integer specifying the address of the 128-byte block to which the processor brand string shall be written.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_BRAND_INFO procedure.
brand_info	Brand information returned. The format of this value is dependent on the input values passed.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-6	Input argument is not implemented

Description: PAL_BRAND_INFO procedure calls are used to ascertain the processor branding information.

The *info_request* input argument for PAL_BRAND_INFO describes which processor branding information is being requested. The *info_request* values are split into two categories: architected and implementation-specific. The architected *info_request* have values from 0-15. The implementation-specific *info_request* have values 16 and above. The architected *info_request* are described in this document. The implementation-specific *info_request* are described in processor-specific documentation.

This call returns the processor brand information as requested with the *info_request* argument. Table 11-25 describes the values.

Table 11-25. Processor Brand Information Requested

Value	Description
0	The ASCII brand identification string will be copied to the address specified in the address input argument. The processor brand identification string is defined to be a maximum of 128 characters long; 127 bytes will contain characters and the 128th byte is defined to be NULL (0). A processor may return less than the 127 ASCII characters as long as the string is null terminated. The string length will be placed in the *brand_info* return argument.
All Other Values	Reserved

This procedure will return an invalid argument if an unsupported *info_request* argument is passed as an input or a -6 if the requested information was not available on the current processor.

10. Addition of PAL_MC_ERROR_INJECT

1. Add a new description row to Table 11-15, “PAL Machine Check Handling Procedures”, of Volume 2, Part I:

Procedure:	PAL_MC_ERROR_INJECT
Idx:	276
Class:	Opt.
Convention:	Stacked
Mode:	Both
Description:	Injects the requested processor error or returns information on the supported injection capabilities for this particular processor implementation.

2. Add the following paragraphs to Section 11.7,”PAL Glossary”:

   Corrected error: All errors of this type are either corrected by the processor/platform hardware/firmware. This severity is for logging purposes only. There is no architectural damage caused by the detecting and reporting functions. Corrected errors require no operating system intervention to correct the error.

   Recoverable error: An uncorrected error which can corrupt state, but the state information is known. Recoverable errors cannot be corrected by either the hardware or firmware. This type of error requires operating system analysis and a corrective action to recover. System operation/state may be impacted.

   Fatal error: An uncorrected error which can corrupt state, and the state information is not known. These type of errors cannot be corrected by the hardware, firmware, or the operating system. The integrity of the system, including the I/O devices is not guaranteed and may require I/O device initialization and a system reboot to continue. Fatal errors may or may not be contained within the processor or memory hierarchy

3. New PAL Procedure:

PAL_MC_ERROR_INJECT

Inject Processor Error

Purpose: Injects the requested processor error or returns information on the supported injection capabilities for this particular processor implementation.

Calling Conv: Stacked

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_MC_ERROR_INJECT within the list of PAL procedures.
err_type_info	Unsigned 64-bit integer specifying the first level error information which identifies the error structure and corresponding structure hierarchy, and the error severity.
err_struct_info	Unsigned 64-bit integer identifying the optional structure specific information that provides the second level details for the requested error.
err_data_buffer	64-bit physical address of a buffer providing additional parameters for the requested error. The address of this buffer must be 8-byte aligned.

Returns:

Return Value	Description
status	Return status of the PAL_MC_ERROR_INJECT procedure.
capabilities	64-bit vector specifying the supported error injection capabilities for the input argument combination of *struct_hier*, *err_struct* and *err_sev* fields in *err_type_info.*
resources	64-bit vector specifying the architectural resources that are used by the procedure.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-4	Call completed with error; the requested error could not be injected due to failure in locating the target location in the specified structure.
-5	Argument was valid, but requested error injection capability is not supported.

Description: This procedure enables error injection into processor structures based on information specified by *err_type_info*, *err_struct_info* and *err_data_buffer.* Each invocation of the procedure enables a single error to be injected. The procedure supports error injection for at least one error of each severity type (correctable, recoverable, fatal).

The *err_type_info* argument specifies details of the error injection operation that is being requested (see Figure 11-51). The *err_struct_info* and *err_data_buffer* specify additional optional information. The format of *err_struct_info* is specified for each supported structure type indicated by the *err_struct* field in *err_type_info*. *err_data_buffer* is optional, depending on the structure type and whether *trigger* functionality is used. If *err_data_buffer* is not required for the error injection, PAL will not attempt to access the memory location specified in this parameter.

Figure 11-51. err_type_info

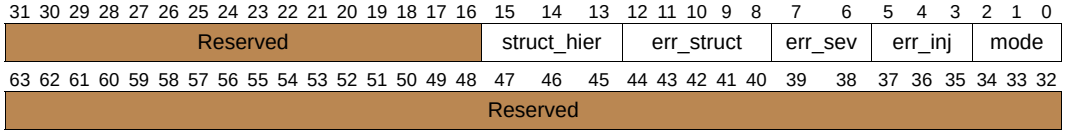


Table 11-88. *err_type_info*

Field	Bits	Description
mode	2:0	Indicates the mode of operation for this procedure: 0 – Query mode 1 – Error inject mode (*err_inj* should also be specified) 2 – Cancel outstanding trigger. All other fields in *err_type_info*, *err_struct_info* and *err_data_buffer* are ignored. All other values are reserved.
err_inj	5:3	Indicates the mode of error injection: 0 – Error inject only (no error consumption) 1 – Error inject and consume All other values are reserved.
err_sev	7:6	Indicates the severity desired for error injection/query. Definitions of the different error severity types is given in Section 11.8, “PAL Glossary” on page 2:333. 0 – Corrected error 1 – Recoverable error 2 – Fatal error 3 – Reserved
err_struct	12:8	Indicates the structure identification for error injection/query: 0 - Any structure (cannot be used during *query mode*). When selected, the structure type used for error injection is determined by PAL. 1 – Cache 2 – TLB 3 – Register file 4 – Bus/System interconnect 5-15 – Reserved 16-31 – Processor specific error injection capabilities. *err_data_buffer* is used to specify error types. Please refer to the processor specific documentation for additional details.
struct_hier	15:13	Indicates the structure hierarchy for error injection/query: 0 - Any level of hierarchy (cannot be used during *query mode*). When selected, the structure hierarchy used for error injection is determined by PAL. 1 – Error structure hierarchy level-1 2 – Error structure hierarchy level-2 3 – Error structure hierarchy level-3 4 – Error structure hierarchy level-4 All other values are reserved.
Reserved	63:16	Reserved

If *query mode* is selected through the mode bit in the *err_type_info* parameter, the return value in the *capabilities* vector indicates which error injection types are *individually* supported on the underlying implementation for the corresponding values of *err_struct*, *struct_hier* and *err_sev* fields in *err_type_info*. The caller is expected to iterate through all combinations of err_struct, *struct_hier* and *err_sev* to determine the full extent of *individual* error injection types supported by the underlying implementation.

The *capabilities* vector does not indicate which combinations of error injection inputs from *err_struct_info* are supported by the implementation. For example, if an implementation supports *tag* error injection only for instruction caches and *data* error injection only for data caches, this cannot be determined by the *capabilities* vector. In this instance, the *capabilities* vector will report *i=1*, *d=1*, *tag=1*, *data=1*, indicating that the error injection is supported *individually* for instruction and data caches, and for *tag* and *data* fields, but not indicating which *combinations* of *i*, *d*,

tag, and *data* are supported for error injection. The caller is required to use the *query mode* with appropriate inputs in *err_struct_info* to determine which combinations of error injection types are supported. If a given combination is not supported, the procedure returns with status -5.

The procedure supports both an *Error inject* and *Error inject and consume* mode. In the former mode, the procedure performs the requested error injection in the specified structure, but does not perform any additional actions that can lead to consumption of the error and generation of the subsequent machine check. In *Error inject and consume* mode, the procedure will inject the error in the specified structure, and will perform additional operations to ensure that the error condition is encountered resulting in a machine check. Note that in this case, the machine check will be generated within the context of this procedure.

The procedure also provides the ability to set error injection triggers for a given structure. In these cases, the error injection is delayed until the operation specified by the trigger is encountered and the executing context has the specified privilege level. In the absence of triggers, the error injection is performed at the time of procedure execution. The *mode* field in *err_type_info* determines whether the error is injected, or injected and consumed, when the trigger operation is encountered. Error injection emulation can be combined with triggers for a given structure when the *Error inject and consume* mode is selected. There can be only one outstanding trigger for each supported structure. Subsequent procedure calls that use the trigger functionality for the same structure will overwrite the previous trigger parameters. It is also possible to cancel outstanding triggers by specifying *Cancel outstanding trigger* via the *mode* bit in *err_type_info*. When using this mode, it is possible that the procedure execution may itself satisfy the trigger conditions while in the process of cancelling the last programmed trigger.

To support triggers, PAL may use existing architectural resources. The *resources* return value defines the list of resources that are being used by PAL (see Figure 11-52). Procedure operation is undefined if there are any conflicts for these resources with other software running on the system.

Figure 11-52. *resources* Return Value

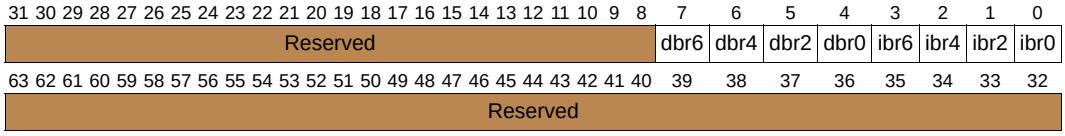


Table 11-89. *resources* Return Value

Field	Bits	Description
ibr0	0	When 1, indicates that IBR0 is being used by the procedure for trigger functionality.
ibr2	1	When 1, indicates that IBR1 is being used by the procedure for trigger functionality.
ibr4	2	When 1, indicates that IBR2 is being used by the procedure for trigger functionality.
ibr6	3	When 1, indicates that IBR3 is being used by the procedure for trigger functionality.
dbr0	4	When 1, indicates that DBR0 is being used by the procedure for trigger functionality.
dbr2	5	When 1, indicates that DBR1 is being used by the procedure for trigger functionality.
dbr4	6	When 1, indicates that DBR2 is being used by the procedure for trigger functionality.
dbr6	7	When 1, indicates that DBR3 is being used by the procedure for trigger functionality.

Multiprocessor coherency is not guaranteed when error injection is performed using this procedure. Please refer to the processor-specific documentation for further details regarding possible scenarios which can result in loss of coherency.

In cases where an error cannot be injected due to failure in locating the specified target location (cacheline, TC, TR, register number) for the given set of input arguments, the procedure will return with status -4. For example, if the caller requests an error injection in the cache and specifies *cl_id*=0 (virtual address provided), then PAL will attempt to locate the cacheline as indicated by the input virtual address. If the corresponding cacheline cannot be found (the cacheline could have been evicted from the cache in the time interval between the procedure call and the search process, or the cacheline may be in *invalid* state), then the procedure returns with a status value of -4.

The procedure does not check the settings of the error promotion bits (bit 53 and bit 60 in PAL_PROC_GET_FEATURES) before injecting an error in the specified structure. Based on the configuration of these bits, the severity of the error reported may vary.

The detailed descriptions of *err_struct_info* and *err_data_buffer* are shown below.

Figure 11-53. *err_struct_info* – Cache

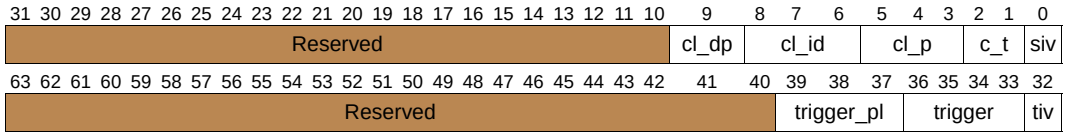


Table 11-90. *err_struct_info* – Cache

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*c_t,cl_p,cl_id*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
c_t	2:1	Indicates which cache should be used for error injection: 0 – Reserved 1 – Instruction cache 2 – Data or unified cache 3 – Reserved
cl_p	5:3	Indicates the portion of the cacheline where the error should be injected: 0 – Reserved 1 – Tag 2 – Data 3 – mesi All other values are reserved.
cl_id	8:6	Indicates which mechanism is used to identify the cacheline to be used for error injection: 0 – Reserved 1 – Virtual address provided in the *inj_addr* field of the buffer pointed to by *err_data_buffer* should be used to identify the cacheline for error injection. 2 – Physical address provided in the *inj_addr* field of the buffer pointed to by *err_data_buffer*should be used to identify the cacheline for error injection. 3 – *way* and *index* fields provided in *err_data_buffer* should be used to identify the cacheline for error injection. All other values are reserved.

Table 11-90. *err_struct_info* – Cache (Continued)

Field	Bits	Description
cl_dp	9	When 1, indicates that a multiple bit, non-correctable error should be injected in the cacheline specified by *cl_id*. If this injected error is not consumed, it may eventually cause a data-poisoning event resulting in a corrected error signal, when the associated cacheline is cast out (implicit or explicit write-back of the cacheline). The error severity specified by *err_sev* in *err_type_info* must be set to 0 (*corrected error*) when this bit is set.
Reserved	31:10	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*: 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148. 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148. All other values are reserved.
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected: 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-54. *capabilities* vector for cache

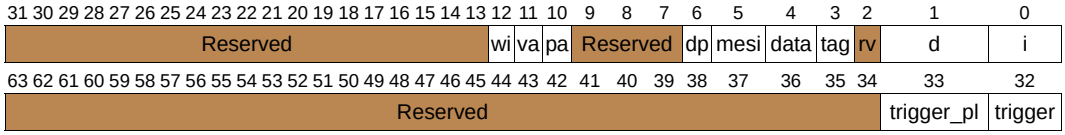


Table 11-91. *capabilities* vector for cache

Field	Bits	Description
i	0	Error injection for instruction caches is supported
d	1	Error injection for data caches is supported
rv	2	Reserved
tag	3	Error injection in *tag* portion of cacheline is supported
data	4	Error injection in *data* portion of cacheline is supported
mesi	5	Error injection in *mesi* portion of cacheline is supported
dp	6	Error injection that results in data poisoning events is supported
Reserved	9:6	Reserved
pa	10	Error injection with physical address input is supported
va	11	Error injection with virtual address input is supported

Table 11-91. *capabilities* vector for cache (Continued)

Field	Bits	Description
wi	12	Error injection with *way* and *index* input is supported
Reserved	31:13	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer always needs to be specified for *cache* only if *siv* is 1 or *tiv* is 1, in *err_struct_info*.

Figure 11-55. Buffer pointed to by *err_data_buffer* – Cache

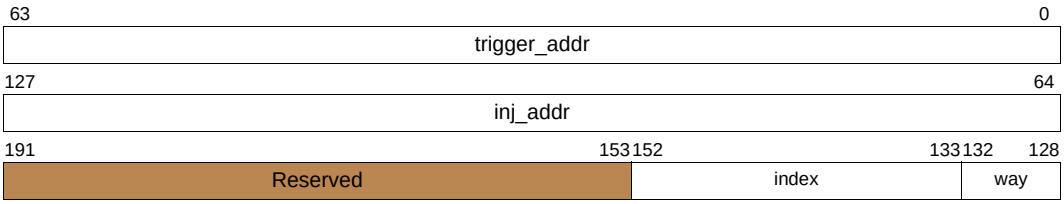


Table 11-92. Buffer pointed to by *err_data_buffer* – Cache

Field	Bits	Description
trigger_addr	63:0	64-bit virtual address to be used by the *trigger* in the *err_struct_info* input argument. This field is ignored if *tiv* in *err_struct_info* is 0. The field is defined similar to the *addr* field in the debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 147.
inj_addr	127:64	64-bit virtual or physical address used to identify the cacheline to be used for error injection. This field is valid only if *cl_id* in *err_struct_info* corresponds to either *va* or *pa* (value 1 or 2).
way	132:128	Indicates the *way* information for error injection. This is used in combination with the *index* field to identify the cacheline for error injection. This field is valid only if *cl_id* in *err_struct_info* is 3, else it is ignored.
index	152:133	Indicates the *index* information for error injection. This is used in combination with the *way* field to identify the cacheline for error injection. This field is valid only if *cl_id* in *err_struct_info* is 3, else it is ignored.
Reserved	191:153	Reserved

Figure 11-56. *err_struct_info* – TLB

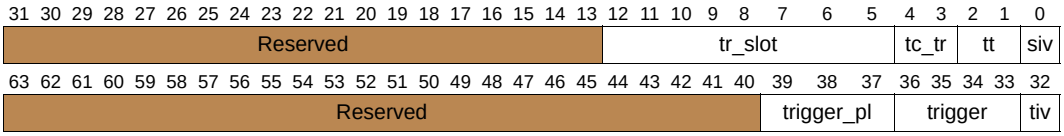


Table 11-93. *err_struct_info* – TLB

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*i_d, tc_tr, tr_slot*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
tt	2:1	Indicates which TLB should be used for error injection: 0 – Reserved 1 – Instruction TLB 2 – Data TLB 3 – Reserved
tc_tr	4:3	Indicates which portion of TLB should be used for error injection: 0 – Reserved 1 – tc: error should in injected in a Translation Cache (TC) entry. For TC insertion, the entry is identified by the *vpn* and *rid* fields in *err_data_buffer* 2 – tr: error should in injected in a Translation Register (TR) entry. For TR insertion, the slot number is specified by the *tr_slot* field. 3 – Reserved
tr_slot	12:5	Indicates the Translation Register (TR) slot number where the error should be injected. This field is valid only when *tc_tr* is 2, else it is ignored.
Reserved	31:13	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The virtual address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*: 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148. 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148. All other values are reserved.
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-57. *capabilities* vector for TLB

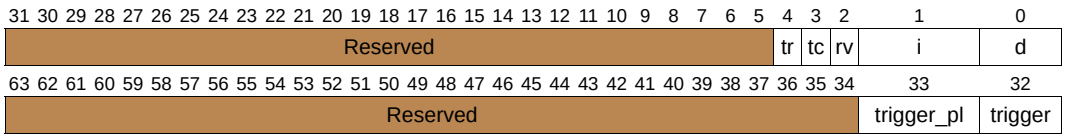


Table 11-94. *capabilities* vector for TLB

Field	Bits	Description
d	0	Error injection for data TLB is supported
i	1	Error injection for instruction TLB is supported
rv	2	Reserved
tc	3	Error injection in TC entries is supported
tr	4	Error injection in TR entries is supported
Reserved	31:5	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer needs to be specified for *TLB* only if *tiv* is 1 or if *tc_tr* value corresponds to *tc*, in *err_struct_info*.

Figure 11-58. Buffer pointed to by *err_data_buffer* – TLB

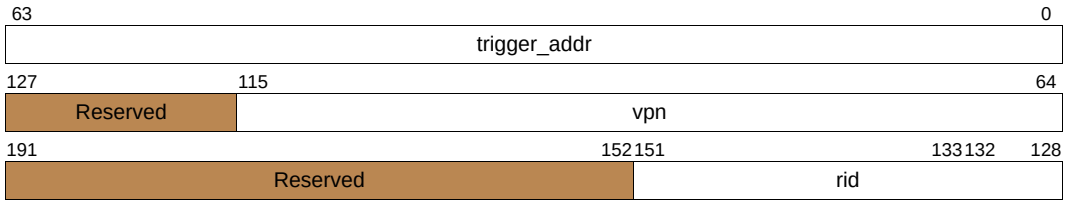


Table 11-95. Buffer pointed to by *err_data_buffer* – TLB

Field	Bits	Description
trigger_addr	63:0	64-bit virtual address to be used by the *trigger* in the *err_struct_info* input argument. The field is defined similar to the *addr* field in debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 147.
vpn	115:64	Indicates the Virtual page number. This field is is valid only when *tc_tr* in *err_struct_info* is 1. *vpn* used in combination with *rid* to identify the TC entry for error injection.
Reserved	127:116	Reserved
rid	151:128	Indicates the region identifier. This field is valid only when *tc_tr* in *err_struct_info* is 0. *rid* is used in combination with *vpn* to identify the TC entry for error injection.
Reserved	191:152	Reserved

Figure 11-59. *err_struct_info* – Register File

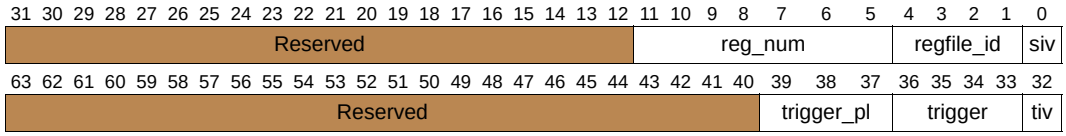


Table 11-96. *err_struct_info* – Register File

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*regfile_id, reg_num*) are valid and should be used for error injection. When 0, the structure information fields are ignored, and the values of these fields used for error injection are implementation-specific.
regfile_id	4:1	Identifies the register file where the error should be injected: 0 – Any register file type. When selected, the register file used for error injection is determined by PAL. 1 – General register (bank0)(GR16-31) 2 – General register (bank1)(GR0-127) 3 – Floating point register 4 – Branch register 5 – Predicate register 6 – Application register 7 – Control register 8 – Region register 9 – Protection key register 10 – Data breakpoint register 11 – Instruction breakpoint register 12 – Performance monitor control register 13 – Performance monitor data register All other values are reserved.
reg_num	11:5	Indicates the register number where the error should be injected. Procedure operation is undefined if there is a conflict between the register number chosen for error injection, and the registers being used by the procedure for code execution (see PAL calling conventions, Section 11.9.2). 0-127: Specific register number corresponding to *regfile_id* 128-254: Reserved for future use 255: Any register number. When selected, the actual register number used for error injection is determined by PAL.
Reserved	31:12	Reserved
tiv	32	When 1, indicates that the trigger information fields (*trigger, trigger_pl*) are valid and should be used for error injection. When 0, the trigger information fields are ignored and error injection is performed immediately.
trigger	36:33	Indicates the operation type to be used as the error trigger condition. The address corresponding to the trigger is specified in the *trigger_addr* field of the buffer pointed to by *err_data_buffer*. 0 – Instruction memory access. The trigger match conditions for this operation type are similar to the IBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148 1 – Data memory access. The trigger match conditions for this operation type are similar to the DBR address breakpoint match conditions as outlined in Section 7.1.2, “Debug Address Breakpoint Match Conditions” on page 148. All other values are reserved.

Table 11-96. *err_struct_info* – Register File (Continued)

Field	Bits	Description
trigger_pl	39:37	Indicates the privilege level of the context during which the error should be injected: 0 – privilege level 0 1 – privilege level 1 2 – privilege level 2 3 – privilege level 3 All other values are reserved. If the implementation does not support privilege level qualifier for triggers (i.e. if *trigger_pl* is 0 in the *capabilities* vector), this field is ignored and triggers can be taken at any privilege level.
Reserved	63:40	Reserved

Figure 11-60. *capabilities* Vector for Register File

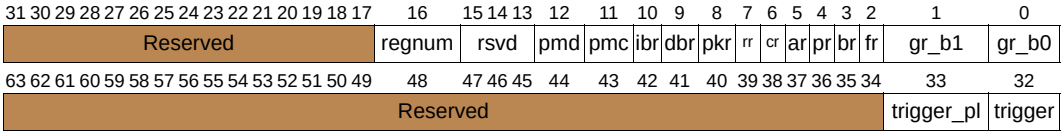


Table 11-97. *capabilities* Vector for Register File

Field	Bits	Description
gr_b0	0	Error injection for General register (bank0) is supported
gr_b1	1	Error injection for General register (bank1) is supported
fr	2	Error injection for Floating point register is supported
br	3	Error injection for Branch register is supported
pr	4	Error injection for Predicate register is supported
ar	5	Error injection for Application register is supported
cr	6	Error injection for Control register is supported
rr	7	Error injection for Region register is supported
pkr	8	Error injection for Protection key register is supported
dbr	9	Error injection for Data breakpoint register is supported
ibr	10	Error injection for Instruction breakpoint register is supported
pmc	11	Error injection for Performance monitor control register is supported
pmd	12	Error injection for Performance monitor data register is supported
Reserved	15:13	Reserved
regnum	16	Error injection with register number input is supported
Reserved	31:17	Reserved
trigger	32	Error injection with trigger is supported
trigger_pl	33	Error injection with privilege level qualifier for trigger is supported
Reserved	63:34	Reserved

err_data_buffer needs to be specified for *register file* only if *tiv* in *err_struct_info* is 1.

Figure 11-61. Buffer pointed to by *err_data_buffer* – Register File

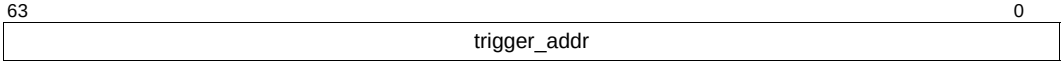


Table 11-98. Buffer pointed to by *err_data_buffer* – Register File

Field	Bits	Description
trigger_addr	63:0	64-bit address to be used by the *trigger* in the *err_struct_info* input argument. The field is defined similar to the *addr* field in the debug breakpoint registers, as specified in Table 7-1, “Debug Breakpoint Register Fields (DBR/IBR)” on page 147.

Figure 11-62. *err_struct_info* – Bus/Processor Interconnect

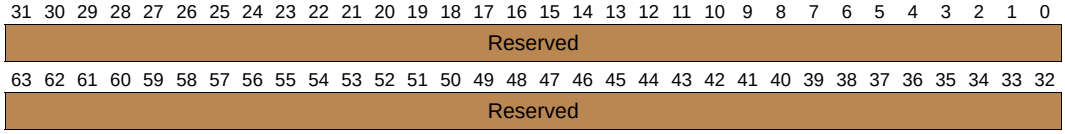


Table 11-99. *err_struct_info* – Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

Figure 11-63. *capabilities* vector for Bus/Processor Interconnect

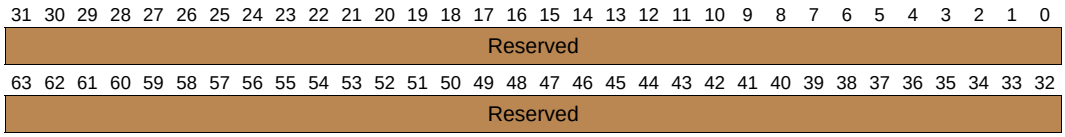


Table 11-100. *capabilities* vector for Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

err_data_buffer does not need to be specified for *bus/system interconnect* only if *tiv* in *err_struct_info* is 1.

11. Addition of PAL_GET_HW_POLICY and PAL_SET_HW_POLICY

1. Add two new description rows to Table 11-14, “PAL Processor Identification, Features, and Configuration Procedures”, Volume 2, Part I:

Procedure	Idx	Class	Conv	Mode	Desc
PAL_GET_HW_POLICY	48	Opt.	Static	Both	Get current hardware resource sharing policy
PAL_SET_HW_POLICY	49	Opt.	Static	Both	Set current hardware resource sharing policy

2. New PAL Procedures:

PAL_GET_HW_POLICY

Retrieve Current Hardware Resource Sharing Policy

Purpose: Returns the current hardware resource sharing policy of the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_GET_HW_POLICY within the list of PAL procedures.
proc_num	Unsigned 64-bit integer that specifies for which logical processor information is being requested. This input argument must be zero for the first call to this procedure and can be a maximum value of one less than the number of logical processors impacted by the hardware resource sharing policy, which is returned by the *num_impacted* return value.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_GET_HW_POLICY procedure.
cur_policy	Unsigned 64-bit integer representing the current hardware resource sharing policy.
num_impacted	Unsigned 64-bit integer that returns the number of logical processors impacted by the *policy* input argument.
la	Unsigned 64-bit integer containing the logical address of one of the logical processors impacted by policy modification.

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure is used to return information on the current hardware resource sharing policy. This procedure is also be used to identify which logical processors (see "Get Information on Logical to Physical Processor Mappings" on page 2:395 for a definition of a logical processor) are impacted by the various hardware sharing policies supported on the processor.

The procedure returns information about the current hardware sharing policy, the total number of logical processors impacted by hardware sharing policies and the logical address of one of the processors impacted by the hardware sharing policy.

The definition of the hardware sharing policies can be returned in the *cur_policy* value are defined in Table 11-73.

Table 11-73. Hardware policies returned in *cur_policy*

Value	Name	Description
0	Performance	The processor has its hardware resources configured to achieve maximum performance across all logical processors that share hardware with the logical processor the procedure was made on.
1	Fairness	The processor has its hardware resources configured to approximately achieve equal sharing of competing hardware resources among all the logical processors that share hardware with the logical processor the procedure was made on.
2	High-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a greater share of the competing hardware resources.
3	Exclusive High-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a greater share of the competing hardware resources. See "Set Current Hardware Resource Sharing Policy" on page 2:441 for differences between high-priority and exclusive high priority.
4	Low-priority	The processor has its hardware resources configured such that the logical processor this procedure was called on has a smaller share of the competing hardware resources. This occurs when a competing logical processor has itself set as high priority or exclusive high priority.
All Other Values		Reserved

The return value *num_impacted* specifies the number of logical processors impacted by the hardware sharing policy. The return value *la* returns the logical address of one of the logical processors impacted by the hardware sharing policy. The return value *la* is the same value and format of that is returned by the PAL_FIXED_ADDR procedure, see "Get Fixed Geographical Address of Processor" on page 2:382 for details.

If the caller is interested in identifying all the logical processors impacted by the hardware sharing policy, this procedure will need to be called a number of times equal to the value returned in *num_impacted* return value. For each subsequent call it needs to increment the 'proc_num' input argument.

The logical processor this procedure is made on can only return information about how the hardware sharing policy impacts logical processors it is sharing hardware resources with. For example a physical processor package may contain two multi-threaded cores. On this example implementation the hardware sharing policy only impacts the two threads on the core and this procedure would only return the two *la*'s of the threads on that core, but would not return the *la*'s of the threads on the other core. When this procedure was made on the other core, then that procedure call would return the *la*'s of the two threads on that core.

This procedure is only supported on processors that have multiple logical processors sharing hardware resources that can be configured. On all other processor implementations, this procedure will return the Unimplemented procedure return status.

PAL_SET_HW_POLICY

Set Current Hardware Resource Sharing Policy

Purpose: Sets the current hardware resource sharing policy of the processor.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Arguments:

Argument	Description
index	Index of PAL_SET_HW_POLICY within the list of PAL procedures.
policy	Unsigned 64-bit integer specifying the hardware resource sharing policy the caller is setting.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_SET_HW_POLICY procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
1	Call completed successfully but could not change the hardware policy since a competing logical processor is set in exclusive high priority
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

Description: This procedure is used to set the hardware resource sharing policy on the logical processor it is called on. The setting of this policy will impact other logical processors on the physical processor package. The logical processors impacted is returned by the PAL_GET_HW_POLICY procedure, see "PAL_GET_HW_POLICY" on page 49 for details.

The input argument *policy* selects the hardware policy the caller would like to set. The supported hardware policies are listed in Table 11-107 below. By default the hardware always sets the processor in the performance policy at reset.

Table 11-107. Processor Hardware Sharing Policies

Value	Name	Description
0	Performance	The processor has its hardware resources configured to achieve maximum performance across all logical processors.
1	Fairness	The processor configures hardware resources to approximately achieve equal sharing of competing hardware resources among all impacted logical processors.

Table 11-107. Processor Hardware Sharing Policies *(Continued)*

Value	Name	Description
2	High-priority	The processor configures hardware resources to provide the logical processor this procedure was called on a greater share of the competing hardware resources. All competing logical processors will get a smaller share of the competing hardware resources.
3	Exclusive High-priority	The processor configures hardware resources such that the logical processor this procedure was called on has a greater share of the competing hardware resources. All competing logical processors will get a smaller share of the competing hardware resources. This policy also ensures that no other competing logical processor can modify the hardware sharing policy until the logical processor that is in exclusive high priority releases exclusive high-priority by selecting a different policy.
All Other Values		Reserved

The caller must be aware of which logical processors are impacted by hardware policy changes, since making a call on one of the logical processors will impact all logical processors that share the same hardware resources. For example if the caller selects the high-priority policy on one logical processor A and then later in time selects fairness policy on one of the competing logical processors B, the procedure will take away high-priority status from logical processor A and change all impacted logical processors to the fairness policy without an error.

If a caller wants to ensure that high-priority will not be taken away from a logical processor, it can use the exclusive high-priority policy. This policy will return an error if any competing logical processor tries to change the hardware policy. This ensures that the caller can ensure a certain logical processor will retain high-priority status until that status is explicitly released by that logical processor.

This procedure is only supported on processors that have multiple logical processors sharing hardware resources that can be configured. On all other processor implementations, this procedure will return the Unimplemented procedure return status.

12. PAL_GET_PSTATE modification

Note: The PAL_GET_PSTATE call in Specification Change 7 of this specification update has been updated with the following changes:

1. Add a new type to the PAL_GET_PSTATE type Argument table:

   type: 3

   Description: The performance_index returned will correspond to the current instantaneous P-state of the logical processor, at the time of the procedure call.

2. Change the following sentence, which follows the PAL_GET_PSTATE type Argument table from:

   “For processors that belong to a software-coordinated dependency domain or a hardware-independent dependency domain, the PAL_GET_PSTATE procedure should always be called with type argument value of 0.”

   to:

   “For processors that belong to a software-coordinated dependency domain or a hardware-independent dependency domain, the PAL_GET_PSTATE procedure should always be called with type argument value of 0 or 3. Note that the performance_index returned for type=0 and type=3 will have identical values for these coordination domains.

This is because the most recent PAL_SET_PSTATE procedure call will always succeed in transitioning to the requested performance state for these coordination domains (see PAL_SET_PSTATE procedure description for additional details).”

13. PAL calling convention change to preserve floating-point registers

1. On page 2:288, Volume 2, Part I, replace Section 11.9.2.1.2, “Stacked Registers”, with:

   “This calling convention is intended for usage after memory has been made available, and for procedures which require memory pointers as arguments. The stacked registers are also used for parameter passing and local variable allocation. This convention conforms to the Itanium Software Conventions and Runtime Architecture Guide. Thus, procedures using the stacked register calling convention can be written in the C language. There are two exceptions to the runtime conventions.

   1. The first argument to the procedure must also be copied to GR28 prior to making the procedure call. This allows procedures written using both static and stacked register calling conventions to call a single PAL_PROC entrypoint. This should be accomplished by having the stacked register procedures call a stub module which copies GR32 to GR28, then call PAL_PROC. It is the responsibility of the caller to provide this stub. Please refer to Table 11-23 for a detailed list of the general register usage for the stacked register calling convention.
   2. Floating point registers 32-127 are preserved (rather than scratch, as in the normal Itanium Software Conventions), except on the PAL_TEST_PROC procedure. This allows OSs to avoid having to save and restore these registers around a stacked-convention PAL procedure call.”

2. On page 2:288, Volume 2, Part I, replace Section 11.9.2.2.4, “Floating-point Registers”, with:

   “Floating point registers 32-127 are preserved. PAL must either not use these, or must save and restore them, except on the PAL_TEST_PROC procedure, which may overwrite these registers without preserving them. The remainder of the floating-point register conventions are the same as those of the Itanium Software Conventions and Runtime Architecture Guide.”

3. Add the following sentence to the end of the PAL_TEST_PROC procedure of the PAL chapter in Volume 2, Part I:

   “PAL_TEST_PROC may overwrite floating-point registers 32-127 without restoring their values upon exit.”

14. Addition of shared error (se) bit to the processor state parameter (PSP)

1. In Figures 11-11 and 11-15, “Processor State Parameter”, Volume 2, Part I on pages 2:268 and 2:275, respectively:

   a. Add a separate field for bit 48, mark field as “se”.

   b. Change reserved bit field to 58:49.

2. Update Tables 11-5 and 11-8, “Processor State Parameter Fields”, Volume 2, Part I on pages 2:268 and 2:276, respectively:

   a. Add row for bit 48 as shown below:

Field Name	Bit	Description
se	48	Shared Error. Machine check corresponds to structure shared by multiple logical processors.

   b. Change row for bit 58:48 as shown below:

Field Name	Bit	Description
rsvd	58:49	Reserved

15. Enhancement to PAL_LOGICAL_TO_PHYSICAL

1. On page 2:335, Volume 2, Part I, “PAL_LOGICAL_TO_PHYSICAL” procedure, update the proc_number description to the following:

   “Signed 64-bit integer that specifies for which logical processor information is being requested. When this input argument is -1, information is returned about the logical processor on which the procedure call is made. When this input argument is zero, in addition to information about the first logical processor, log_overview contains the overview information as well. This input argument must be in the range of -1 up to one less than the number of logical processors returned by num_log in the log_overview return value.”

2. In the Returns table, delete the sentence from the log_overview description:

   “This value is only valid if the “proc_number” input argument was zero when the procedure was called, otherwise return zero.”

3. After the following sentence near the end of the fourth paragraph of the Description section:

   > “This procedure may be called from any logical processor on the physical processor package to gather information about all the logical processors.”

   add:

   > “It may also be called to get information about the logical processor on which the procedure is running.”

4. Change the num_log bulleted definition on page 2:336 to:

   “num_log – Total number of logical processors on this physical processor package that are enabled.”

5. Replace the paragraph after Figure 11-32 on page 2:336 from:

   > “As part of the processor boot flow, some testing of the processor occurs. There is a chance that a thread experienced a testing failure that did not allow it to successfully boot. Due to this reason, it is not ensured that num_log will always be equal to cpp multiplied by tpc.”

   to:

   > “It is not ensured that num_log will always be equal to cpp multiplied by tpc. This is possible if some logical processors are disabled through implementation-specific means.”

16. Change of maximum alignment requirements for PAL_COPY_INFO

1. On page 2:216 in Volume 2, Part I, “PAL_COPY_INFO” procedure, change the last sentence in the description section from:

   > “The 'buffer_align' return value must be a power of two between 4 KB and 256 KB.”

   to:

   > “The 'buffer_align' return value must be a power of two between 4 KB and 1 MB.”

17. Indication of variable P-states

1. On page 2:361 in Volume 2, Part I, add a new row to Processor Features, Table 11-54 of PAL_PROC_GET_FEATURES

Table 11-54. Processor Features

Bit	Class	Control	Description
39	Opt.	No	Variable P-state performance: A value of 1, indicates that a processor implements techniques to optimize performance for the given P-state power budget by dynamically varying the frequency, such that maximum performance is achieved for the power budget. A value of 0, indicates that P-states have no frequency variation or very small frequency variations for their given power budget. This feature may only be interrogated by PAL_PROC_GET_FEATURES. it may not be enabled or disabled by PAL_PROC_SET_FEATURES. The corresponding argument is ignored.

2. Modify the return value -2 of PAL_PSTATE_INFO from:

   "Invalid argument, or P-states are not supported on this implementation"

   to:

   "Invalid argument"

Note: The PAL_PSTATE_INFO call in Specification Change 7 of this specification update has been updated with the above change.

Specification Clarifications

1. Volume 2: PSR.dt serialization clarification

1. Volume 2, Part I, Section 3.3.2, Table 3-2:

   — On page 2:20, change the Serialization Required column for PSR.dt from:

   “data”

   to:

   “inst/data”

   and add a cross-reference to footnote c.

2. Volume 2: Unaligned debug fault clarification

1. Volume 2, Part I, Section 8.3, Debug vector page:

   — On page 2:175, add the following paragraph to the end of the Notes section:

   If unaligned accesses are being performed with debug faults enabled, this fault may be taken even though there is not a match for the address programmed in the breakpoint register. See Volume 2, Section 7.1.2, “Debug Address Breakpoint Match Conditions.”

3. Volume 3: Clarification on PSR requirements for `br.ia`/`rfi` instructions during PSR.is transition

1. `br.ia` instruction page (Volume 3, p. 3:18):

   a. Under “ia” bullet, add the following paragraph after the 3rd paragraph:

   Software must set PSR properly before branching to the IA-32 instruction set; otherwise processor operation is undefined. See Volume 2, Table 3-2, “Processor Status Register Fields” on page 2:19 for details.

   b. In the “Operation” section on page 3:22 under “case 'ia',” add below “`tmp_taken = 1;`”:

```
if (CR[IPSR].ic==0 || CR[IPSR].dt==0 || CR[IPSR].mc==1 ||
CR[IPSR].it==0)
     undefined_behavior();
```

2. `rfi` instruction page (Volume 3, p. 3:204):

   a. In the “Description” section, before the paragraph beginning “Software must issue a `mf` instruction...,” add the following paragraph:

   If IPSR.is is 1, software must set other IPSR fields properly for IA-32 instruction set execution; otherwise processor operation is undefined. See Volume 2, Table 3-2, “Processor Status Register Fields” on page 2:19 for details.

   b. In the “Operation” section:

   Add the following below, “`if (CR[IPSR].is == 1) {`”:

```
if (CR[IPSR].ic==0 || CR[IPSR].dt==0 || CR[IPSR].mc==1 ||
CR[IPSR].it==0)
     undefined_behavior();
```

3. Table 3-1, Volume 3: Pseudo-Code Functions chapter:

   a. On page 3:253, replace the bullet list of faults in the Operation column of the tlb_translate() row of the Pseudo-Code Functions table with this new bullet list:

- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault
- Data NaT Page Consumption fault
- Data Key Miss fault
- Data Key Permission fault
- Data Access Rights fault
- Data Dirty Bit fault
- Data Access Bit fault
- Data Debug fault
- Unaligned Data Reference fault
- Unsupported Data Reference fault

b. Replace the bullet list of faults in the Operation column of the tlb_translate_nonaccess() row of the Pseudo-Code Functions table with this new bullet list:

- Unimplemented Data Address fault
- Data Nested TLB fault
- Alternate Data TLB fault
- VHPT Data fault
- Data TLB fault
- Data Page Not Present fault
- Data NaT Page Consumption fault
- Data Access Rights fault (`fc` only)

4. Volume 3: Added Illegal Operation fault to fnma I-page

1. Volume 3, `fnma` instruction page:

   On page 3:81, add “Illegal Operation fault” to the list of interruptions in the Interruptions section.

5. Clarify INTA/XTP definition

1. On page 2:112, Volume 2, Part I, Section 5.8.4.3, “Interrupt Acknowledge (INTA) Cycle”:
   — Add the following sentence to the end of the 2nd paragraph:
   “Any memory operation to the INTA address other than a single byte load is undefined.”
2. On page 2:112, Volume 2, Part I, Section 5.8.4.4, “External Task Priority (XTP) Cycle”:
   — Add the following sentence to the end of the 1st paragraph:
   “Any memory operation to the XTP address other than a single byte store is undefined.”

6. Clarify VHPT insert rules

1. Volume 2, Part I, Section 4.1.8:

   — Replace the 2nd paragraph and insert new table on the bottom of page 2:58 with:

   The VHPT walker's inserts into the TC follow purge-before-insert rules similar to those for software inserts (see Table 4-1, “Purge Behavior of TLB Instructions,” on page 2:49). VHPT walker inserts into the DTC behave similar to `itc.d`; VHPT walker inserts into the ITC behave similar to `itc.i`. If an instruction reference results in a VHPT walk that misses in the data TLB, the DTC insert for the translation for the VHPT acts similar to an `itc.d`.

   As described in Section 4.1, “Virtual Addressing” on page 2:43, processors may optionally use VRN bits when searching for a matching translation for a memory reference (references other than inserts and purges). In processors that do use VRN bits for such searches, VHPT inserts may also use VRN bits in searching for overlapping entries. Thus, if a VHPT insertion overlaps a translation in the TC, but the VRN of the address being inserted does not match the VRN of the existing TC translation, the purge of the existing TC entry is optional. If a VHPT insertion overlaps a translation in a TR, but the VRN of the address being inserted does not match the VRN of the TR translation, the VHPT insertion is allowed, and a machine check is optional. In processors which do not use VRN bits when searching for a matching translation for a memory reference, the behavior of VHPT inserts is identical to that of software inserts (see Table 4-1, “Purge Behavior of TLB Instructions,” on page 2:49).

   If a VHPT insert overlaps with an existing TR entry and the VRN of the insertion matches the VRN of the existing TR entry (for example, if the translation being inserted is for a large page which overlaps with a small page translation in the TR), the VHPT insertion can be done, but a machine check must be raised. Software must not create overlapping translations in the VHPT that are larger than a currently existing TR translation.

   The behavior of VHPT inserts is summarized in Table 4-9.

Table 4-9. Behavior of VHPT Inserts

VHPT Inserts	VRN bits used for TLB searching		VRN bits not used for TLB searching
	VRN match	No VRN match	
VHPT insert overlaps TC entry	May insert[a] Must purge[b]	May insert May purge[c]	May insert Must purge
VHPT insert overlaps TR entry	May insert Must Machine Check[d]	May insert May Machine Check[e]	Must not insert Must Machine Check

a. May insert: indicates that the VHPT may perform an insert into the TC
b. Must purge: requires that all partially or fully overlapped translations are removed prior to the insert or purge operation.
c. May purge: indicates that a processor may remove partially or fully overlapped translations prior to the insert or purge operation. However, software must not rely on the purge.
d. Must Machine Check: indicates that a processor will cause a Machine Check abort.
e. May Machine Check: indicates that a processor may cause a Machine Check abort based on the implementation.

7. Adding FP-readers to support table

1. Volume 3, Table 5-5 - “Instruction Classes”, on page 3:352, add:

   “mem-writers-fp”

   to the row:

   “fr-readers”

8. cmpxchg clarifications

1. Volume 2, Section 7.1.2, page 2:134:

   — Add a new bullet list item at the bottom of the first bullet list, with this text:

   "The `cmp8xchg16` operands are treated as 16-byte datums for both read and write breakpoint matching, even though this instruction only reads 8 bytes."

2. `cmpxchg` I-page (Volume 3, page 3:40):

   — In first paragraph of the Description section, change this sentence from:

   "For `cmp8xchg16`, if the two are equal, then 8-bytes from GR r2 are stored at the specified address ignoring bit 3 (GR r3 & ~0x4), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR r3 & ~0x4) + 8)."

   to:

   "For `cmp8xchg16`, if the two are equal, then 8-bytes from GR r2 are stored at the specified address ignoring bit 3 (GR r3 & ~0x8), and 8 bytes from the Compare and Store Data application register (AR[CSD]) are stored at that address + 8 ((GR r3 & ~0x8) + 8)."

9. Add Illegal Operation fault

1. Volume 3, Part I, Chapter 3:

   a. On page 3:247, add a new function to Table 3-1:

   Function: instruction_implemented (inst)

   Operation: Implementation-dependent routine which returns TRUE or FALSE, depending on whether inst is implemented.

   b. Remove the row "long_branch_implemented" from Table 3-1.

2. Volume 3, `ld` I-page on page 3:131

   a. Add the following paragraph to the end of the Description section:

   "For the sixteen_byte_form, an Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See "Processor Identification Registers" on page 1:29 for details."

   b. Operation section, after:

```
if (size == 16) itype |= UNCACHE_OPT;
```

   add:

```
if (sixteen_byte_form &&!instruction_implemented(LD16))
    illegal_operation_fault();
```

3. Volume 3, `st` I-page on page 3:219

   a. Add the following paragraph to the end of the Description section:

   "For the sixteen_byte_form, an Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See "Processor Identification Registers" on page 1:29 for details."

   b. Operation section, after:

```
otype = (sttype == 'rel')? RELEASE: UNORDERED;
```

   add:

```
if (sixteen_byte_form &&!instruction_implemented(ST16))
    illegal_operation_fault();
```

4. Volume 3, `cmpxchg` I-page on page 3:41
   a. Add the following paragraph to the end of the Description section:

      "For `cmp8xchg16`, an Illegal Operation fault is raised on processor models that do not support the instruction. CPUID register 4 indicates the presence of the feature on the processor model. See "Processor Identification Registers" on page 1:29 for details."

   b. Operation section, after:

      ```
      if(PR[qp]) {
      ```

      add:

      ```
      if (sixteen_byte_form
      &&!instruction_implemented(CMP8XCHG16))
           illegal_operation_fault();
      ```

5. Volume 3, `brl` I-page on 3:26, Operation section:
   — Replace the following:

   ```
   if (!long_branch_implemented())
        illegal_operation_fault();
   ```

   with:

   ```
   if (!instruction_implemented(BRL))
        illegal_operation_fault();
   ```

10. Non-speculative reference for WBL attribute clarification

1. Volume 2, Part I, add a new Section 4.4.6.1, at the end of Section 4.4.6:

4.4.6.1 Limited Speculation and the WBL Physical Addressing Attribute

Processors are allowed to reference limited speculation pages (WBL pages) speculatively, in order to increase performance, but this speculation is limited to prevent speculative references to 4Kbyte physical pages for which there is no actual memory (which would cause spurious machine checks).

Processors must not make hardware-generated speculative references to a given WBL 4Kbyte page until a **verified reference** has been made. Processors may optionally implement storage to hold the addresses of WBL 4Kbyte pages for which verified references have been made and may make subsequent hardware-generated speculative references to these pages. Such pages are termed **verified pages**.

A verified reference is an instruction or data reference made to the page by an in-order execution of the program; that is, a reference which would have been made had the instructions from the program been fetched and executed one at a time. A hardware-generated speculative reference does not constitute a verified reference. Hardware-generated speculative references include:

- Instruction fetches when the processor has not yet determined whether prior branches were predicted correctly.
- Instruction fetches when the processor has not yet determined whether prior instructions will raise faults or traps.
- Data references by instructions when the processor has not yet determined whether prior branches were predicted correctly.
- Data references by instructions when the processor has not yet determined whether prior instructions will raise faults or traps.
- Hardware-generated instruction prefetch references.
- Hardware-generated data prefetch references.

- Eager RSE data references.

For an instruction fetch to constitute a verified reference, it must only be determined that an in-order execution of the program requires that the IP point to this address, independent of whether the instruction at this address will subsequently take a fault or interrupt.

For a data reference to constitute a verified reference, the instruction must meet one of the following requirements:

- It executes without any fault or interrupt
- It takes an Unaligned Data Reference fault
- It takes a Data Debug fault
- It takes an External interrupt, but if it had not taken an External interrupt, it would have met one of the above qualifications (execute without fault, take an Unaligned Data Reference fault, or take a Data Debug fault)

Data-speculative loads are treated the same as normal loads, and if an in-order execution of the program requires the execution of a data speculative load, it constitutes a verified reference. Control-speculative loads to limited-speculation pages always defer and thus never constitute verified references.

It is not necessary for a processor to determine whether a reference will complete without generating a machine check for it to be a verified reference. If software actually references a physical address which will cause a machine check, hardware may generate multiple speculative references to the same page, potentially causing multiple machine checks.

Processors may access verified pages normally, as they would WB pages, including the use of caching, pipelining, and hardware-generate speculative references to improve performance.

Calling the PAL_PREFETCH_VISIBILITY procedure forces the processor to clear the storage holding the addresses of verified pages.

2. Remove the two paragraphs from Volume 2, Part I, Section 4.4.6 that talk about limited speculation (the paragraphs beginning, “Limited speculation is used to improve performance...”, and “Unless a limited-speculation page is speculatively accessible,...”).
3. In footnote “d” in Table 4-12 on page 2:68, change this text from:

   “The processor may only issue hardware-generated speculative references to a 4K-byte physical page while the page is speculatively accessible.”

   to:

   “The processor may only issue hardware-generated speculative references to a 4K-byte physical page if it is a verified page.”
4. On page 2:76, Volume 2, Part I, Section 4.4.11.2, change these two paragraphs from:

   “When a non-speculative reference is made to a physical address with the WBL attribute, the 4K page containing that address becomes speculatively accessible. This allows the processor that made the non-speculative reference to subsequently make speculative references to this page. (See the description of limited speculation in Section 4.4.6, “Speculation Attributes” on page 2:70.)

   If the same physical memory is then to be accessed with the U attribute, software must first make all such addresses speculatively inaccessible and flush any cached copies from the cache. Otherwise, an uncacheable reference may hit in cache, causing a Machine Check abort.”

   to:

   “When a verified reference is made to a physical address with the WBL attribute, the 4K page containing that address becomes speculatively accessible. This allows the processor that made the verified reference to subsequently make speculative references to this page.

(See the description of limited speculation in Section 4.4.6.1, “Limited Speculation and the WBL Physical Addressing Attribute” on page 2:70.)

If the same physical memory is then to be accessed with the UC attribute, software must first cause all such 4K pages to no longer be verified pages and flush any cached copies from the cache. Otherwise, an uncacheable reference may hit in cache, causing a Machine Check abort.”

5. Volume 2, Part I, Section 4.4.11.2, bullet point 1 on page 2:76, change this paragraph from:

   “Call PAL_PREFETCH_VISIBILITY with the input argument trans_type equal to one to indicate that the transition is for physical memory attributes. This PAL call terminates the processor's rights to make speculative references to any limited speculation pages (i.e., it makes all WBL pages speculatively inaccessible - see the discussion on limited speculation in Section 4.4.6.)”

   to:

   “Call PAL_PREFETCH_VISIBILITY with the input argument trans_type equal to one to indicate that the transition is for physical memory attributes. This PAL call terminates the processor's rights to make speculative references to any limited speculation pages (i.e., it causes all WBL pages to no longer be verified pages - see the discussion on limited speculation in Section 4.4.6.1.)”

6. Volume 2, Part I, Section 4.4.11.2 on page 2:77, the very last paragraph of this section is not part of bullet point 5, but rather a summation of the bulleted sequence.

7. In Volume 2, Part I, Section 11.9.3 PAL Procedure Specification, PAL_PREFETCH_VISIBILITY (Page 2:358) Description, paragraph 4, last sentence should be changed from:

   “For the processor to make any speculative reference to a limited speculation page after this call, there must be a non-speculative reference made to that page after this call.”

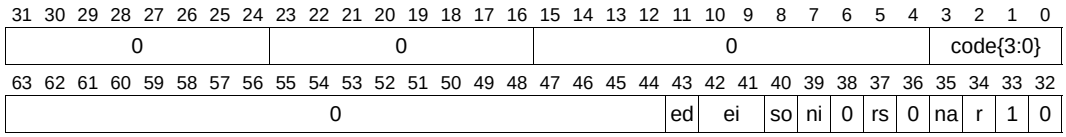
   to:

   “For the processor to make any speculative reference to a limited speculation page after this call, there must be a verified reference made to that page after this call. See the discussion on limited speculation in Section 4.4.6.1."

11. Dirty-bit fault ISR.code clarification

1. Volume 2, Part I, Section 8.3, Dirty-bit vector on page 2:160:

   a. Update diagram for ISR field to indicate that bits[3:0] represent ISR.code as shown below:



   b. Change following ISR statement on same page from:

      “ISR - The value for the ISR bits depend upon the type of access performed and are specified below. For mandatory RSE spill references, ISR.ed is always 0. For IA-32 memory references, ISR.ed, ei, ni, and rs are 0."

      to:

      “ISR - The value for the ISR bits depend upon the type of access performed and are specified below. For mandatory RSE spill references, ISR.ed is always 0. For IA-32 memory references, ISR.ed, ei, ni, and rs are 0. If the interruption was due to a non-access operation then the ISR.code bits {3:0} are set to indicate the type of the non-access instruction; otherwise they are set to 0."

c. Change following statement after the Notes section on the same page from:

"For probe.w.fault and probe.rw.fault the ISR.na bit is set"

to:

"For probe.w.fault and probe.rw.fault the ISR.na bit is set, and the ISR.code field is written with a value of 5."

12. FC data dependency ordering clarification

1. Volume 2, Part I, Section 4.4.7 on page 2:72, change the following sentence from:

"The flush cache instruction (`fc`, `fc.i`) instruction follows data dependency ordering. `fc` and `fc.i` are ordered with respect to previous and subsequent load, store, or semaphore instructions to the same line, regardless of the specified memory attribute."

to:

"The flush cache instruction (`fc`, `fc.i`) instruction follows data dependency ordering. `fc` and `fc.i` are ordered only with respect to previous load, store, or semaphore instructions to the same line, regardless of the specified memory attribute. Subsequent memory operations to the same line need not wait for prior `fc` or `fc.i` completion before being globally visible."

2. Volume 3, `fc` I-page (page 3:55), 5th paragraph, change the following sentence from:

"These instructions follow data dependency rules; they are ordered with respect to preceding and following memory references to the same line. `fc` and `fc.i` have data dependencies in the sense that any prior stores by this processor will be included in the flush operation."

to:

"These instructions follow data dependency ordering rules; they are ordered only with respect to previous load, store, or semaphore instructions to the same line. `fc` and `fc.i` have data dependencies in the sense that any prior stores by this processor will be included in the flush operation. Subsequent memory operations to the same line need not wait for prior `fc` or `fc.i` completion before being globally visible."

13. PAL_MC_DRAIN clarification

1. Volume 2, Part I, PAL_MC_DRAIN on page 2:339, change the first sentence of the Description section from:

"This call causes all outstanding transactions in the processor to be completed (for example, loads get their data returned, store updates are completed, and prefetches are either completed or cancelled)."

to:

"This call causes all outstanding transactions in the processor to be completed. For example:

i. Flushes (`fc`) invalidate the cache; lines that have been modified are written back (issued to the fabric) to memory before invalidation.

ii. Instruction cache coherence flushes (`fc.i`) invalidate lines and/or write them back to main memory, if this is required to make the instruction caches coherent with the data caches.

iii. Loads get their data returned.

iv. Stores either update the cache or issue transactions to the system fabric.

v. Prefetches are either completed or cancelled."

14. Add hint instructions to support table

1. Volume 3, page 3:356, Table 5-5:
   — Add the following to 'pr-readers-br' (in the appropriate alphabetical location):
   “hint.b”
2. Volume 3, page 3:357, Table 5-5:
   — Add the following to 'pr-readers-nobr-nomovpr' (in the appropriate alphabetical location):
   “hint.f, hint.i, hint.m, hint.x”

15. Clarify speculative operation fault handler requirements

1. On the Speculation vector (0x5700) page in Section 8.3 of Volume 2, Part I, page 2:174 add a “Notes” section below the ISR diagram that reads:
   “The Speculative Operation fault handler is required to perform the following steps:
   1. Read the predicates and the IIM, IIP, IPSR, and ISR control registers into scratch bank-0 general registers.
   2. Copy the IIP value to IIPA.
   3. Sign-extend the IIM value (from 21 bits to 64), shift it left by 4 bits, and add it to the IIP value.
   4. Set the IPSR.ri field to 0.
   5. Check whether either IPSR.tb (Taken Branch trap) or IPSR.ss (Single Step enable) is 1. If not, emulation is complete, so restore the predicates and rfi. If so, then the check instruction would have taken one of these traps instead of branching to its target, so this handler needs to branch directly to the appropriate trap handler instead of performing the `rfi` (see steps 6 - 7).
   6. If IPSR.tb was 1, then update ISR.code with its “tb” bit set to 1 and its “ss” bit also set to 1 if IPSR.ss was 1 and all other bits 0. Restore the predicates, execute a srlz.d, and branch to the taken branch vector (IVT offset 0x5f00).
   7. If IPSR.ss was 1 (but not IPSR.tb), then update ISR.code with its “ss” bit set to 1, and all other bits 0. Restore the predicates, execute a srlz.d, and branch to the single step vector (IVT offset 0x6000).”
2. In Table 5-7 “Interruption Vector Table (IVT)”, change the “Reserved” text in the Vector Name column of the offset 0x5800 row to “Reserved for software use” and attach a footnote to this entry. The text of the footnote should read:
   “Unlike the other Reserved IVT vectors, which may be defined in future revisions of the architecture, vector 0x5800 is permanently reserved. Software may use this vector for any purpose, such as placing in this area portions of other handlers that don't fit into their assigned vector.”
3. Add the following to the Speculation vector (0x5700) page, just below the list added by (A):
   “The Speculative Operation fault handler does not need to check for unimplemented instruction addresses. They will be checked automatically by processor hardware when the handler executes its `rfi`. If an emulated check instruction targets an unimplemented address and also needs to take a Single Step trap or Taken Branch trap (or both), the Unimplemented Instruction Address trap will not be raised until after the Single Step and/or Taken Branch trap has been handled, making it appear that the Unimplemented Instruction Address trap has the wrong priority. A Speculative Operation fault handler with this behavior is architecturally compliant.”

4. In Table 5-6 “Interruption Priorities” on page 2:94, add a footnote to the “Unimplemented Instruction Address trap” cell, which reads:

   “Unimplemented Instruction Address traps on emulated check instructions have a lower priority than Taken Branch trap and Single Step trap. See Speculation vector (0x5700) on page 2:174.”

16. Clarify role of PMC.ev bit as implementation-specific

1. Volume 2, Part I, Section 7.2.1, Table 7-4, page 2:137, change the ev row from:

   “External visibility - When 1, an external notification (such as a pin or transaction) is provided whenever the monitor overflows. Overflow occurs when a carry out from bit W-1 is detected.”

   to:

   “External visibility - When 1, an external notification (such as a pin or transaction) may be provided, dependent upon implementation, whenever the monitor overflows. Overflow occurs when a carry out from bit W-1 is detected.”

17. Relax IA-32 Application Registers Reserved/Ignored checking

1. Update Section 3.1.8.6 to:

3.1.8.6 Compare and Store Data register (CSD – AR 25)

The Compare and Store Data register is a 64-bit register that provides data to be stored by the Itanium st16 and cmp8xchg16 instructions, and receives data loaded by the Itanium ld16 instruction.

For implementations that do not support the ld16, st16 and cmp8xchg16 instructions, bits 61:60 may be optionally implemented. This means that on move application register instructions the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads. For implementations that do support the ld16, st16 and cmp8xchg16 instructions, all bits of CSD are implemented.

For IA-32 execution, this register is the IA-32 Code Segment Descriptor. See Section 6.2.3, “IA-32 Segment Registers” on page 1:117.

2. Update the av and ig rows of Table 6-2, page 1:108, Volume 1.

Table 6-2. IA-32 Segment Register Fields

Field	Bits	Description
av	60	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field is available for IA-32 software use and there will be no future use for this field. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.
ig	61	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by IA-32 software. For Itanium instructions, implementations which do not support the `ld16`, `st16` and `cmp8xchg16` instructions can either ignore writes and return zero on reads, or write the value and return the last value written on reads. Implementations which do support these instructions write the value and return the last value written on reads.

3. Update the following three rows of both Table 6-5 of Volume 1 on page 1:113 and Table 10-3 of Volume 2 on page 2:218:

Table 6-5. IA-32 EFLAGS Register Fields

EFLAG[a]	Bits	Description
	1	Ignored – For IA-32 instructions, writes are ignored, reads return one. For Itanium instructions, the implementation can either ignore writes and return one on reads; or write the value, and return the last value written on reads.
	3,5, 15	Ignored – For IA-32 instructions, writes are ignored, reads return zero. For Itanium instructions, the implementation can either ignore writes and return zero on reads, or write the value and return the last value written on reads.
	63:22	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads.

a. On entry into the IA-32 instruction set all bits may be read by subsequent IA-32 instructions, after exit from the IA-32 instruction set these bits represent the results of all prior IA-32 instructions. None of the EFLAG bits alter the behavior of Itanium instruction set execution.

4. Change the last sentence of Volume 1, Section 6.2.4, 1st paragraph, change from:

   "When Itanium architecture-based software loads this application register (AR24), a Reserved Register/Field fault will be raised if a non-zero value is written into bits listed as reserved. See Section 10.3.2, "IA-32 System EFLAG Register" on page 2:235."

   to:

   See Table 6-5 "IA-32 EFLAGS Register Fields" for the behavior on IA-32 and Itanium instruction reads/writes to this application register. For details on system flags in the IA-32 EFLAGS register, see Section 10.3.2, "IA-32 System EFLAG Register" on page 2:235.

5. Change the last paragraph of Section 10.3.2, Volume 2, on page 2:217, change from:

   "When Itanium architecture-based software loads this application register (AR24), a Reserved Register/Field fault will be raised if a non-zero value is written into bits listed as reserved."

   to:

   "See Table 10-3 "IA-32 EFLAGS Field Definition" for the behavior on IA-32 and Itanium instruction reads/writes to this application register."

6. Change the last paragraph of Section 6.2.5.3, Volume 1, on page 1:117, change from:

   "Software must ensure that FCR and FSR are properly loaded for IA-32 numeric execution before entering the IA-32 instruction set. When Itanium architecture-based software loads these application registers (AR21 and AR28), a Reserved Register/Field fault will be raised if a non-zero value is written to bits listed as reserved. No field encoding values will be verified when these registers are written."

   to:

   "Software must ensure that FCR and FSR are properly loaded for IA-32 numeric execution before entering the IA-32 instruction set. For Itanium instructions accessing ignored fields, the implementation can either ignore writes and return the specified constant on reads, or write the value and return the last value written on reads. For Itanium instructions accessing reserved fields, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault), and return the last value written on reads."

7. Change the last paragraph of Section 6.2.5.4, Volume 1, on page 1:118, change from:

   "When Itanium architecture-based software loads these application registers (AR29 and AR30), a Reserved Register/Field fault will be raised if a non-zero value is written to bits

listed as reserved. No field encoding values will be verified when these registers are written."

to:

"For Itanium instructions, the implementation can either raise Reserved Register/Field faults on non-zero writes to the reserved fields, or write the value and return the last value written on reads."

8. Update the following two rows of Table 10-4 of Volume 2 on page 2:222

Table 10-4. IA-32 Control Register Field Definition

Field	Intel® Itanium® State	Bits	Description
ignored		9:15, 17, 19:28	Ignored – This field is ignored by the processor during IA-32 instruction set execution. This field may have a future use and should be set to zero by IA-32 software. For Itanium instructions, the implementation can either ignore the writes and return zero on reads, or write the value and return the last value written on reads.
reserved		43:63	This field is reserved for IA-32 instructions – reads return zeros and non-zero writes causes IA_32_Exception (General Protection) faults. For Itanium instructions, the implementation can either raise Reserved Register/Field fault on non-zero writes and return zero on reads, or write the value (no Reserved Register/Field fault) and return the last value written on reads.

18. Relax ordering constraints for VHPT walks

1. After the second paragraph in Section 4.4.7 of Volume 2, Part I, add the following paragraph:

   For VHPT walks, visibility is defined by the memory read(s) which retrieves translation information, and the associated insertion of the translation into the TLB. VHPT walks are performed asynchronously with respect to program execution, and each walker VHPT read (which appears as though it were performed atomically) is made visible at some single point in the program order. Ordering constraints from table 4-15 do not prevent VHPT walks from becoming visible.

2. In the second paragraph in Section 4.4.7, delete the footnote that reads:

   "1. Although VHPT walks are performed somewhat asynchronously with respect to program execution, each VHPT read appears as though it were performed atomically, at some single point in the program order."

3. Change the first sentence of footnote 1 on page 2:81 from:

   "1. This includes all types of loads (ld and ld.acq), and RSE and VHPT memory reads."

   to:

   "1. This includes all types of loads (ld and ld.acq), and RSE memory reads."

4. Delete the following line from Volume 2, Part II, Section 5.3.3, "VHPT updates":

   "The VHPT walker uses unordered load semantics to access the in-memory VHPT."

19. Clarify illegal operation fault behavior for predicated off reserved ops

On page 2:92 of Volume 2, Part I, Table 5-6:

1. Add a superscript numeral after "Illegal Operation fault" to indicate that there is a footnote below.

2. Add this footnote at the bottom of the page:

   "Illegal Operation faults can be taken for certain predicated off reserved opcodes. For details, refer to Volume 3, Section 4.1"

20. Clarify opcode hint fields in encodings

All changes listed are with respect to Volume 3, Chapter 4, “Instruction Formats”.

1. Add the following paragraph to the end of Section 4.1, just after the existing “Ignored (white space) Fields” paragraph:

   “Unused opcode hint extension values (white color entries in Hint Completer tables) should not be used by software. Processors must perform the architected functional behavior of the instruction independent of the hint extension value (whether defined or unused), but different processor models may interpret unused opcode hint extension values in different ways, resulting in undesirable performance effects.”

2. Section 4.4.1, just before Table 4-39 “Load Hint Completer”, change:

   “opcode extension field in bits 29:28 (hint) which encodes locality hint information.”

   to:

   “cache locality opcode hint extension field in bits 29:28 (hint).”

3. Section 4.4.2, just before Table 4-41 “Line Prefetch Hint Completer”, change:

   “opcode extension field in bits 29:28 (hint) which encodes locality hint information”

   to:

   “cache locality opcode hint extension field in bits 29:28 (hint)”

4. Section 4.4.3, add the following to the end of the first (and only) paragraph:

   “These instructions have the same cache locality opcode hint extension field in bits 29:28 (hint) as load instructions. See Table 4-39 on p. 3:291.”

5. Section 4.5.1, just before Table 4-51 “Sequential Prefetch Hint Completer”, change:

   “All of the branch instructions have a 1-bit opcode extension field, p, in bit 12 which provides a sequential prefetch hint.”

   to:

   “All of the branch instructions have a 1-bit sequential prefetch opcode hint extension field, p, in bit 12.”

6. Section 4.5.1, just before Table 4-52 “Branch Whether Hint Completer”, change:

   “The IP-relative and indirect branch instructions all have a 2-bit opcode extension field in bits 34:33 (wh) which encodes branch prediction “whether” hint information as shown in Table 4-52. Indirect call instructions have a 3-bit opcode extension field in bits 34:32 (wh) for “whether” hint information as shown in Table 4-53.”

   to:

   “The IP-relative and indirect branch instructions all have a 2-bit branch prediction “whether” opcode hint extension field in bits 34:33 (wh) as shown in Table 4-52. Indirect call instructions have a 3-bit “whether” opcode hint extension field in bits 34:32 (wh) as shown in Table 4-53.”

7. Section 4.5.1, just before Table 4-54 “Branch Cache Deallocation Completer”, change:

   “opcode extension field in bit 35 (d) which encodes a branch cache deallocation hint”

   to:

   “branch cache deallocation opcode hint extension field in bit 35 (d)”

8. Section 4.5.2, just before Table 4-56 “Branch Importance Hint Completer”, change:

   “opcode extension field in bit 35 (ih) which encodes a branch importance hint.”

   to:

   “branch importance opcode hint extension field in bit 35 (ih).”

9. Section 4.5.2, just before Table 4-57 “IP-Relative Predict Whether Hint Completer”, change:

   “opcode extension field in bits 4:3 (wh) which encodes branch prediction “whether” hint information”

   to:

   “branch prediction “whether” opcode hint extension field in bits 4:3 (wh)”

10. Section 4.5.2, just before Table 4-58 “Indirect Predict Whether Hint Completer”, change:

    “opcode extension field in bits 4:3 (wh) which encodes branch prediction “whether” hint information”

    to:

    “branch prediction “whether” opcode hint extension field in bits 4:3 (wh)”

21. Clarify speculative operation fault handler requirements

All changes listed are with respect to Volume 2, Part I.

1. On the Speculation vector (0x5700) page in Section 8.3, add a “Notes” section below the ISR picture, which reads:

   NOTES:

   1. The Speculative Operation fault handler is required to perform the following steps:
      a. Read the predicates and the IIM, IIP, IPSR, and ISR control registers, into scratch bank-0 general registers.
      b. Copy the IIP value to IIPA.
      c. Sign-extend the IIM value (from 21 bits to 64), shift it left by 4 bits, add it to the IIP value, and write this value back into IIP.
      d. Set the IPSR.ri field to 0.
      e. Check whether either IPSR.tb (Taken Branch trap) or IPSR.ss (Single Step enable) is 1. If not, emulation is complete, so restore the predicates and rfi. If so, then the check instruction would have taken one of these traps instead of branching to its target, so this handler needs to branch directly to the appropriate trap handler instead of performing the rfi (see steps 6 - 7).
      f. If IPSR.tb was 1, then update ISR.code with its “tb” bit set to 1 and its “ss” bit also set to 1 if IPSR.ss was 1, and all other bits 0. Restore the predicates, execute a srlz.d, and branch to the taken branch vector (IVT offset 0x5f00).
      g. If IPSR.ss was 1 (but not IPSR.tb), then update ISR.code with its “ss” bit set to 1, and all other bits 0. Restore the predicates, execute a srlz.d, and branch to the single step vector (IVT offset 0x6000).

2. Add the following to the Speculation vector (0x5700) page, just below the “Notes”:

   “The Speculative Operation fault handler does not need to check for unimplemented instruction addresses. They will be checked automatically by processor hardware when the handler executes its `rfi`. If an emulated check instruction targets an unimplemented address and also needs to take a Single Step trap or Taken Branch trap (or both), the Unimplemented Instruction Address trap will not be raised until after the Single Step and/or Taken Branch trap has been handled, making it appear that the Unimplemented Instruction Address trap has the wrong priority. A Speculative Operation fault handler with this behavior is architecturally compliant.”

3. In Table 5-7 on page 2:96 “Interruption Vector Table (IVT)”, change the “Reserved” text in the Vector Name column of the offset 0x5800 row to “Reserved for software use” and attach a footnote to this entry. The text of the footnote should read:

   “Unlike the other Reserved IVT vectors, which may defined in future revisions of the architecture, vector 0x5800 is permanently reserved.

   Software may use this vector for any purpose, such as placing in this area portions of other handlers that don't fit into their assigned vector.”

4. In Table 5-6 “Interruption Priorities”, add a footnote to the “Unimplemented Instruction Address trap” cell, which reads:

   “Unimplemented Instruction Address traps on emulated check instructions have a lower priority than Taken Branch trap and Single Step trap. See Speculation vector (0x5700) on p. 2:174.”

22. PAL_CACHE_FLUSH clarification

1. Change the third paragraph of the description on page 2:298 from:

   “When the caller specifies to make local instruction caches coherent with the data caches, this procedure will ensure that the local instruction caches will see the effects of stores of instruction code done on the processor. Refer to Section 4.4.3, “Cacheability and Coherency Attribute” on page 2:65 for more information on stores and their coherency requirements with local instruction caches.”

   to:

   “When the caller specifies to make local instruction caches coherent with the data caches, this procedure will ensure that the instruction caches on the processor that this procedure call was made, will see the effects of stores to instruction code performed by this processor. This procedure is not required to ensure coherency of instruction caches on other processors in the system when this input argument is used. Refer to Section 4.4.3, “Cacheability and Coherency Attribute” on page 2:65 for more information on stores and their coherency requirements with local instruction caches.”

23. Interruption serialization clarification

1. Replace the second and third paragraphs of Section 3.3.3, Volume 2, Part II, on page 2:413 with:

   “When an interruption is delivered and before execution begins in the interruption handler, the processor hardware automatically performs an instruction and data serialization on all “in-flight” resources. As described in Section 3.3.1 and Section 3.3.2 above, the following resources determine the execution environment of the interruption handler:

   CR[IVA] – determines new IP

   CR[DCR].be – determines new value of PSR.be

   CR[DCR].pp – determines new value of PSR.pp

   PSR.ic – determines whether interruption collection is enabled

   RR[7:0] – determines new value of CR[ITIR] and CR[IHA]

   CR[PTA] – determines new value of CR[IHA]

   Although these resources are guaranteed to be serialized prior to interruption handler execution, there is no guarantee that they will be serialized prior to the determination of the handler's execution environment. If there is a value in-flight for any of these resources at the time of interruption delivery, either the old or new value may be used to generate the values of IP, PSR, CR[ITIR] and CR[IHA] seen by the handler.

   As a result, if the handler requires the latest value of the listed resources to determine its execution environment, software must ensure that external interrupts are disabled and that no instruction or data references will take an exception until the resource updates have been appropriately serialized. Typically, the code toggling these resources is mapped by an instruction translation register to avoid TLB related faults.

   Note that CR[IPSR] is guaranteed to get the latest value of the PSR on an interruption, even if there are PSR updates in-flight that have not been previously serialized by software.”

24. Clarification of Local ID fields

1. Section 5.8.3.1, “Local ID (LID - CR64)”, Volume 2, Part I, on page 2:105, change the second sentence of the id/eid field description in Table 5-10 from:

   “The high order bits of id and the eid field correspond to a unique address of the local system bus within the entire system.”

   to:

   “The eid field and the higher order bits of the id field correspond to a unique address of the local system bus within the entire system.”

25. PSR.ri field clarification

1. On page 2:23, Volume 2, Part I, in Table 3-2, change the second paragraph of the PSR.ri description from:

   “When restarting instructions with `rfi`, this field specifies which instruction(s) in the bundle are restarted.”

   to:

   “When restarting instructions with `rfi`, this field in IPSR specifies which instruction(s) in the bundle are restarted.”

26. Missing instructions in the Ordering Semantics table

1. On page 2:70, Volume 2, Part I, Table 4-14, add `cmp8xchg16.rel` to the Release row.
2. In the same table, in the Acquire row, add `cmp8xchg16.acq`.

27. `ttag` clarification

1. On page 2:54, Volume 2, Part I, remove “unique” from the following sentence in the second paragraph of Section 4.1.6.

   “The `ttag` instruction is only useful for long-format hashing, and generates a unique 64-bit ti/tag identifier that the processor’s VHPT walker will check when it looks up a given virtual address and region identifier.”

   to:

   “The `ttag` instruction is only useful for long-format hashing, and generates a 64-bit ti/tag identifier that the processor’s VHPT walker will check when it looks up a given virtual address and region identifier.”

28. Missing fault for `mov cr`

1. On page 3:152, Volume 3, Part I, the mov cr I-page, add “Unimplemented Data Address Fault” to the list of interruptions.

29. Clarifications to `tak` instruction

1. On page 3:227, Volume 3, Part I, the `tak` I-page, change the description sentence from:

   “If a matching present translation is found the protection key of the translation is placed in GR r1.”

   to:

   “If a matching present translation is found the protection key of the translation is placed in bits 31:8 of GR r1.”

2. In Table 3-1 on page 3:251 of Volume 3, Part I, the tlb_access_key(`vaddr`) row, change the Operation field to:

   “This function returns, in bits 31:8, the access key from the TLB for the entry corresponding to `vaddr` and `itype`; bits 63:32 and 7:0 return 0. If `vaddr` is an

unimplemented virtual address, or a matching present translation is not found, the value 1 is returned."

30. Clarification for the dirty bit during TLB insertion

1. On page 2:45, Volume 2, Part I, in the Dirty Bit row of Table 4-2, change the last sentence of the Description from:

   "The processor does not update the Dirty bit on a write reference."

   to:

   "The processor does not update the Dirty bit on a store or semaphore reference."

Documentation Changes

1. Update IA-32 CPUID I-Page

Volume 3: Updated IA-32 CPUID Instruction

CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers, according to the input value entered initially in the EAX register.

Description

Returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The information returned is selected by entering a value in the EAX register before the instruction is executed. Table 2-4 shows the information returned, depending on the initial value loaded into the EAX register.

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction.

The information returned with the CPUID instruction is divided into two groups: basic information and extended function information. Basic information is returned by entering an input value starting at 0 in the EAX register; extended function information is returned by entering an input value starting at 80000000H. When the input value in the EAX register is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning basic information. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX return value for basic information. When the input value in the EAX register is 80000000H, the processor returns the highest value the CPUID instruction recognizes in the EAX register for returning extended function information. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX return value for extended function information.

The CPUID instruction can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

Table 2-4. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0	EAX	Maximum CPUID Input Value
	EBX	756E6547H “Genu” (G in BL)
	ECX	6C65746EH “ntel” (n in CL)
	EDX	49656E69H “inel” (i in DL)

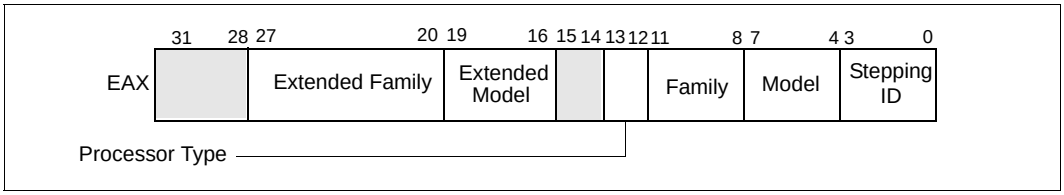
Table 2-4. Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
1H	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Bits 7-0: Brand Index[a] Bits 15-8: CLFLUSH line size (Value * 8 = cache line size in bytes) Bits 23-16: Number of logical processors per physical processor Bits 31-24: Local APIC ID[b]
	ECX	Reserved
	EDX	Feature Information (see Table 2-5)
2H	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information
	Extended Function CPUID Information	
8000000H	EAX	Maximum Input Value for Extended Function CPUID Information
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
8000001H	EAX	Extended Processor Signature and Extended Feature Bits. (Currently reserved.)
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
8000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
8000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued

a. This field is not supported for processors based on Itanium architecture, zero (unsupported encoding) is returned.
b. This field is invalid for processors based on Itanium architecture, reserved value is returned.

When the input value is 1, the processor returns version information in the EAX register (see Figure 2-3). The version information consists of an Intel architecture family identifier, a model identifier, a stepping ID, and a processor type.

Figure 2-3. Version Information in Registers EAX



If the values in the family and/or model fields reach or exceed FH, the CPUID instruction will generate two additional fields in the EAX register: the extended family field and the extended model field. Here, a value of FH in either the model field or the family field indicates that the extended model or family field, respectively, is valid. Family and model numbers beyond FH range from 0FH to FFH, with the least significant hexadecimal digit always FH.

See AP-485, *Intel® Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on identifying Intel architecture processors.

When the input value in EAX is 1, three unrelated pieces of information are returned to the EBX register:

- Brand index (low byte of EBX) – this number provides an entry into a brand string table that contains brand strings for IA-32 processors. Please refer to AP-485, *Intel® Processor Identification and the CPUID Instruction* (Order Number 241618) for information on brand indices.

  Note: The Brand index field is not supported for processors based on Itanium architecture, zero (unsupported encoding) is returned.

- CLFLUSH instruction cache line size (second byte of EBX) – this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field is valid only when the CLFSH feature flag is set.

- Local APIC ID (high byte of EBX) – this number is the 8-bit ID that is assigned to the local APIC on the processor during power up.

  Note: The local APIC ID field is invalid for processors based on the Itanium architecture, reserved value is returned. Software should check the feature flags to make sure they are not running on processors based on the Itanium architecture before interpreting the return value in this field.

When the EAX register contains a value of 1, the CPUID instruction (in addition to loading the processor signature in the EAX register) loads the EDX register with the feature flags. The feature flags (when a Flag = 1) indicate what features the processor supports. Table 2-5 lists the currently defined feature flag values.

A feature flag set to 1 indicates the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags.

Table 2-5. Feature Flags Returned in EDX Register

Bit	Mnemonic	Description
0	FPU	**Floating Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4Mbyte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2 Mbyte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.

Table 2-5. Feature Flags Returned in EDX Register (Continued)

Bit	Mnemonic	Description
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model-specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default – some processors permit the APIC to be relocated).
10	Reserved	Reserved.
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**PTE Global Bit.** The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported.
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address.
17	PSE-36	**32-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	NX	Execute Disable Bit.
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities.
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.

Table 2-5. Feature Flags Returned in EDX Register (Continued)

Bit	Mnemonic	Description
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Hyper-Threading Technology.** The processor implements Hyper-Threading technology.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Processor based on the Intel Itanium architecture	The processor is based on the Intel Itanium architecture and is capable of executing the Intel Itanium instruction set. IA-32 application level software MUST also check with the running operating system to see if the system can also support Itanium architecture-based code before switching to the Intel Itanium instruction set.
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

When the input value is 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers. The encoding of these registers is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors.

Please see the processor-specific supplement for further information on how to decode the return values for the processors internal caches and TLBs.

CPUID performs instruction serialization and a memory fence operation.

CPUID—CPU Identification (Continued)

Operation

```
CASE (EAX) OF
   EAX = 0H:
        EAX ← Highest input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
   BREAK;
   EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor Type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Always zero for processors based on Itanium
architecture *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Number of logical processors per physical processor;
        EBX[31:24] ← Initial APIC ID; (* Reserved for processors based on Itanium
architecture *)
        ECX ← Reserved;
        EDX ← Feature flags;
   BREAK;
   EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
   BREAK;
   EAX = 80000000H:
        EAX ← Highest extended function input value understood by CPUID;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
   BREAK;
   EAX = 80000001H:
        EAX ← Extended Processor Signature and Feature Bits; (* Currently Reserved *)
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
   BREAK;
   EAX = 80000002H:
        EAX ← Processor Name;
        EBX ← Processor Name;
        ECX ← Processor Name;
        EDX ← Processor Name;
   BREAK;
   EAX = 80000003H:
        EAX ← Processor Name;
        EBX ← Processor Name;
        ECX ← Processor Name;
        EDX ← Processor Name;
   BREAK;
   EAX = 80000004H:
        EAX ← Processor Name;
        EBX ← Processor Name;
```

```
            ECX ← Processor Name;
            EDX ← Processor Name;
        BREAK;
        DEFAULT: (* EAX > highest value recognized by CPUID *)
            EAX ← Reserved, Undefined;
            EBX ← Reserved, Undefined;
            ECX ← Reserved, Undefined;
            EDX ← Reserved, Undefined;
        BREAK;
    ESAC;

    memory_fence();
    instruction_serialize();
```

Flags Affected

None.

Additional Itanium System Environment Exceptions

Itanium Reg Faults	NaT Register Consumption Abort.

Exceptions (All Operating Modes)

None.

Intel Architecture Compatibility

The CPUID instruction is not supported in early models of the Intel486 processor or in any Intel architecture processor earlier than the Intel486 processor. The ID flag in the EFLAGS register can be used to determine if this instruction is supported. If a procedure is able to set or clear this flag, the CPUID is supported by the processor running the procedure.

2. PAL_BUS_GET/SET_FEATURES fix

1. Volume 2, Part I, Table 11-25.
   — On page 2:296, change the following bit 52 description from:

   “Enable a bus cache line replacement transaction when a cache line in the shared state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.”

   to:

   “Enable a bus cache line replacement transaction when a cache line in the shared or exclusive state is replaced from the highest level processor cache and is not present in the lower level processor caches. When 0, no bus cache line replacement transaction will be seen on the bus. When 1, bus cache line replacement transactions will be seen on the bus when the above condition is detected.”

3. PAL_COPY_PAL update

1. Volume 2, Part I, page 2:317:
   a. Change the following argument in the Arguments section of the PAL_COPY_PAL procedure from:

   “processor - Unsigned integer denoting whether the call is being made on the boot processor or an application processor.”

to:

"copy_option - Unsigned integer indicating whether relocatable PAL code and PAL PMI code should be copied from firmware address space to main memory."

b. Change the following sentences in the first paragraph of the Description section of PAL_COPY_PAL from:

"This procedure also updates the PALE_PMI entrypoint in hardware. If the call in made on a application processor the copy is not performed. The processor argument denotes whether the call is made on the boot processor (value of 0) or an application processor (value of 1)."

to:

"A value of 0 for the copy_option indicates that the relocation should be performed; a value of 1 indicates that the relocation should not be performed. This procedure also updates the PALE_PMI entrypoint in hardware."

4. Fixing X-Unit text correction

1. Volume 3, Section 4.7.4 "Nop/Hint (X-Unit)", Table 4-73 on page 3:332, change:

   "nop.m" to "nop.x"

   "hint.m" to "hint.x"

5. PAL_CACHE_SHARED_INFO text correction

1. Volume 2, Part I, page 2:311:

   For the entire PAL_CACHE_SHARED_INFO pages, change any instance of "proc_n_log_info" to "proc_n_cache_info".

6. PAL_CACHE_FLUSH minor code sequence fix

1. Volume 2, Part I, page 2:431, make a change to the assembly code in Section 5.1.1.3 (first line of assembly code). The code is trying to address region register 2, but indexed it incorrectly.

   Change from: `mov r2 = 2`

   to: `movl r2 = (2 << 61)`

7. PAL_GET_PROC_FEATURES table fix

1. Volume 2, Part I, PAL_GET_PROC_FEATURES table on page 2:361, fix bit 40.

   Change from:

   Bit: 40-0

   Class: N/A

   Control: N/A

   Description: reserved

   to:

   Bit: 40

   Class: N/A

   Control: N/A

   Description: reserved

8. Correct the role of X-resources during MCA

1. Volume 2, Section 11.3.1.1, change the XIP, XPSR, XFS bullet from:

   "XIP, XPSR, XFS: interruption resources implemented to store information about the IP, PSR and IFS when the machine check occurred. A model-specific version of the `rfi`

instruction must also be implemented to restore the machine context from these resources."

to:

"XIP, XPSR, XFS: interruption resources implemented to store information about the IIP, IPSR and IFS when the machine check occurred. A model-specific version of the `rfi` instruction must also be implemented to restore the machine context from these resources."

9. Clarification on the short format VHPT

1. Volume 2, Part II, Section 5.3, "Virtual Hash Page Table", change the second sentence in "Short" bullet from:

   "The short format VHPT cannot use protection keys (there are not enough PTE bits for that)."

   to:

   "The short format VHPT does not contain protection key information (there are not enough PTE bits for that)."

10. Floating-point correction

1. Volume 1, Section 5.4.1, at the end of the 2nd paragraph, change:

   "For example, dividing an SNaN by zero causes an invalid operation exception (due to the SNaN) and not a zero-divide exception; the exception disabled result is the QNaN indefinite, not infinity."

   to:

   "For example, dividing an SNaN by zero causes an invalid operation exception (due to the SNaN) and not a zero-divide exception; the exception disabled result is the quieted version of the SNaN, not infinity."

11. Clarify effect of sending IPI to non-existent processor

1. Volume 2, Part I, Section 5.8.4:, add the following paragraph to the end of Section 5.8.4, just before Section 5.8.4.1:

   "Any memory operation targeted at the lower half of the Processor Interrupt Block which does not correspond to any actual processor is undefined."

12. Add a new instruction class

1. In Table 5-5, "Instruction Classes", in Volume 3 on page 3:352 create and add the following class:

   Class: non-access Events/Instructions: fc, lfetch, probe-all, tpa, tak.

13. Updated RAW Dependence Table

1. In Section 5.3.5, add a new rule just after rule 7:

   "Rule 8. CR[TPR] has a RAW dependency only between mov-to-CR-TPR and mov to psr.l or ssm instructions that set PSR.i, PSR,pp or PSR.up."

2. In Volume 2, Section 5.8.3.3, change this paragraph:

   "To ensure that new priority levels are established by a given point in program execution (e.g., before PSR.i is set to 1), software must perform a data serialization operation after a TPR write and prior to that point. A data serialization operation must be performed after TPR is written and before IVR is read to ensure that the reported IVR vector is correctly masked. The TPR fields are described in Figure 5-8 and Table 5-11."

   to:

"To ensure that new priority levels are established by a given point in program execution, software must perform a data serialization operation after a TPR write and prior to that point. For example, if PSR.i is subsequently set to 1, thus enabling interrupts, and the new priority levels need to be in place before this enabling, a data serialization must be performed prior to the setting of PSR.i. Similarly, if PSR.pp or PSR.up is set to 1, potentially enabling performance monitor interrupts, and the new priority levels need to be in place before this enabling, a data serialization must be performed. (Note that there's no dependence between writing TPR and then changing the PSR for any other bits in the PSR than these.) A data serialization operation must be performed after TPR is written and before IVR is read to ensure that the reported IVR vector is correctly masked. The TPR fields are described in Figure 5-8 and Table 5-11."

3. Replace Table 5-2 "RAW Dependencies Organized by Resource" in Volume 3 with the following (Change bars have been kept to help identify modifications):

Table 5-2. RAW Dependencies Organized by Resource

Resource Name	Writers	Readers	Semantics of Dependency
ALAT	chk.a.clr, **mem-readers-alat**, **mem-writers**, **invala-all**	**mem-readers-alat**, **mem-writers**, **chk-a**, invala.e	none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi	br.call, brl.call, br.ia, br.ret, cover, flushrs, loadrs, **mov-from-AR-BSP**, rfi	impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, **mov-from-AR-BSPSTORE**	impliedF
AR[CCV]	**mov-to-AR-CCV**	br.ia, **cmpxchg**, **mov-from-AR-CCV**	impliedF
AR[CFLG]	**mov-to-AR-CFLG**	br.ia, **mov-from-AR-CFLG**	impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**	br.ia, cmp8xchg16, **mov-from-AR-CSD**, st16	impliedF
AR[EC]	**mod-sched-brs**, br.ret, **mov-to-AR-EC**	br.call, brl.call, br.ia, **mod-sched-brs**, **mov-from-AR-EC**	impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**	br.ia, **mov-from-AR-EFLAG**	impliedF
AR[FCR]	**mov-to-AR-FCR**	br.ia, **mov-from-AR-FCR**	impliedF
AR[FDR]	**mov-to-AR-FDR**	br.ia, **mov-from-AR-FDR**	impliedF
AR[FIR]	**mov-to-AR-FIR**	br.ia, **mov-from-AR-FIR**	impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0	br.ia, **fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**, fsetc, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1	br.ia, **fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**, **mov-from-AR-FPSR**	
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2	br.ia, **fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**, **mov-from-AR-FPSR**	
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3	br.ia, **fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**, **mov-from-AR-FPSR**	

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
AR[FPSR].sf0.flags	**fp-arith-s0**, fclrf.s0, **fcmp-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	br.ia, fchkf, **mov-from-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, fclrf.s1, **fcmp-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	br.ia, fchkf.s1, **mov-from-AR-FPSR**	
AR[FPSR].sf2.flags	**fp-arith-s2**, fclrf.s2, **fcmp-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	br.ia, fchkf.s2, **mov-from-AR-FPSR**	
AR[FPSR].sf3.flags	**fp-arith-s3**, fclrf.s3, **fcmp-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	br.ia, fchkf.s3, **mov-from-AR-FPSR**	
AR[FPSR].traps	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**	br.ia, **fp-arith**, fchkf, fcmp, fpcmp, **mov-from-AR-FPSR**	impliedF
AR[FSR]	**mov-to-AR-FSR**	br.ia, **mov-from-AR-FSR**	impliedF
AR[ITC]	**mov-to-AR-ITC**	br.ia, **mov-from-AR-ITC**	impliedF
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]	br.ia, **mov-from-AR-K**[1]	impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**	br.ia, **mod-sched-brs-counted**, **mov-from-AR-LC**	impliedF
AR[PFS]	br.call, brl.call	alloc, br.ia, br.ret, epc, **mov-from-AR-PFS**	impliedF
	mov-to-AR-PFS	alloc, br.ia, epc, **mov-from-AR-PFS**	impliedF
		br.ret	none
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RNAT**	impliedF
AR[RSC]	**mov-to-AR-RSC**	alloc, br.ia, flushrs, loadrs, **mov-from-AR-RSC**, **mov-from-AR-BSPSTORE**, **mov-to-AR-RNAT**, **mov-from-AR-RNAT**, **mov-to-AR-BSPSTORE**	impliedF
AR[SSD]	**mov-to-AR-SSD**	br.ia, **mov-from-AR-SSD**	impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill	br.ia, ld8.fill, **mov-from-AR-UNAT**	impliedF
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 45-47, 67-111	**none**	br.ia, **mov-from-AR-rv**[1]	none
AR%, % in 48-63, 112-127	**mov-to-AR-ig**[1]	br.ia, **mov-from-AR-ig**[1]	impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**indirect-brs**[1], **indirect-brp**[1], **mov-from-BR**[1]	impliedF
	mov-to-BR[1]	**indirect-brs**[1]	none
		indirect-brp[1], **mov-from-BR**[1]	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CFM	**mod-sched-brs**	**mod-sched-brs**	impliedF
		cover, alloc, rfi, loadrs, br.ret, br.call, brl.call	impliedF
		cfm-readers[2]	impliedF
	br.call, brl.call, br.ret, clrrrb, cover, rfi	**cfm-readers**	impliedF
	alloc	**cfm-readers**	none
CPUID#	**none**	**mov-from-IND-CPUID**[3]	specific
CR[CMCV]	**mov-to-CR-CMCV**	**mov-from-CR-CMCV**	data
CR[DCR]	**mov-to-CR-DCR**	**mov-from-CR-DCR**, **mem-readers-spec**	data
CR[EOI]	**mov-to-CR-EOI**	**none**	SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 118
CR[GPTA]	**mov-to-CR-GPTA**	**mov-from-CR-GPTA**, thash	data
CR[IFA]	**mov-to-CR-IFA**	itc.i, itc.d, itr.i, itr.d	implied
		mov-from-CR-IFA	data
CR[IFS]	**mov-to-CR-IFS**	**mov-from-CR-IFS**	data
		rfi	implied
	cover	rfi, **mov-from-CR-IFS**	implied
CR[IHA]	**mov-to-CR-IHA**	**mov-from-CR-IHA**	data
CR[IIM]	**mov-to-CR-IIM**	**mov-from-CR-IIM**	data
CR[IIP]	**mov-to-CR-IIP**	**mov-from-CR-IIP**	data
		rfi	implied
CR[IIPA]	**mov-to-CR-IIPA**	**mov-from-CR-IIPA**	data
CR[IPSR]	**mov-to-CR-IPSR**	**mov-from-CR-IPSR**	data
		rfi	implied
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**	**mov-from-CR-IRR**[1]	data
CR[ISR]	**mov-to-CR-ISR**	**mov-from-CR-ISR**	data
CR[ITIR]	**mov-to-CR-ITIR**	**mov-from-CR-ITIR**	data
		itc.i, itc.d, itr.i, itr.d	implied
CR[ITM]	**mov-to-CR-ITM**	**mov-from-CR-ITM**	data
CR[ITV]	**mov-to-CR-ITV**	**mov-from-CR-ITV**	data
CR[IVA]	**mov-to-CR-IVA**	**mov-from-CR-IVA**	instr
CR[IVR]	**none**	**mov-from-CR-IVR**	SC Section 5.8.3.2, “External Interrupt Vector Register (IVR – CR65)” on page 117

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
CR[LID]	**mov-to-CR-LID**	**mov-from-CR-LID**	SC Section 5.8.3.1, “Local ID (LID – CR64)” on page 116
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]	**mov-from-CR-LRR**[1]	data
CR[PMV]	**mov-to-CR-PMV**	**mov-from-CR-PMV**	data
CR[PTA]	**mov-to-CR-PTA**	**mov-from-CR-PTA**, **mem-readers**, **mem-writers**, **non-access**, thash	data
CR[TPR]	**mov-to-CR-TPR**	**mov-from-CR-TPR**, **mov-from-CR-IVR**	data
		mov-to-PSR-I[8], ssm[8]	SC Section 5.8.3.3, “Task Priority Register (TPR – CR66)” on page 117
		rfi	implied
CR%, % in 3-7, 10-15, 18, 26-63, 75-79, 82-127	**none**	**mov-from-CR-rv**[1]	none
DBR#	**mov-to-IND-DBR**[3]	**mov-from-IND-DBR**[3]	impliedF
		probe-all, **lfetch-all**, **mem-readers**, **mem-writers**	data
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	**mem-readers**, **mem-writers**, non-access	data
	itc.i, itc.d, itr.i, itr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	impliedF
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	none
		itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF
DTR	itr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d, itr.d	impliedF
	ptr.d	**mem-readers**, **mem-writers**, **non-access**	data
		ptc.g, ptc.ga, ptc.l, ptr.d	none
		itr.d, itc.d	impliedF
FR%, % in 0 - 1	**none**	**fr-readers**[1]	none
FR%, % in 2 - 127	**fr-writers**[1]**ldf-c**[1]**ldfp-c**[1]	**fr-readers**[1]	impliedF
	ldf-c[1], **ldfp-c**[1]	**fr-readers**[1]	none
GR0	**none**	**gr-readers**[1]	none
GR%, % in 1 - 127	**ld-c**[1,14]	**gr-readers**[1]	none
	gr-writers[1]**ld-c**[1,14]	**gr-readers**[1]	impliedF
IBR#	**mov-to-IND-IBR**[3]	**mov-from-IND-IBR**[3]	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
InService*	**mov-to-CR-EOI**	**mov-from-CR-IVR**	data
	mov-from-CR-IVR	**mov-from-CR-IVR**	impliedF
	mov-to-CR-EOI	**mov-to-CR-EOI**	impliedF
IP	**all**	**all**	none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d	epc, vmsw	instr
		itc.i, itc.d, itr.i, itr.d	impliedF
		ptr.i, ptr.d, ptc.e, ptc.g, ptc.ga, ptc.l	none
	itc.i, itc.d, itr.i, itr.d	epc, vmsw	instr
		itc.d, itc.i, itr.d, itr.i, ptr.d, ptr.i, ptc.g, ptc.ga, ptc.l	impliedF
ITC_LIMIT*	ptc.g, ptc.ga	ptc.g, ptc.ga	impliedF
ITR	itr.i	itr.i, itc.i, ptc.g, ptc.ga, ptc.l, ptr.i	impliedF
		epc, vmsw	instr
	ptr.i	itc.i, itr.i	impliedF
		ptc.g, ptc.ga, ptc.l, ptr.i	none
		epc, vmsw	instr
memory	**mem-writers**	**mem-readers**	none
PKR#	**mov-to-IND-PKR**[3]	**mem-readers**, **mem-writers**, **mov-from-IND-PKR**[4], **probe-all**	data
		mov-to-IND-PKR[4]	none
		mov-from-IND-PKR[3]	impliedF
		mov-to-IND-PKR[3]	impliedF
PMC#	**mov-to-IND-PMC**[3]	**mov-from-IND-PMC**[3]	impliedF
		mov-from-IND-PMD[3]	SC[3] Section 7.1.1, “Data and Instruction Breakpoint Registers” on page 144
PMD#	**mov-to-IND-PMD**[3]	**mov-from-IND-PMD**[3]	impliedF
PR0	**pr-writers**[1]	**pr-readers-br**[1], **pr-readers-nobr-nomovpr**[1], **mov-from-PR**[13], **mov-to-PR**[13]	none
PR%, % in 1 - 15	**pr-writers**[1], **mov-to-PR-allreg**[7]	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[13]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7]	**pr-readers-br**[1]	none
PR%, % in 16 - 62	**pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[13]	impliedF
	pr-writers-fp[1]	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PR63	**mod-sched-brs**, **pr-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[13]	impliedF
	pr-writers-fp[1], **mod-sched-brs**	**pr-readers-br**[1]	impliedF
	pr-writers-int[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-readers-br**[1]	none
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**	**mem-readers**, **mem-writers**	implied
	sys-mask-writers-partial[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**	data
	user-mask-writers-partial[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**, **mov-from-PSR-um**	impliedF
	rfi	**mem-readers**, **mem-writers**, **mov-from-PSR**, **mov-from-PSR-um**	impliedF
PSR.bn	bsw, rfi	**gr-readers**[11], **gr-writers**[11]	impliedF
PSR.cpl	epc, br.ret	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	implied
	rfi	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-K**, **mov-from-IND-PMD**, **probe-all**, **mem-readers**, **mem-writers**, **lfetch-all**	impliedF
PSR.da	rfi	**mem-readers**, **lfetch-all**, **mem-writers**, **probe-fault**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.db	**mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-fault**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-fault**	impliedF
PSR.dd	rfi	**lfetch-all**, **mem-readers**, **probe-fault**, **mem-writers**	impliedF
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-readers**[9], **fr-writers**[9]	data
		mov-from-PSR	impliedF
	rfi	**fr-readers**[9], **fr-writers**[9], **mov-from-PSR**	impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-writers**[9], **fr-readers**[9]	data
		mov-from-PSR	impliedF
	rfi	**fr-writers**[9], **fr-readers**[9], **mov-from-PSR**	impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	br.ia	data
		mov-from-PSR	impliedF
	rfi	br.ia, **mov-from-PSR**	impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mem-readers**, **mem-writers**, **non-access**	data
		mov-from-PSR	impliedF
	rfi	**mem-readers**, **mem-writers**, **non-access**, **mov-from-PSR**	impliedF
PSR.ed	rfi	**lfetch-all**, **mem-readers-spec**	impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ia	rfi	**all**	none
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	data
	rfi	**mov-from-PSR**, cover, itc.i, itc.d, itr.i, itr.d, **mov-from-interruption-CR**, **mov-to-interruption-CR**	impliedF
PSR.id	rfi	**all**	none
PSR.is	br.ia, rfi	**none**	none
PSR.it	rfi	**branches**, **mov-from-PSR**, chk, epc, fchkf, vmsw	impliedF
PSR.lp	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		br.ret	data
	rfi	**mov-from-PSR**, br.ret	impliedF
PSR.mc	rfi	**mov-from-PSR**	impliedF

Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PSR.mfh	**fr-writers**[10], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.mfl	**fr-writers**[10], **user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**lfetch-all**, **mem-readers**, **mem-writers**, **probe-all**	data
		mov-from-PSR	impliedF
	rfi	**lfetch-all**, **mem-readers**, **mem-writers**, **mov-from-PSR**, **probe-all**	impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR**	impliedF
PSR.ri	rfi	**all**	none
PSR.rt	**mov-to-PSR-l**	**mov-from-PSR**	impliedF
		alloc, flushrs, loadrs	data
	rfi	**mov-from-PSR**, alloc, flushrs, loadrs	impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-AR-ITC	data
	rfi	**mov-from-AR-ITC**, **mov-from-PSR**	impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**mov-from-PSR**	impliedF
		mov-from-IND-PMD, **mov-to-PSR-um**, rum, sum	data
	rfi	**mov-from-IND-PMD**, **mov-from-PSR**, **mov-to-PSR-um**, rum, sum	impliedF
PSR.ss	rfi	**all**	impliedF
PSR.tb	**mov-to-PSR-l**	**branches**, chk, fchkf	data
		mov-from-PSR	impliedF
	rfi	**branches**, chk, fchkf, **mov-from-PSR**	impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**mov-from-PSR-um**, **mov-from-PSR**	impliedF
RR#	**mov-to-IND-RR**[6]	**mem-readers**, **mem-writers**, itc.i, itc.d, itr.i, itr.d, **non-access**, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, thash, ttag	data
		mov-from-IND-RR[6]	impliedF
RSE	**rse-writers**[15]	**rse-readers**[15]	impliedF

14. ISR.ir typo

1. On page 2:148, Volume 2, Part I, Table 8-2 in footnote C, change from:

   “ISR.ri”

   to:

   “ISR.ir”