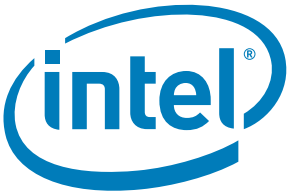
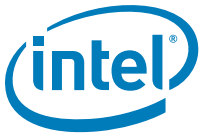


Intel® Itanium® Architecture Software Developer's Manual Rev 2.3

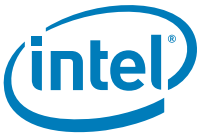
Specification Update

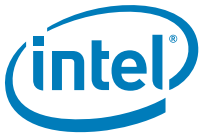
June 2012

Document Number: 248699-014



Contents

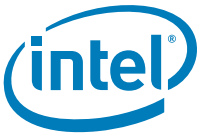


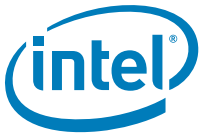


Revision History

Document Number	Version Number	Description	Date
248699	014	This document applies to Version 2.3 of the *Intel® Itanium® Architecture Software Developer's Manual*, published in May 2010.	June 2012

§





1 Preface

This document is an update to the *Intel® Itanium® Architecture Software Developer's Manual*, *Revision 2.3* which is comprised of the following volumes:

Title / Volume	Document #
Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3, Volume 1: Application Architecture	245317
Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3, Volume 2: System Architecture	245318
Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3, Volume 3: Intel® Itanium® Instruction Set Reference	323207
Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3, Volume 4: IA-32 Instruction Set Reference	323208

This document is a compilation of specification changes, clarifications, and corrections that collectively comprise an update to the *Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3* (also called SDM 2.3 in rest of this document). Section 2, Summary of Changes, shows the list of specification changes, clarifications, and corrections. Each entry represents a specific theme where SDM 2.3 is being updated in this document, each entry has a brief title indicating the relevant topic(s) involved in the update, and the update text itself can be found in corresponding "Replacement Sections" which are included in this document and are meant to replace the relevant sections of SDM 2.3 in its current form. Please take note that any one entry in the list of changes can have change text in several Replacement Sections, and any one Replacement Section can contain the changed text of several entries.

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.

1.1 Nomenclature

Specification Changes are modifications to the current published specifications for Intel® Itanium® 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*.

2 Summary Table of Changes

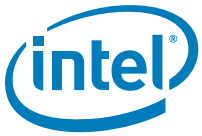
The following tables indicate the specification changes and specification clarifications that apply to the *Intel® Itanium® Architecture Software Developer's Manual, Rev 2.3*.

2.1 Specification Changes

No.	SPECIFICATION CHANGES	Replacement Sections in this Specification Update
A-926	Framework for configurable data access hints	- Vol.1: sections 4.4.6.1, 4.4.6.2, 4.4.6.3 and 4.7 - Vol.3: section 2.2, pages for *branch, branch long, mov-to-DAHR, lfetch, ld, st, hint* - Vol.3: chapters 3, 4 and 5
A-946	lfetch.count	- Vol 1: section 4.4.6.1 - Vol 3: sections 2.2 and 4.1
A-1001	AR.ruc intervals add "approximately" to CR.itc intervals	- Vol 1: section 3.1.8.11
A-1002	Promote PMC.vmm and PMC.ch to architectural state	Vol 2: sections 7.2.1, 7.2.3 and 11.10.3 (PAL_PROC_GET_PROCEDURE)
A-1003	New implementation-specific policies range to PAL_GET/ SET_HW_POLICY	Vol 2: section 11.10.3 (PAL_SET/GET_HW_POLICY)
A-1004	New DAHR fields and hints	Vol 1: section 4.4.6.3
A-1005	Updates to PAL_PLATFORM_ADDR	Vol 2: section 11.10.3 (PAL_PLATFORM_ADDR)
A-1006	Updates to TLB insert serialization requirements	- Vol 2: Part 1, Section 4.1.1.2 (Translation Cache); Part 2, Sections 5.2.1.1 and 5.2.2.1 - Vol 3: section 2.2, pages for *itc* and *itr*
A-1007	Correction to faulting on mov-from-DAHR	Vol 3: section 2.2, page for *mov* - Mov Indirect Register
A-1008	Purge behaviors of VHPT	Vol 2: Part 1, Table 4-2 (Purge behavior of VHPT Inserts)

2.2 Specification Clarifications

No.	SPECIFICATION CLARIFICATIONS	Replacement Sections in this Specification Update
D-941	Lfetch.count Clarifications	Vol 3: section 2.2, page for *lfetch*
D-1001	Clarify CFM/CR.ifs state on SAL return from PALE_PMI	Vol 2: section 11.5.2
D-1002	Clarify DAHR dependency semantics	vol 3: section 5.3.2 and 5.3.3
D-1004	Clarify PAL procedure calling conventions	Vol 2: sections 11.10, 11.10.2.1.1, 11.10.2.2.6, 11.10.2.4, Table 11-48



2.2 Specification Clarifications

No.	SPECIFICATION CLARIFICATIONS	Replacement Sections in this Specification Update
D-1005	Clarifications of memory parameters for PAL procedures	Vol 2: Sec 11.10.3, pages for PAL_BRAND_INFO, PAL_HALT_INFO, PAL_MC_DYNAMIC_STATE, PAL_MC_ERROR_INJECT, PAL_PERF_MON_INFO, PAL_PSTATE_INFO, PAL_TEST_PROC PAL_VM_TR_READ, PAL_VP_INFO, PAL_VP_INIT_ENV, PAL_VP_REGISTER, PAL_VP_RESTORE, PAL_VP_SAVE, PAL_VP_TERMINATE
D-1006	Delete reference to IA-32 Code Performance Accounting	Vol 2: Section 7.2
D-1007	Clarifications to PAL_MC_DYNAMIC_STATE buffer alignment	Vol 2: Section 11.17

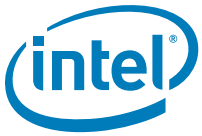
2.3 Documentation Changes

No.	DOCUMENTATION	Replacement Sections in this Specification Update
D-1003	Typographical errors in various sections	- Vol 2: Sections 4.1.1.5, 5.8.3.9, 11.6.1.3, 11.6.1.5, Table 11-16, sections 11.7.4.1.3, 11.7.4.3.5, 13.3.3, Fig.13-6 - Vol. 3: Table 3-1, Sections 4.1, 5.3.5 - Vol.4: pages for CALL, CPUID, FPTAN, FSIN, FXCH, FYL2X, FYL2XP1

3 Table of Contents for Replacement Sections

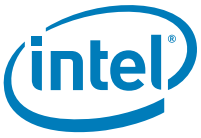
The following is the list of updated sections arranged in sequential order (per volume) of the *Intel® Itanium® Architecture Software Developer's Manual*, *Revision 2.3* based on this specification update:

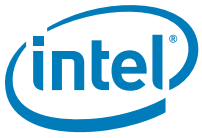
	Replacement Sections	Appendix No.	Page No.
Vol 1	section 3.1.8.11	V1-A	13
	sections 4.4.6.1, 4.4.6.2, 4.4.6.3	V1-B	15
	section 4.7	V1-C	25
Vol 2	Part 1, Section 4.1.1.2	V2-A	27
	Part 2, Sections 5.2.1.1 and 5.2.2.1	V2-B	29
	Part 1 Table 4-2	V2-C	31
	Part 1, Section 4.1.1.5	V2-D	33
	Part 1 Section 5.8.3.9	V2-E	37
	Part 1 Section 7.2	V2-F	39
	Part 1 Section 7.2.1 and 7.2.3	V2-G	41
	Part 1 Table 11-12 and Section 11.5.2	V2-H	45
	Part 1 Section 11.6.1.3	V2-I	47
	Part 1 Section 11.6.1.5	V2-J	51
	Part 1 Table 11-16	V2-K	53
	Part 1 Section 11.7.4.1.3 and 11.7.4.3.5	V2-L	59
	Part 1 Section 11.10	V2-M	61
	Part 1 Section 11.10.3 PAL_BRAND_INFO	V2-N	65
	Part 1 Section 11.10.3 PAL_HALT_INFO	V2-O	67
	Part 1 Section 11.10.3 PAL_MC_DYNAMIC_STATE	V2-P	69
	Part 1 Section 11.10.3 PAL_MC_ERROR_INFO/ PAL_MC_ERROR_INJECT	V2-Q	71
	Part 1 Section 11.10.3 PAL_PERF_MON_INFO	V2-R	93
	Part 1 Section 11.10.3 PAL_VP_INFO	V2-S	95
	Part 1 Section 11.10.3 PAL_VP_REGISTER – PAL	V2-T	97
	Part 1 Section 11.10.3 PAL_PLATFORM_ADDR	V2-U	99
	Part 1 Section 11.10.3 PAL_PSTATE_INFO	V2-V	101
	Part 1 Section 11.10.3 PAL_SET_GET	V2-W	103
	Part 1 Section 11.10.3 PAL_TEST_PROC.	V2-X	107
	Part 1 Section 11.10.3 PAL_VM_TR_READ	V2-Y	111
	Part 1 Section 11.10.3 PAL_VP_INIT_ENV	V2-Z	113
	Part 1 Section 11.10.3 PAL_VP_RESTORE	V2-AA	117
	Part 1 Section 11.10.3 PAL_VP_SAVE/ PAL_VP_TERMINATE	V2-BB	119
	Part 1 Section 11.10.3 PROC_GET/SAVE_FEATURES	V2-CC	121
	Part 1 Fig 13-6	V2-DD	127
	Part 1 Section 13.3.3	V2-EE	129



Vol 3	Section 2.2 - branch	V3-A	131
	Section 2.2 - branch_long	V3-B	141
	Section 2.2 - DAHR	V3-C	145
	Section 2.2 - hint	V3-D	147
	Section 2.2 - Insert Translation Cache	V3-E	149
	Section 2.2 - itr — Insert Translation Register	V3-F	151
	Section 2.2 - ld	V3-G	153
	Section 2.2 - lfetch	V3-H	159
	Section 2.2 - Move Indirect Register	V3-I	163
	Section 2.2 - st — Store	V3-J	167
	Chapter 3	V3-K	157
	Chapter 4	V3-L	169
	Chapter 5	V3-M	253
Vol 4	Section 2.3 - CPUID	V4-A	289

Each Replacement Section is provided as separate sections in this document and is meant to replace the corresponding portion(s) of the *Intel® Itanium® Architecture Software Developer's Manual, Revision 2.3.*





V1-A Section 3.1.8.11

3.1.8.11 Resource Utilization Counter (RUC – AR 45)

The Resource Utilization Counter (RUC) is a 64-bit register which counts up at a fixed relationship to the input clock to the processor, when the processor is active. RUC provides an estimate of the portion of resources used by a logical processor with respect to all resources provided by the underlying physical processor.

The Resource Utilization Counter (RUC) is a 64-bit register which provides an estimate of the portion of resources used by a logical processor with respect to all resources provided by the underlying physical processor.

In a given time interval, the difference in the RUC values for all of the logical processors on a given physical processor add up to approximately the difference seen in the ITC on that physical processor for that same interval.

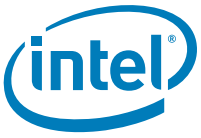
A sequence of reads of the RUC is guaranteed to return ever-increasing values (except for the case of the counter wrapping back to 0) corresponding to the program order of the reads.

System software can secure the resource utilization counter from non-privileged access. When secured, a read of the RUC at any privilege level other than the most privileged causes a Privileged Register fault.

The RUC for a logical processor does not count when that logical processor is in LIGHT_HALT, unless all logical processors on a given physical processor are in LIGHT_HALT, in which case the last logical on a given physical processor to enter LIGHT_HALT has its RUC continue to count.

With processor virtualization, the RUC can be used to communicate the portion of resources used by a virtual processor. See Section 3.4, “Processor Virtualization” on page 2:44 and Section 11.7, “PAL Virtualization Support” on page 2:324 for details on virtual processors.

The RUC register is not supported on all processor implementations. Software can check CPUID register 4 to determine the availability of this feature. The RUC register is reserved when this feature is not supported.

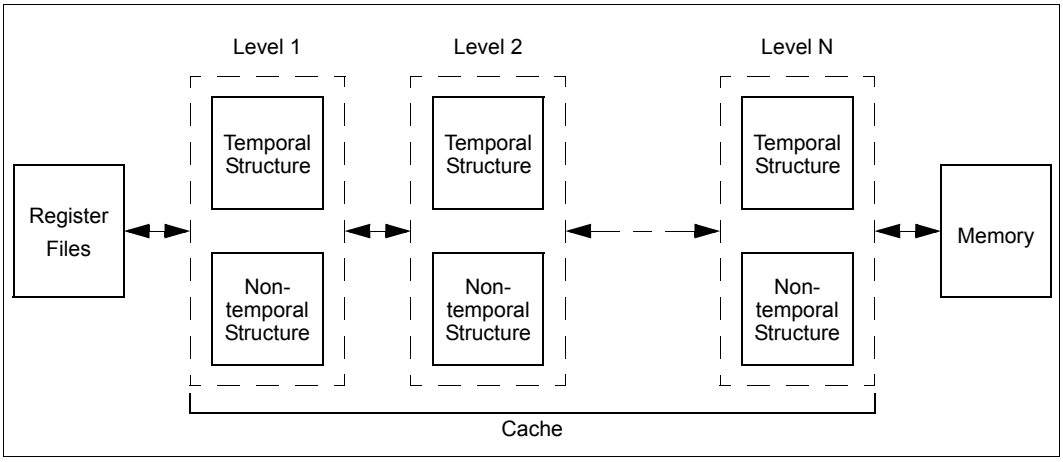


V1-B Sections 4.4.6.1, 4.4.6.2, 4.4.6.3

4.4.6.1 Hierarchy Control and Hints

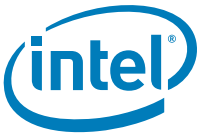
Memory access instructions are defined to specify whether the data being accessed possesses temporal locality. In addition, memory access instructions can specify which levels of the memory hierarchy are affected by the access. This leads to an architectural view of the memory hierarchy depicted in Figure 4-1 composed of zero or more levels of cache between the register files and memory where each level may consist of two parallel structures: a temporal structure and a non-temporal structure. Note that this view applies to data accesses and not instruction accesses.

Figure 4-1. Memory Hierarchy



The temporal structures cache memory accessed with temporal locality; the non-temporal structures cache memory accessed without temporal locality. Both structures assume that memory accesses possess spatial locality. The existence of separate temporal and non-temporal structures, as well as the number of levels of cache, is implementation dependent. Please see the processor-specific documentation for further information.

Three mechanisms are defined for allocation control: locality hints; explicit prefetch; and implicit prefetch. Locality hints are specified by load, store, and explicit prefetch (`lfetch`) instructions. A locality hint specifies a hierarchy level (e.g., 1, 2, all). An access that is temporal with respect to a given hierarchy level is treated as temporal with respect to all lower (higher numbered) levels. An access that is non-temporal with respect to a given hierarchy level is treated as temporal with respect to all lower levels. Finding a cache line closer in the hierarchy than specified in the hint does not demote the line. This enables the precise management of lines using `lfetch` and then subsequent uses by `.nta` loads and stores to retain that level in the hierarchy. For



example, specifying the `.nt2` hint by a prefetch indicates that the data should be cached at level 3. Subsequent loads and stores can specify `.nta` and have the data remain at level 3.

Locality hints do not affect the functional behavior of the program and may be ignored by the implementation. The locality hints available to loads, stores, and explicit prefetch instructions are given in Table 4-18. Instruction accesses are considered to possess both temporal and spatial locality with respect to level 1.

Table 4-18.Locality Hints Specified by Each Instruction Class

Locality Hint	Mnemonic Supported by Instruction Type			
	Load[a]	Store[a]	Semaphore	lfetch, lfetch.fault
Temporal, level 1	*none* or d0	*none* or d0	*none* or d0	*none* or d0
Non-temporal, level 1	nt1 or d1	d1	nt1 or d1	nt1 or d1
Non-temporal, level 2	d2	d2	d2	nt2 or d2
Non-temporal, all levels	nta or d3	nta or d3	nta or d3	nta or d3
Hint d4	d4	d4		d4
Hint d5	d5	d5		d5
Hint d6	d6	d6		d6
Hint d7	d7	d7		d7

a. Hints d4, d5, d6 and d7 cannot be specified in the form of the instruction with address post increment.

Each locality hint implies a particular allocation path in the memory hierarchy. The allocation paths corresponding to the locality hints are depicted in Figure 4-2. The allocation path specifies the structures in which the line containing the data being referenced would best be allocated. If the line is already at the same or higher level in the hierarchy no movement occurs. Hinting that a datum should be cached in a temporal structure indicates that it is likely to be read in the near future. Hinting that a datum should not be cached in a temporal structure indicates that it is not likely to be read in the near future. For stores, the `.nta` completer also hints that the store may be part of a set of streaming stores that would likely overwrite the entire cache line without any data in that line first being read, enabling the processor to avoid fetching the data.\

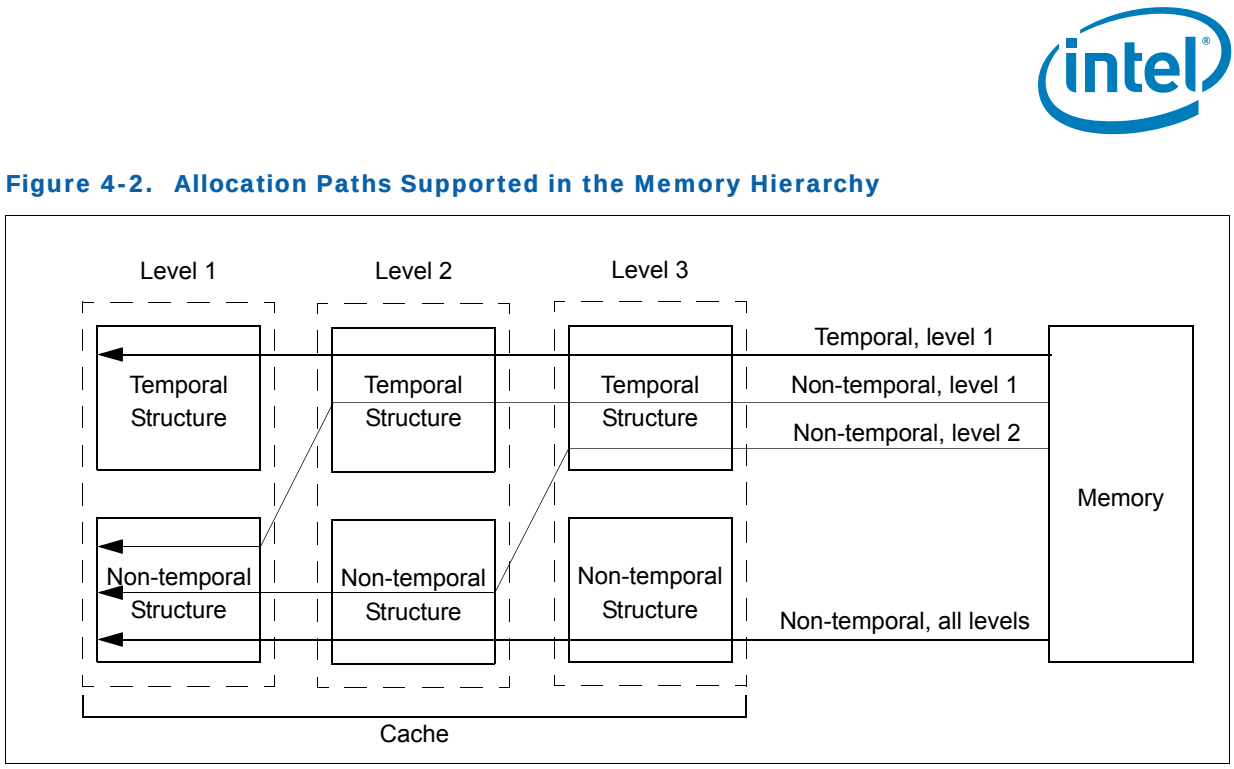


Figure 4-2. Allocation Paths Supported in the Memory Hierarchy

Explicit prefetch is defined in the form of the line prefetch instruction (`lfetch`, `lfetch.fault`). The lfetch instructions moves the line containing the addressed byte to a location in the memory hierarchy specified by the locality hint. If the line is already at the same or higher level in the hierarchy, no movement occurs. Both immediate and register post-increment are defined for `lfetch` and `lfetch.fault`. The `lfetch` instruction does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation. The `lfetch.fault` instruction affects the memory hierarchy in exactly the same way as `lfetch` but takes exceptions as if it were a 1-byte load instruction.

Implicit prefetch is based on the address post-increment of loads, stores, `lfetch` and `lfetch.fault`. The line containing the post-incremented address is moved in the memory hierarchy based on the locality hint of the originating load, store, `lfetch` or `lfetch.fault`. If the line is already at the same or higher level in the hierarchy then no movement occurs. Implicit prefetch does not cause any exceptions, does not affect program behavior, and may be ignored by the implementation.

Another form of hint that can be provided on loads is the `ld.bias` load type. This is a hint to the implementation to acquire exclusive ownership of the line containing the addressed data. The bias hint does not affect program functionality and may be ignored by the implementation.

The following instructions are defined for flush control: flush cache (`fc`, `fc.i`) and flush write buffers (`fwb`). The `fc` instruction invalidates the cache line in all levels of the memory hierarchy above memory. If the cache line is not consistent with memory, then it is copied into memory before invalidation. The `fc.i` instruction ensures the data cache line associated with an address is coherent with the instruction caches. The `fc.i` instruction is not required to invalidate the targeted cache line nor write the targeted cache line back to memory if it is inconsistent with memory, but may do so if this is required to make the targeted cache line coherent with the instruction caches. The `fwb` instruction provides a hint to flush all pending buffered writes to memory (no indication of completion occurs).

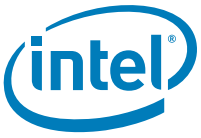


Table 4-19 summarizes the memory hierarchy control instructions and hint mechanisms.

Table 4-19.Memory Hierarchy Control Instructions and Hint Mechanisms

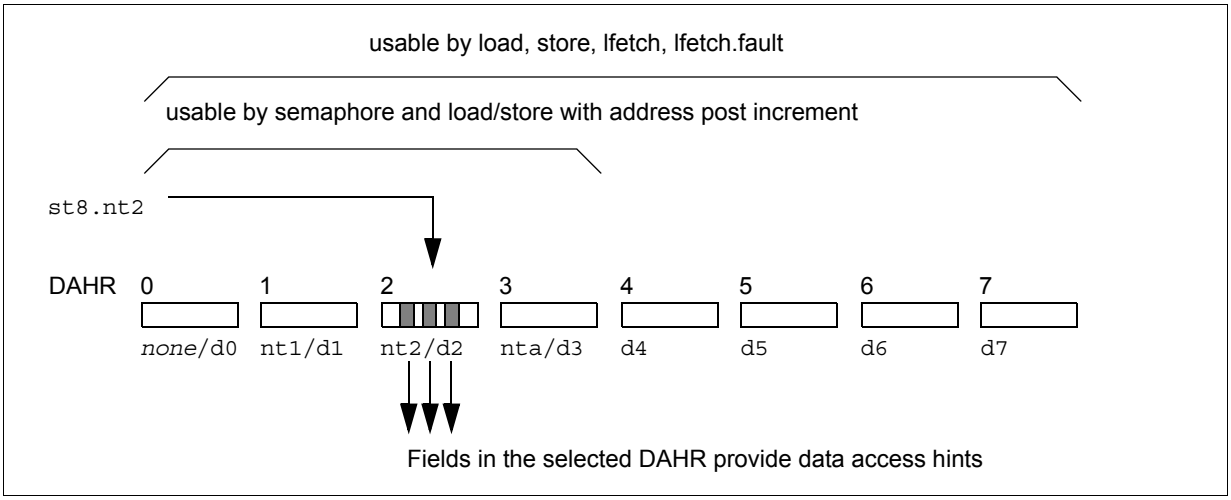
Mnemonic	Operation
`.nt1`, `.nta`, `.d0`, `.d1`, `.d2`, `.d3`, `.d4`, `.d5`, `.d6`, `.d7` completer on loads	Load usage hints[a]
`.nta`, `.d0`, `.d1`, `.d2`, `.d3`, `.d4`, `.d5`, `.d6`, `.d7` completer on stores	Store usage hints[a]
Prefetch line at post-increment address on loads and stores	Prefetch hint
lfetch, lfetch.fault with .nt1, .nt2, .nta, .d0, .d1, .d2, .d3, .d4, .d5, .d6, .d7 hints	Prefetch line
fc, fc.i	Flush cache
fwb	Flush write buffers

a. Hints d4, d5, d6 and d7 cannot be specified in the form of the instruction with address post increment.

4.4.6.2 Programmable Data Access Hints

Some processor implementations allow software to refine precisely how the processor should respond to the locality hints specified by load, store, semaphore and explicit prefetch (`lfetch`) instructions. This is done through a set of Data Access Hint Registers (DAHRs). In processors that implement them, the locality hint specified in the instruction selects one of the DAHRs, which then provides the hint information for the memory access. There are eight DAHRs usable by load, store and lfetch instructions (DAHR[0-7]); semaphore instructions, and load and store instructions with address post increment can use only the first four of these (DAHR[0-3]). See Figure 4-3.

Figure 4-3. Data Access Hint Registers



Each DAHR contains fields which provide the processor with various types of data access hints. (See Section 4.4.6.3 for information about specific data access hint fields.) When a DAHR has not been explicitly programmed by software, these data access hint fields are automatically set to **default** values that best implement the generic locality hints as shown in Table 4-20.

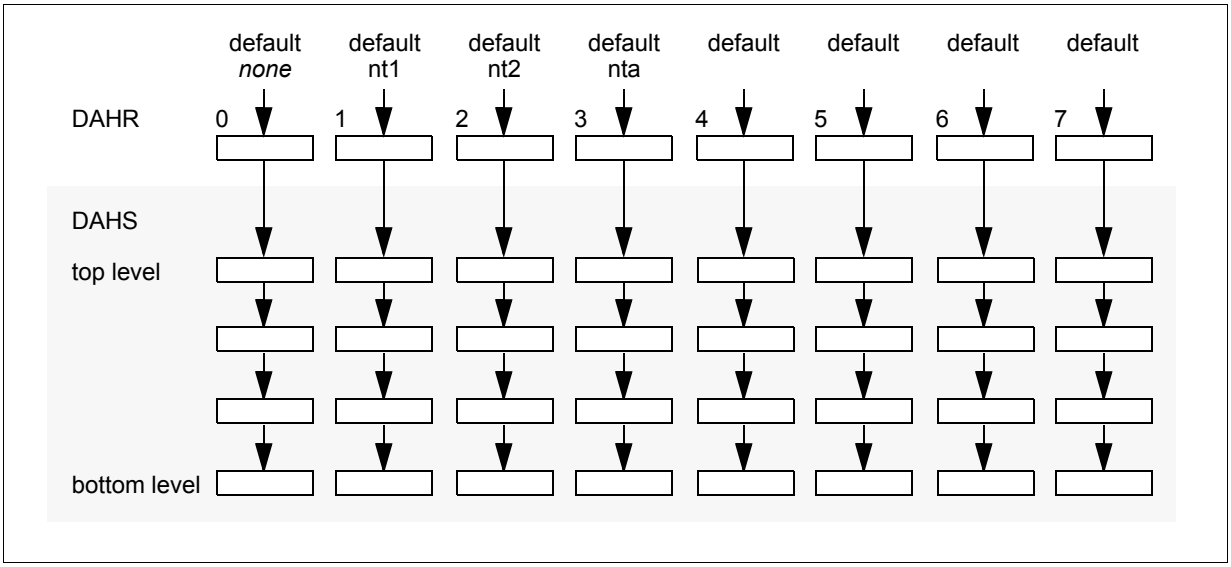
Table 4-20.Default values for DAHRs

DAHR	Default data access hint settings correspond to
0	Temporal, level 1
1	Non-temporal, level 1
2	Non-temporal, level 2
3	Non-temporal, all levels
4	DAHR[4] default
5	DAHR[5] default
6	DAHR[6] default
7	DAHR[7] default

DAHRs are not saved and restored as part of process context in Operating Systems, but are ephemeral state like the ALAT. When DAHR state is lost due to a context switch, the DAHRs revert to the default values. DAHRs also revert to default values upon execution of a br.call instruction.

Processors that implement DAHRs may also optionally automatically save and restore the DAHRs on `br.call` and `br.ret` in a structure called the Data Access Hint Stack (DAHS) within the processor. Each stack level consists of eight elements corresponding to the eight DAHRs. The number of stack levels is implementation-dependent. On a `br.call` (and on interruptions with PSR.ic==1), the elements in the stack are pushed down one level (the elements in the bottom stack level are lost), the values in the DAHRs are copied into the elements in the top stack level, and then the DAHRs revert to default values. See Figure 4-4. On a `br.ret` (and on return from interruption with IPSR.ic==1), the elements in the top stack level are copied into the DAHRs, and the elements in the stack are popped up one level, with the elements in the bottom stack level reverting to default values. See Figure 4-5. On a mov-to-BSPSTORE instruction (used in context switch, but rarely otherwise), all DAHRs and all elements at all levels of the DAHS revert to default values.

Figure 4-4. Data Access Hint Stack Operation on br.call



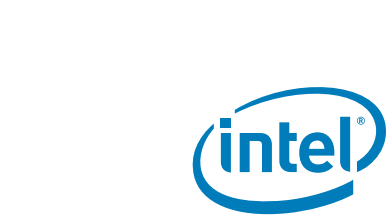
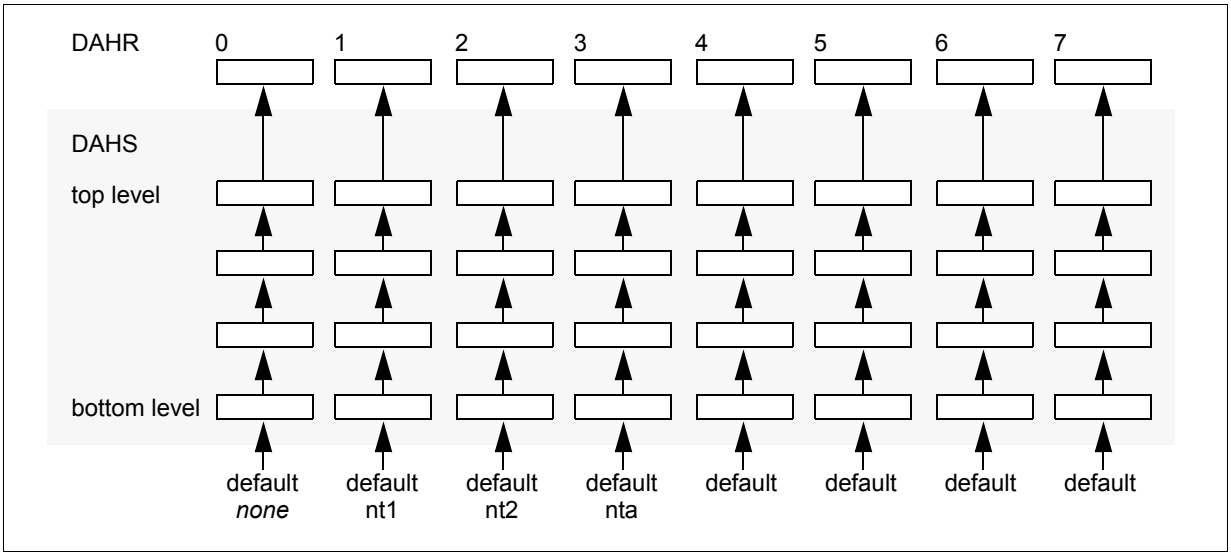


Figure 4-5. Data Access Hint Stack Operation on br.ret



A DAHR can be programmed with a mov-to-DAHR instruction, which copies a set of data access hint fields, encoded within an immediate field in the instruction, into the DAHR. This instruction executes as a nop on processors that do not implement DAHRs, and hence can be used in generic code.

The value in a DAHR can be copied to a GR with a mov-from-DAHR instruction. This instruction takes an Illegal Operation fault on processors that do not implement DAHRs.

4.4.6.3 Data Access Hint Fields

This section describes specific data access hints within DAHRs.

Figure 4-6.Data Access Hint Register Format

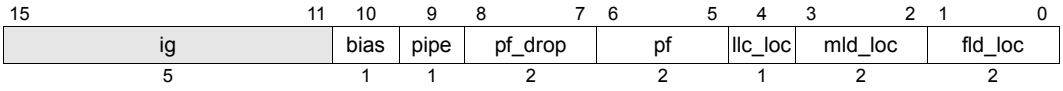


Table 4-21.Data Access Hint Fields

Field	Bits	Description
fld_loc	1:0	First-level (L1) data cache locality
mld_loc	3:2	Mid-level (L2) data cache locality
llc_loc	4	Last-level (L3) data cache locality
pf	6:5	Data prefetch
pf_drop	8:7	Data prefetch drop
pipe	9	Block pipeline vs. background handling for lfetch and speculative loads
bias	10	Bias cache allocation to shared or exclusive
ig	15:11	Writes are ignored; reads return 0

The semantics of the hints for these hint fields are described in the following tables.

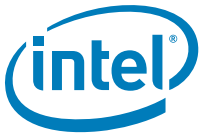


Table 4-22.fld_loc Hint Field

Bit pattern	Name	Description
00	fld_normal	normal cache allocation and fill
01	fld_nru	mark cache line as not recently used (most eligible for replacement), whether the access requires an L1 allocation and fill or the access hits in the L1 cache
10	fld_no_allocate	if the access does not hit in the L1 cache, do not allocate nor fill into the L1 cache
11	unused	

The hints specified by the *fld_loc* field allow software to specify the locality, or likelihood of data reuse, with regards to the first-level (L1) cache. For example, the *fld_nru* hint can be used to indicate that the data has some non-temporal (spatial) locality (meaning that adjacent memory objects are likely to be referenced as well) but poor temporal locality (meaning that the referenced data is unlikely to be re-accessed soon). Processors may implement this hint by placing the data in a separate non-temporal structure at the first level, if implemented, or by encaching the data in the level 1 cache, but marking the line as eligible for replacement. The *fld_no_allocate* hint is stronger, indicating that the data is unlikely to have any kind of locality (or likelihood of data reuse), with regards to the level 1 cache. Processors may implement this hint by not allocating space at all for the data at level 1.

Table 4-23.mld_loc Hint Field

Bit pattern	Name	Description
00	mld_normal	normal cache allocation and fill
01	mld_nru	mark cache line as not recently used (most eligible for replacement), whether the access requires an L2 allocation and fill or the access hits in the L2 cache
10	mld_no_allocate	if the access does not hit in the L2 cache, do not allocate nor fill into the L2 cache
11	unused	

The hints specified by the *mld_loc* field allow software to specify the locality, or likelihood of data reuse, with regards to the mid-level (L2) cache, similarly to the level 1 cache hints.

Table 4-24.llc_loc Hint Field

Bit pattern	Name	Description
0	llc_normal	normal cache allocation and fill
1	llc_nru	mark cache line as not recently used (most eligible for replacement), whether the access requires an L3 allocation and fill or the access hits in the L3 cache

The hints specified by the *llc_loc* field allow software to specify the locality, or likelihood of data reuse, with regards to the last-level (L1) cache, similarly to the level 1 and 2 cache hints, except that there isn't a no-allocate hint.

Table 4-25.pf Hint Field

Bit pattern	Name	Description
00	pf_normal	normal processor-initiated prefetching enabled
01	pf_no_fld	disable processor-initiated prefetching into the first-level (L1) data cache; all other processor-initiated prefetching enabled

Table 4-25.pf Hint Field

Bit pattern	Name	Description
10	pf_no_mld	disable processor-initiated prefetching into the first-level (L1) data and mid-level (L2) caches; all other processor-initiated prefetching enabled
11	pf_none	disable all processor-initiated prefetching

The hints specified by the *pf* field allow software to control any data prefetching that may be initiated by the processor based on this reference. Such automatic data prefetching can be disabled at the first-level cache (*pf_no_fld*), the mid-level cache (*pf_no_mld*), or at all cache levels (*pf_none*).

Table 4-26.pf_drop Hint Field

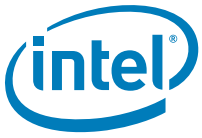
Bit pattern	Name	Description
00	pfd_normal	normal software-initiated and processor-initiated data prefetching
01	pfd_tlb	an attempted data prefetch is dropped if the address misses in the data TLB
10	pfd_tlb_mld	an attempted data prefetch is dropped if the address misses in the data TLB or the mid-level (L2) data cache
11	pfd_any	an attempted data prefetch is dropped if the address misses in the data TLB or the mid-level (L2) data cache, or if any other events occur which would require additional execution resources to handle

The hints specified by the *pf_drop* field allow software further control over any software-initiated data prefetching due to this instruction (for the lfetch instruction) or any data prefetching that may be initiated by the processor based on this reference. Rather than disabling prefetching into various levels of cache, as provided by hints in the *pf* field, hints specified by this field allow software to specify that prefetching should be done, unless the processor determines that such prefetching would require additional execution resources. For example, prefetches may be dropped if it is determined that the virtual address translation needed is not already in the Data TLB (*pfd_tlb*); if it is determined that either the translation is not present or the data is not already at least at the mid-level cache level (*pfd_tlb_mld*); or if these or any other additional execution resources are needed in order to perform the prefetch (*pfd_any*).

Table 4-27.pipe Hint Field

Bit pattern	Name	Description
0	pipe_defer	lfetch instructions that miss in the TLB need not block the pipeline, but the VHPT walker may fill their TLB translations in the background, while the pipeline continues; speculative loads may be spontaneously deferred on a TLB miss or an MLD miss
1	pipe_block	lfetch instructions block the pipeline until they are done fetching their TLB translations; speculative loads are not spontaneously deferred and block uses of their target registers until they have completed

The hints specified by the *pipe* field allow software to specify how likely or soon it is to need the data specified by an lfetch instruction or a speculative load instruction. The *pipe_defer* hint indicates that the data should be prefetched as soon as possible (lfetch instruction) or copied into the target general register (speculative load instruction) if it would not be very disruptive to the execution pipeline to do so. If this data movement might delay the pipeline execution of subsequent instructions (for example, due to TLB or mid-level cache misses), the instruction is instead executed in the background, allowing the pipeline to continue executing subsequent instructions. For speculative



load instructions, if this background execution would take significantly extra time, the processor may spontaneously defer the speculative load, as allowed by the recovery model (see SDM vol 2, section 5.5.5, “Deferral of Speculative Load Faults”).

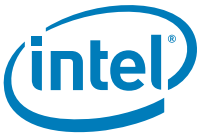
The pipe_block hint indicates that the data should be prefetched as soon as possible (lfetch instruction) or copied into the target general register (speculative load instruction) independent of whether this might delay the pipeline execution of subsequent instructions. For speculative load instructions, no spontaneous deferral is done.

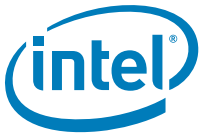
Table 4-28.bias Hint Field

Bit pattern	Name	Description
0	bias_excl	if the processor has a choice of getting a line in either the shared or exclusive MESI states, it should choose exclusive
1	bias_shared	if the processor has a choice of getting a line in either the shared or exclusive MESI states, it should choose shared

The hints specified by the *bias* field allow software to optimize cache coherence activities. For load instructions and lfetch instructions, if the referenced line is not already present in the processor’s cache, and if the processor can encache the data in either the shared or the exclusive MESI states, the bias_excl hint indicates that the processor should encache the data in the exclusive state, while the bias_shared hint indicates that the processor should encache the data in the shared state.

§





V1-C Section 4.7

4.7 Register File Transfers

Table 4-32 shows the instructions defined to move values between the general register file and the floating-point, branch, predicate, performance monitor, processor identification, and application register files. Several of the transfer instructions share the same mnemonic (`mov`). The value of the operand identifies which register file is accessed.

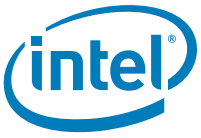
Table 4-32.Register File Transfer Instructions

Mnemonic	Operation
`getf.exp, getf.sig`	Move FR exponent or significand to GR
`getf.s, getf.d`	Move single/double precision memory format from FR to GR
`setf.s, setf.d`	Move single/double precision memory format from GR to FR
`setf.exp, setf.sig`	Move from GR to FR exponent or significand
`mov =br`	Move from BR to GR
`mov br=`	Move from GR to BR
`mov =pr`	Move from predicates to GR
`mov pr=, mov pr.rot=`	Move from GR to predicates
`mov ar=`	Move from GR to AR
`mov =ar`	Move from AR to GR
`mov =psr.um`	Move from user mask to GR
`mov psr.um=`	Move from GR to user mask
`sum, rum`	Set and reset user mask
`mov =pmd[...]`	Move from performance monitor data register to GR
`mov =cpuid[...]`	Move from processor identification register to GR
`mov =ip`	Move from Instruction Pointer
`mov =dahr[...]`	Move from DAHR to GR
`mov dahr=`	Move immediate to DAHR

Memory access instructions only target or source the general and floating-point register files. It is necessary to use the general register file as an intermediary for transfers between memory and all other register files except the floating-point register file.

Two classes of move are defined between the general registers and the floating-point registers. The first type moves the significand or the sign/exponent (`getf.sig`, `setf.sig`, `getf.exp`, `setf.exp`). The second type moves entire single or double precision numbers (`getf.s`, `setf.s`, `getf.d`, `setf.d`). These instructions also perform a conversion between the deferred exception token formats.

Instructions are provided to transfer between the branch registers and the general registers. The move to branch register instruction can also optionally include branch hints. See “Branch Prediction Hints” on page 1:78.



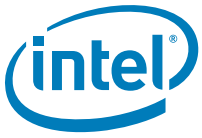
Instructions are defined to transfer between the predicate register file and a general register. These instructions operate in a “broadside” manner whereby multiple predicate registers are transferred in parallel (predicate register N is transferred to and from bit N of a general register). The move to predicate instruction (`mov pr=`) transfers a general register to multiple predicate registers according to a mask specified by an immediate. The mask contains one bit for each of the static predicate registers (PR 1 through PR 15 – PR 0 is hardwired to 1) and one bit for all of the rotating predicates (PR 16 through PR63). A predicate register is written from the corresponding bit in a general register if the corresponding mask bit is set. If the mask bit is clear then the predicate register is not modified. The rotating predicates are transferred as if CFM.rrb.pr were zero. The actual value in CFM.rrb.pr is ignored and remains unchanged. The move from predicate instruction (`mov =pr`) transfers the entire predicate register file into a general register target.

In addition, instructions are defined to move values between the general register file and the user mask (`mov psr.um=` and `mov =psr.um`). The `sum` and `rum` instructions set and reset the user mask. The user mask is the non-privileged subset of the Process Status Register (PSR).

The `mov =pmd[]` instruction is defined to move from a performance monitor data (PMD) register to a general register. If the operating system has not enabled reading of performance monitor data registers in user level then all zeroes are returned. The `mov =cpuid[]` instruction is defined to move from a processor identification register to a general register.

The `mov =ip` instruction is provided for copying the current value of the instruction pointer (IP) into a general register.

Instructions are provided to copy an immediate value into a DAHR and to copy the current value of a DAHR into a general register.



V2-A Section 4.1.1.2

4.1.1.2 Translation Cache (TC)

The Translation Cache (TC) is an implementation-specific structure defined to hold the large working set of dynamic translations for memory references (including IA-32). Please see the processor-specific documentation for further information on Itanium processor TC implementation details. The processor directly controls the replacement policy of all TC entries.

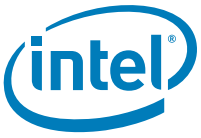
Entries are installed by software into the translation cache with the Insert Data Translation Cache (`itc.d`) and Insert Instruction Translation Cache (`itc.i`) instructions. The Purge Translation Cache Local (`ptc.l`) instruction purges all ITC/DTC entries in the local processor that match the specified virtual address range and region identifier. Purges of all ITC/DTC entries matching a specified virtual address range and region identifier among all processors in a TLB coherence domain can be globally performed with the Purge Translation Cache Global (`ptc.g`, `ptc.ga`) instruction. The TLB coherence domain covers at least the processors on the same local bus on which the purge was broadcast. Propagation between multiple TLB coherence domains is platform dependent. Software must handle the case where a purge does not propagate to all processors in a multiprocessor system. Translation cache purges do not invalidate TR entries.

All the entries in a local processor's ITC and DTC can be purged of all entries with a sequence of Purge Translation Cache Entry (`ptc.e`) instructions. A `ptc.e` does not propagate to other processors.

In all processor models, the translation cache has at least 1 instruction and 1 data entry in addition to the specified 8 instruction and 8 data translation registers. Implementations are free to implement translation cache arrays of larger sizes. Implementations may also choose to implement additional hierarchies for increased performance. At least one translation cache level is required to support all implemented page sizes. Additional hierarchy levels may or may not be performance optimized for the preferred page size specified by the virtual region, may be set-associative or fully associative, and may support a limited set of page sizes. Please see the processor-specific documentation for further information on the Itanium processor implementation details of the translation cache.

The translation cache is managed by both software and hardware. In general, software cannot assume any entry installed will remain, nor assume the lifetime of any entry since replacement algorithms are implementation specific. The processor may discard or replace a translation at any point in time for any reason (subject to the forward progress rules below). TC purges may remove more entries than explicitly requested. In the presence of a processor hardware error, the processor may remove TC entries and optionally raise a Corrected Machine Check Interrupt.

In order to ensure forward progress for Itanium architecture-based code, the following rules must be observed by the processor and software.

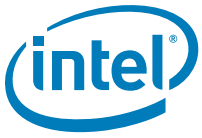


Software may insert multiple translation cache entries per TLB fault, provided that only the last installed translation is required for forward progress.

- The processor may occasionally invalidate the last TC entry inserted. The processor must eventually guarantee visibility of the last inserted TC entry to all references while PSR.ic is zero. The processor must eventually guarantee visibility of the last inserted TC entry until an `rfi` sets PSR.ic to 1 and at least one instruction is executed with PSR.ic equal to 1, and completes without a fault or interrupt. The last inserted TC entry may be occasionally removed before this point, and software must be prepared to re-insert the TC entry on a subsequent fault. For example, eager or mandatory RSE activity, speculative VHPT walks, or other interruptions of the restart instruction may displace the software-inserted TC entry, but when software later re-inserts the same TC entry, the processor must eventually complete the restart instruction to ensure forward progress, even if that restart instruction takes other faults which must be handled before it can complete. If PSR.ic is set to 1 by instructions other than `rfi`, the processor does not guarantee forward progress.
- If psr.ic is to be set to 1 after the itc instruction with an ssm or mov-to-psr instruction, a srlz.i instruction is required between the itc and the instruction that sets psr.ic. This srlz.i instruction must be in a separate instruction group from the one containing the itc, and must be in a separate instruction group from the one containing the instruction that sets psr.ic to 1.
- If software inserts an entry into the TLB with an overlapping entry (same or larger size) in the VHPT, and if the VHPT walker is enabled, forward progress is not guaranteed. See “VHPT Searching” on page 2:63.
- Software may only make references to memory with physical addresses or with virtual addresses which are mapped with TRs, or to addresses mapped by the just-inserted translation, between the insertion of a TC entry, and the execution of the instruction with PSR.ic equal to 1 which is dependent on that entry for forward progress. Software may also make repeated attempts to execute the same instruction with PSR.ic equal to 1. If software makes any other memory references than these, the processor does not guarantee forward progress.
- Software must not defeat forward progress by consistently displacing a required TC entry through a global or local translation cache purge.

IA-32 code has more stringent forward progress rules that must be observed by the processor and software. IA-32 forward progress rules are defined in Section 10.6.3, “IA-32 TLB Forward Progress Requirements” on page 2:261.

The translation cache can be used to cache TR entries if the TC maintains the instruction vs. data distinction that is required of the TRs. A data reference cannot be satisfied by a TC entry that is a cache of an instruction TR entry, nor can an instruction reference be satisfied by a TC entry that is a cache of a data TR entry. This approach can be useful in a multi-level TLB implementation.



V2-B Part 2, Sections 5.2.1.1 and 5.2.2.1

5.2.1.1 TR Insertion

To insert a translation into a TR, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[1]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.
4. Place the slot number of the instruction or data TR into which the translation is be inserted into a general register.
5. Place the base physical address of the translation into another general register.
6. Using the general registers from steps 4. and 5., execute the `itr.i` or `itr.d` instruction.
7. *If PSR.ic was 1 in step 1 and software wishes to restore the PSR.ic back to its original value of 1 with an ssm or a mov-to-psr instruction, then it is required to execute a srlz.i instruction before restoring PSR.ic back to a 1.*

A data or instruction serialization operation must be performed after the insert (for `itr.d` or `itr.i`, respectively) before the inserted translation can be referenced.

Software may insert a new translation into a TR slot already occupied by another valid translation. However, software must perform a TR purge to ensure that the overwritten translation is no longer present in any of the processor's TLB structures.

Instruction TR inserts will purge any instruction TC entries which overlap the inserted translation, and may purge any data TC entries which overlap it. Data TR inserts will purge any data TC entries which overlap the inserted translation and may purge any instruction TC entries which overlap it.

Software may insert the same (or overlapping) translation into both the instruction TRs and the data TRs. This may be desirable for locked pages which contain both code and data, for example.

5.2.2.1 TC Insertion\ \

To insert a TC entry, software performs the following steps:

1. If PSR.ic is 1, clear it and execute a `srlz.d` instruction to ensure the new value of PSR.ic is observed.
2. Place the base virtual address of the translation into the IFA control register.[1]
3. Place the page size of the translation into the ps field of the ITIR control register. If protection key checking is enabled, also place the appropriate translation key into the key field of the ITIR control register. See below for an explanation of protection keys.

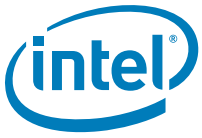
---

1. The upper 3 bits (VRN) of this address specify a region register whose contents are inserted along with the rest of the translation. See Section 5.1.1 for details.

4. Place the base physical address of the translation into a general register.
5. Using the general register from step 4., execute the `itc.i` or `itc.d` instruction.
6. *If PSR.ic was 1 in step 1 and software wishes to restore the PSR.ic back to its original value of 1 with an ssm or a mov-to-psr instruction, then it is required to execute a srlz.i instruction before restoring PSR.ic back to a 1.*

A data or instruction serialization operation must be performed after the insert (for `itc.d` or `itc.i`, respectively) before the inserted translation can be referenced.

Instruction TC inserts always purge overlapping instruction TCs and may purge overlapping data TCs. Likewise, data TC inserts always purge overlapping data TCs and may purge overlapping instruction TCs.

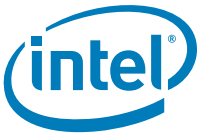


V2-C Table 4-2

Table 4-2. Purged behavior of VHPT Inserts

Case	VRN bits used for TLB searching on VHPT insert						VRN bits not used for TLB searching on VHPT insert		
	VRN Match			No VRN Match					
	Insert?	Purge?	Machine Check?	Insert?	Purge?	Machine Check?	Insert?	Purge?	Machine Check?
[ID]VHPT overlaps [ID]TC[a]	Must[b]	Must[c]	Must not[d]	Must	May	Must not	Must	Must	Must not
[ID]VHPT overlaps [DI]TC[e]	Must	May[f]	Must not	Must	May	Must not	Must	May	Must not
[ID]VHPT overlaps [ID]TR	May[g]	May	Must[h]	May	May[i]	May[i]	May	May	Must
[ID]VHPT overlaps [DI]TR	Must	Must not	Must not	Must	Must not	Must not	Must	Must not	Must not

a. Bracketed notation is intended to specify TC and TR overlaps in the same stream, e.g. `itc.i` and ITC.
b. Must Insert: requires that the translation specified by the operation is inserted into a TC or TR as appropriate. For `itc` and VHPT walker inserts, there is no guarantee to software that the entry will exist in the future, with the exception of the relevant forward-progress requirements specified in Section 4.1.1.2, “Translation Cache (TC)”.
c. Must Purge: requires that all partially or fully overlapped translations are removed prior to the insert or purge operation.
d. Must not Machine Check: indicates that a processor does not cause a Machine Check abort as a result of the operation.
e. Bracketed notation is intended to specify TC and TR overlaps in the opposite stream, e.g. `itc.i` and DTC.
f. 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.
g. May Insert: indicates that the translation specified by the operation may be inserted into a TC. However, software must not rely on the insert.
h. Must Machine Check: indicates that a processor will cause a Machine Check abort if an attempt is made to insert or purge a partially or fully overlapped translation. The Machine Check abort may not be delivered synchronously with the TLB insert or purge operation itself, but is guaranteed to be delivered, at the latest, on a subsequent instruction serialization operation.
i. If the processor actually purges the TR entry, then a Machine Check is required.



V2-D Section 4.1.1.5

4.1.1.5 Translation Insertion Format

Figure 4-5 shows the register interface to insert entries into the TLB. TLB insertions are performed by issuing the Insert Translation Cache (`itc.d`, `itc.i`) and Insert Translation Registers (`itr.d`, `itr.i`) instructions. The first 64-bit field containing the physical address, attributes and permissions is supplied by a general purpose register operand. Additional protection key and page size information is supplied by the Interruption TLB Insertion Register (ITIR). The Interruption Faulting Address register (IFA) specifies the virtual address for instruction and data TLB inserts. ITIR and IFA are defined in "Control Registers" on page 29. The upper 3 bits of IFA (VRN bits{63:61}) select a virtual region register that supplies the RID field for the TLB entry. The RID of the selected region is tagged to the translation as it is inserted into the TLB.

Reserved fields or encodings are checked as follows:

- The GR[r] value is checked when a TLB insert instruction is executed, and if reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the TLB insert instruction. If GR[r]{0} is zero (not-present Translation Insertion Format), the rest of GR[r] is ignored.
- The RR[vrn] value is checked when a mov to RR instruction is executed, and if reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the mov to RR instruction.
- The ITIR value is checked either when a mov to ITIR instruction is executed, or when a TLB insert instruction is executed, depending on the processor implementation. If reserved fields or reserved encodings are used, a Reserved Register/Field fault is raised on the mov to ITIR or TLB insert instruction. In implementations where ITIR is checked on a TLB insert instruction, ITIR{63:32} and ITIR{31:8} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
- The IFA value is checked either when a mov to IFA instruction is executed, or when a TLB insert instruction is executed, depending on the processor implementation. If an unimplemeted virtual address is used, an Unimplemented Data Address fault is raised on the mov to IFA or TLB insert instruction.

Software must issue an instruction serialization operation to ensure installs into the ITLB are observed by dependent instruction fetches and a data serialization operation to ensure installs into the DTLB are observed by dependent memory data references.

Figure 4-5. Translation Insertion Format

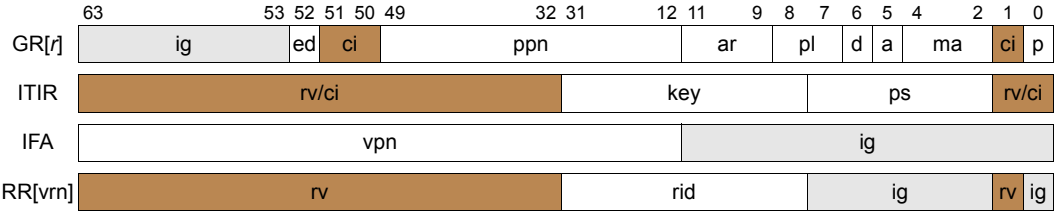


Table 4-3 describes all the translation interface fields.

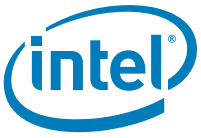


Table 4-3. Translation Interface Fields

TLB Field	Source Field	Description
ci	GR[*r*]{1,51:50}	Checked on Insert – Checked on a TLB insert instruction. If reserved fields or encodings are used, a Reserved Register/Field fault is raised on the TLB insert instruction.
rv/ci	ITIR{1:0,63:32}	Reserved/Checked on Insert – Depending on implementation, may be reserved (checked on a mov to ITIR instruction) or checked on a TLB insert instruction. If reserved fields or encodings are used, a Reserved Register/ Field fault is raised on the mov to ITIR or TLB insert instruction. In implementations where ITIR is checked on a TLB insert instruction, ITIR{63:32} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
rv	RR[vrn]{1,63:32}	Reserved – Checked on a mov to RR instruction. If reserved fields or encodings are used, a Reserved Register/Field fault is raised on the mov to RR instruction.
p	GR[*r*]{0}	Present bit – When 0, references using this translation cause an Instruction or Data Page Not Present fault. Most other fields are ignored by the processor, see Figure 4-6 for details. This bit is typically used to indicate that the mapped physical page is not resident in physical memory. The present bit is not a valid bit. For each TLB entry, the processor maintains an additional hidden valid bit indicating if the entry is enabled for matching.
ma	GR[*r*]{4:2}	Memory Attribute – describes the cacheability, coherency, write-policy and speculative attributes of the mapped physical page. See “Memory Attributes” on page 2:75 for details.
a	GR[*r*]{5}	Accessed Bit – When 0 and PSR.da is 0, data references to the page cause a Data Access Bit fault. When 0 and PSR.ia is 0, instruction references to the page cause an Instruction Access Bit fault. When 0, IA-32 references to the page cause an Instruction or Data Access Bit fault. This bit can trigger a fault on reference for tracing or debugging purposes. The processor does not update the Accessed bit on a reference.
d	GR[*r*]{6}	Dirty Bit – When 0 and PSR.da is 0, Intel Itanium store or semaphore references to the page cause a Data Dirty Bit fault. When 0, IA-32 store or semaphore references to the page cause a Data Dirty Bit fault. The processor does not update the Dirty bit on a store or semaphore reference.
pl	GR[*r*]{8:7}	Privilege Level – Specifies the privilege level or promotion level of the page. See “Page Access Rights” on page 2:56 for complete details.
ar	GR[*r*]{11:9}	Access Rights – page granular read, write and execute permissions and privilege controls. See “Page Access Rights” on page 2:56 for details.
ppn	GR[*r*]{49:12}	Physical Page Number – Most significant bits of the mapped physical address. Depending on the page size used in the mapping, some of the least significant PPN bits are ignored.
ig	GR[*r*]{63:53} IFA{11:0}, RR[vrn]{0,7:2}	available – Software can use these fields for operating system defined parameters. These bits are ignored when inserted into the TLB by the processor.
ed	GR[*r*]{52}	Exception Deferral – For a speculative load that results in an exception, the speculative load’s instruction page TLB.ed bit is one of the conditions which determines whether the exception must be deferred. See “Deferral of Speculative Load Faults” on page 105 for complete details. This bit is ignored in the data TLB for data memory references and for IA-32 memory references.
ps	ITIR{7:2}	Page Size – Page size of the mapping. For page sizes larger than 4K bytes the low-order bits of PPN and VPN are ignored. Page sizes are defined as 2^{ps} bytes. See “Page Sizes” on page 2:57 for a list of supported page sizes.

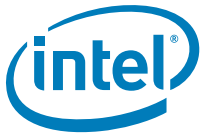
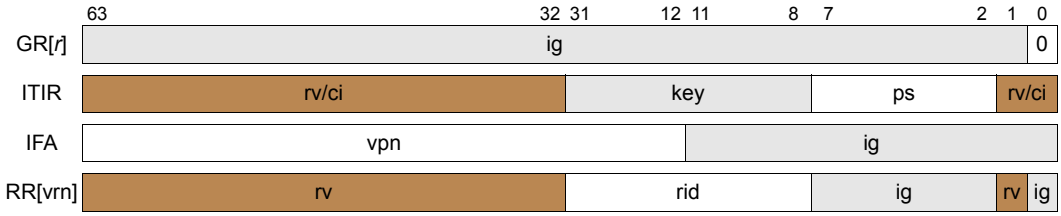


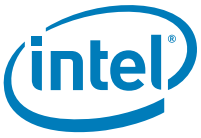
Table 4-3. Translation Interface Fields (Continued)

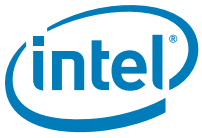
TLB Field	Source Field	Description
key	ITIR{31:8}	Protection Key – Uniquely tags the translation to a protection domain. If a translation's Key is not found in the Protection Key Registers (PKRs), access is denied and a Data or Instruction Key Miss fault is raised. See "Protection Keys" on page 2:59 for complete details. In implementations where ITIR is checked on a TLB insert instruction, ITIR{31:8} may be ignored if GR[r]{0} is zero (not-present Translation Insertion Format).
vpn	IFA{63:12}	Virtual Page Number – Depending on a translation's page size, some of the least-significant VPN bits specified are ignored in the translation process. VPN{63:61} (VRN) selects the region register.
rid	RR[VRN].rid	Virtual Region Identifier – On TLB inserts the Region Identifier selected by VPN{63:61} (VRN) is used as additional match bits for subsequent accesses and purges (much like vpn bits).

The format in Figure 4-6 is defined for not-present translations (P-bit is zero).

Figure 4-6. Translation Insertion Format – Not Present







V2-E Section 5.8.3.9

5.8.3.9 Local Redirection Registers (LRR0-1 – CR80,81)

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.

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. The LRR fields are defined in Figure 5-14 and Table 5-15.

Figure 5-14. Local Redirection Register (LRR – CR80,81)

Table 5-15. Local Redirection Register Fields

Field	Bits	Description	
vector	7:0	External interrupt vector number to use when generating an interrupt for this entry. For INT delivery mode, allowed vector values are 0, 2, or 16-255. All other vectors are ignored and reserved for future use. For all other delivery modes this field is ignored.	
dm	10:8	000	INT – pend an external interrupt for the vector number specified by the vector field in LRR. Allowed vector values are 0, 2, or 16-255. All other vector numbers are ignored and reserved for future use.
		001	reserved
		010	PMI – pend a Platform Management Interrupt Vector number 0 for system firmware. The vector field is ignored.
		011	reserved
		100	NMI – pend a Non-Maskable Interrupt. This interrupt is pended at external interrupt vector number 2. The vector field is ignored.
		101	INIT – pend an Initialization Interrupt for system firmware. The vector field is ignored.
		110	reserved
		111	ExtINT – pend an Intel 8259A-compatible interrupt. This interrupt is delivered at external interrupt vector number 0. For details on servicing ExtINT external interrupts see “Interrupt Acknowledge (INTA) Cycle” on page 2:130. The vector field is ignored.
ipp	13	Interrupt Pin Polarity – specifies the polarity of the interrupt signal. When 0, the signal is active high. When 1, the signal is active low.	

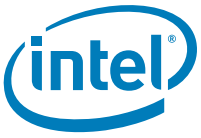
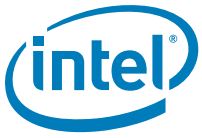


Table 5-15. Local Redirection Register Fields (Continued)

Field	Bits	Description
tm	15	Trigger Mode – When 0, specifies edge sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an interrupt for the specified vector corresponding to the dm field. The pending interrupt indication is cleared by software servicing the interrupt. When 1, specifies level sensitive interrupts. If the m field is 0, assertion of the corresponding LINT pin pends an external interrupt for the specified vector. Deassertion of the corresponding LINT pin clears the pending interrupt indication. The processor has undefined behavior if the dm and tm fields specify level sensitive PMIs or INITs.
m	16	Mask – When 1, edge or level occurrences of the local interrupt pins are discarded and not pended. When 0, edge or level occurrences of local interrupt pins are pended.



V2-F Section 7.2

7.2 Performance Monitoring

Performance monitors allow processor events to be monitored by programmable counters or give an external notification (such as a pin or transaction) on the occurrence of an event. Monitors are useful for tuning application, operating system and system performance. Two sets of performance monitor registers are defined. Performance Monitor Configuration (PMC) registers are used to control the monitors. Performance Monitor Data (PMD) Registers either provide data values from the monitors, or hold data values used by the PMU. ~~The performance monitors can record performance values from either the IA-32 or Itanium instruction set.~~

As shown in Figure 7-3, all processor implementations provide at least four performance counters (PMC/PMD[4]..PMC/PMD[7] pairs), and four performance counter overflow status registers (PMC[0]..PMC[3]). Performance monitors are also controlled by bits in the processor status register (PSR), the default control register (DCR) and the performance monitor vector register (PMV). Processor implementations may provide additional implementation-dependent PMC and PMD registers to increase the number of “generic” performance counters (PMC/PMD pairs). The remainder of the PMC and PMD register set is implementation dependent.

Event collection for implementation-dependent performance monitors is not specified by the architecture. Enabling and disabling functions are implementation dependent. For details, consult processor-specific documentation.

Processor implementations may not populate the entire PMC/PMD register space. Reading of an unimplemented PMC or PMD register returns zero. Writes to unimplemented PMC or PMD registers are ignored; i.e., the written value is discarded.

Writes to PMD and PMC and reads from PMC are privileged operations. At non-zero privilege levels, these operations result in a Privileged Operation fault, regardless of the register address.

Reading of PMD registers by non-zero privilege level code is controlled by PSR.sp. When PSR.sp is one, PMD register reads by non-zero privilege level code return zero.

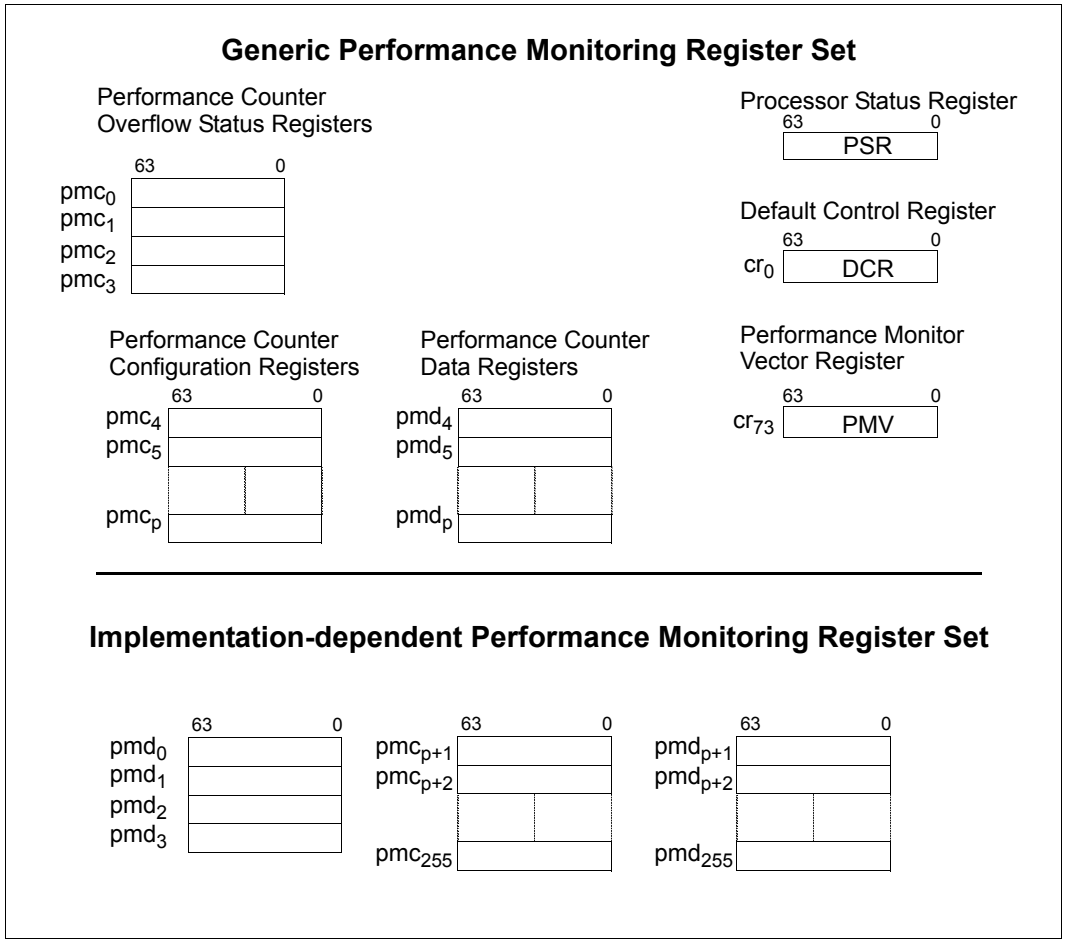
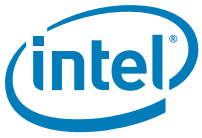


Figure 7-3. Performance Monitor Register Set



V2-G Sections 7.2.1 and 7.2.3

7.2.1 Generic Performance Counter Registers

Generic performance counter registers are PMC/PMD pairs that contiguously populate the PMC/PMD name space starting at index 4. At least 4 performance counter register pairs (PMC/PMD[4]..PMC/PMD[7]) are implemented in all processor models. Each counter can be configured to monitor events for any combination of privilege levels and one of several event metrics. The number of performance counters is implementation specific. The figures and tables use the symbol “p” to represent the index of the last implemented generic PMC/PMD pair. The bit-width W of the counters is also implementation specific.

A counter overflow interrupt occurs when the counter wraps; i.e., a carry out from bit W-1 is detected. Counter overflow interrupts are edge-triggered; that is, the event of a counter incrementing and causing carry out from bit W-1 thus setting the overflow bit and the freeze bit, generates one PMU interrupt. Provided that software does not clear the freeze bit, while either or both of PSR.up and pp are 1, without also clearing the overflow bit (before or concurrent with the write to the freeze bit), no further interrupts are generated based on the fact that the carry out had been earlier detected.

Figure 7-4 and Figure 7-5 show the fields in PMD and PMC respectively, while Table 7-3 and Table 7-4 describe the fields in PMD and PMC respectively.

Figure 7-4. Generic Performance Counter Data Registers (PMD[4]..PMD[p])

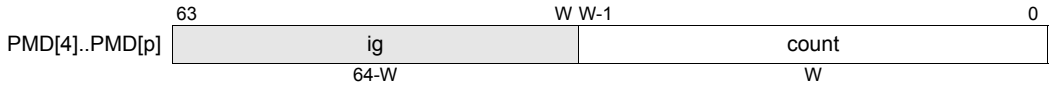
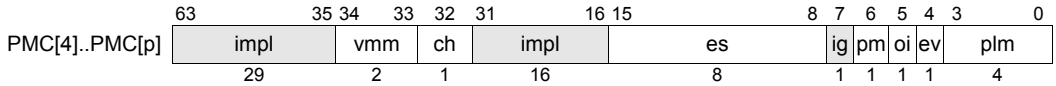


Table 7-3. Generic Performance Counter Data Register Fields

Field	Bits	Description
ig	63:W	Writes are ignored. Reads return 0.
count	W-1:0	Event Count. The counter is defined to overflow when the count field wraps (carry out from bit W-1).

Some implementations do not treat the upper, unimplemented bits of PMDs as ignored bits on reads, but rather return a copy of bit W-1 in these bit positions so that count values appear as if they were sign extended. Subsequent implementations will return 0 for these bits on reads.

Figure 7-5. Generic Performance Counter Configuration Register (PMC[4]..PMC[p])



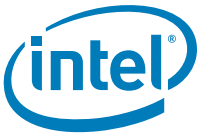
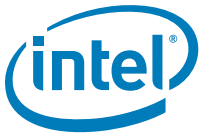


Table 7-4. Generic Performance Counter Configuration Register Fields (PMC[4]..PMC[p])

Field	Bits	Description
plm	3:0	Privilege Level Mask – controls performance monitor operation for a specific privilege level. Each bit corresponds to one of the 4 privilege levels, with bit 0 corresponding to privilege level 0, bit 1 with privilege level 1, etc. A bit value of 1 indicates that the monitor is enabled at that privilege level. Writing zeros to all plm bits effectively disables the monitor. In this state, the corresponding PMD register(s) do not preserve values, and the processor may choose to power down the monitor.
ev	4	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.
oi	5	Overflow interrupt – If 1, when the monitor overflows, a Performance Monitor Interrupt is raised and the performance monitor freeze bit (PMC[0].fr) is set. If 0, no interrupt is raised and the performance monitor freeze bit (PMC[0].fr) remains unchanged. Overflow occurs when a carry out from bit W-1 is detected. See "Performance Monitor Overflow Status Registers (PMC[0]..PMC[3])" for details on configuring interrupt vectors.
pm	6	Privileged monitor – When 0, the performance monitor is configured as a user monitor, and enabled by PSR.up. When PMC.pm is 1, the performance monitor is configured as a privileged monitor, enabled by PSR.pp, and the corresponding PMD can only be read by privileged software.
ig	7	ignored
es	15:8	Event select – selects the performance event to be monitored. Actual event encodings are implementation dependent. Some processor models may not implement all event select (es) bits. At least one bit of es must be implemented on all processors. Unimplemented es bits are ignored.
impl	31:16	Implementation-specific bits – Reads from implemented bits return implementation-dependent values. For portability, software should write what was read; i.e., software may not use these bits as storage. Hardware will ignore writes to unimplemented bits.
ch	32	Count Halted State - controls performance monitor operation depending on which power state the processor is in (see Figure 11-8 on page 2:316). If set to 0, the processor will capture events that occur while the processor is in the NORMAL/LOW-POWER state. If set to 1, the processor will capture events that occur while the processor is in one of the HALT states." NOTE: This bit is not supported on all processor implementations. Software can call PAL_PROC_GET_FEATURES to determine the availability of this feature, see "PAL_PROC_GET FEATURES" page 121 for details. The bit is implementation specific when this feature is not supported.
vmm	34:33	Virtual Machine Mask - controls performance monitor operation depending on whether or not the processor is operating in Virtual Machine mode (as reflected by the value in PSR.vm). 00 - Performance monitoring is enabled regardless of the state of PSR.vm. 01 - Performance monitoring is enabled if PSR.vm == 0. 10 - Performance monitoring is enabled if PSR.vm == 1. 11 - Ignored NOTE: This field is not supported on all processor implementations. Software can call PAL_PROC_GET_FEATURES to determine the availability of this feature, see "PAL_PROC_GET FEATURES" page 121 for details. The field is implementation specific when this feature is not supported
impl	63:35	Implementation-specific bits – Reads from implemented bits return implementation-dependent values. For portability, software should write what was read; i.e., software may not use these bits as storage. Hardware will ignore writes to unimplemented bits.



Event collection is controlled by the Performance Monitor Configuration (PMC) registers and the processor status register (PSR). Five PSR fields (PSR.up, PSR.pp, PSR.cpl, PSR.vm and PSR.sp) and the performance monitor freeze bit (PMC[0].fr) affect the behavior of all generic performance monitor registers. Finer, per monitor, control of generic performance monitors is provided by three PMC register fields (PMC[i].plm, PMC[i].vmm and PMC[i].pm). Event collection for a generic monitor is enabled under the following constraints:

- Generic Monitor Enable[i] = (not PMC[0].fr) and PMC[i].plm[PSR.cpl] and (not (PMC[i].vmm == 01 and PSR.vm)) and (not (PMC[i].vmm == 10 and not PSR.vm)) and ((not (PMC[i].pm) and PSR.up) or (PMC[i].pm and PSR.pp))

Generic performance monitor data registers (PMD[i]) can be configured to be user readable (useful for user level sampling and tracing user level processes) by setting the PMC[i].pm bit to 0. All user-configured monitors can be started and stopped synchronously by the user mask instructions (`rum` and `sum`) by altering PSR.up. User-configured monitors can be secured by setting PSR.sp to 1. A user-configured secured monitor continues to collect performance values; however, reads of PMD, by non-privileged code, return zeros until the monitor is unsecured.

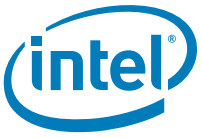
Monitors configured as privileged (PMC[i].pm is 1) are accessible only at privilege level 0; otherwise, reads return zeros. All privileged monitors can be started and stopped synchronously by the system mask instructions (`rsm` and `ssm`) by altering PSR.pp. Table 7-5 summarizes the effects of PSR.sp, PMC[i].pm, and PSR.cpl on reading PMD registers.

Updates to generic PMC registers and PSR bits (up, pp, is, sp, cpl, vm) require implicit or explicit data serialization prior to accessing an affected PMD register. The data serialization ensures that all prior PMD reads and writes as well as all prior PMC writes have completed.

Table 7-5. Reading Performance Monitor Data Registers

PSR.sp	PMC[i].pm	PSR.cpl	PMD Reads Return
0	0	0	PMD value
0	1	0	PMD value
1	0	0	PMD value
1	1	0	PMD value
0	0	>0	PMD value
0	1	>0	0
1	0	>0	0
1	1	>0	0

Generic PMD counter registers may be read by software without stopping the counters. Under normal counting conditions (PMC[0].fr is zero and has been serialized), the processor guarantees that a sequence of reads of a given PMD will return non-decreasing values corresponding to the program order of the reads. Under frozen count conditions (PMC[0].fr is one and has been serialized), the counters are unchanging and ordering is irrelevant. When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed.



Software must accept a level of sampling error when reading the counters due to various machine stall conditions, interruptions, and bus contention effects, etc. The level of sampling error is implementation specific. More accurate measurements can be obtained by disabling the counters and performing an instruction serialize operation for instruction events or data serialize operation for data events before reading the monitors. Other (non-counter) implementation-dependent PMD registers can only be read reliably when event monitoring is frozen (PMC[0].fr is one).

For accurate PMD reads of disabled counters, data serialization (implicit or explicit) is required between any PMD read and a subsequent `ssm` or `sum` (that could toggle PSR.up or PSR.pp from 0 to 1), or a subsequent `epc`, demoting `br.ret` or branch to IA-32 (`br.ia`) (that could affect PSR.cpl or PSR.is). Note that implicit post-serialization semantics of `sum` do not meet this requirement.

Table 7-6 defines the instructions used to access the PMC and PMD registers.

Table 7-6. Performance Monitor Instructions

Mnemonic	Description	Operation	Instr Type	Serialization Required
mov pmd[r_3] = r_2	Move to performance monitor data register	PMD[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmd[r_3]	Move from performance monitor data register	GR[r_1] ← PMD[GR[r_3]]	M	none[a]
mov pmc[r_3] = r_2	Move to performance monitor configure register	PMC[GR[r_3]] ← GR[r_2]	M	data/inst
mov r_1 = pmc[r_3]	Move from performance monitor configure register	GR[r_1] ← PMC[GR[r_3]]	M	none

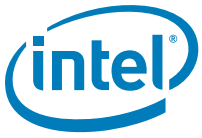
a. When the freeze bit is in-flight, whether counters count events and reads return non-decreasing values is implementation dependent. Instruction serialization is required to ensure that the behavior specified by PMC[0].fr is observed.

7.2.3 Performance Monitor Events

The set of monitored events is implementation-specific. All processor models are required to provide at least two events:

1. The number of retired instructions. These are defined as all instructions which execute without a fault, including nops and those which were predicated off. Generic counters configured for this event count only when the processor is in the NORMAL or LOW-POWER state (see Figure 11-8 on page 2:316).
2. The number of processor clock cycles. Dependent on the setting of PMC.ch, generic counters may be configured to count this event when the processor is either in the NORMAL/LOW-POWER state or one of the HALT states (see Figure 11-8 on page 2:316).

Events may be monitorable only by a subset of the available counters. PAL calls provide an implementation-independent interface that provides information on the number of implemented counters, their bit-width, the number and location of other (non-counter) monitors, etc.



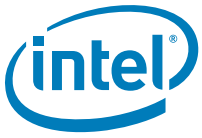
V2-H Section 11.5.2

11.5.2 PALE_PMI Exit State

The state of the processor on exiting PALE_PMI is:

- GRs: The contents of non-banked general registers are unchanged from the time of the interruption.
    - Bank 1 GRs: The contents of all bank one general registers are unchanged from the time of the interruption.
    - Bank 0:GR16-23: The contents of these bank zero general registers are unchanged from the time of the interruption.
    - Bank 0:GR24-31: contain parameters which PALE_PMI passes to SALE_PMI:
        - GR24 contains the value decoded as follows:
            - Bits 7-0: PMI Vector Number
            - Bit 63-8: Reserved
        - GR25 contains the value of the min-state save area address stored in XR0.
        - GR26 contains the value of saved RSC. The contents of this register shall be preserved by SAL PMI handler.
        - GR27 contains the value of saved B0. The contents of this register shall be preserved by SAL PMI handler.
        - GR28 contains the value of saved B1. The contents of this register shall be preserved by SAL PMI handler.
        - GR29 contains the value of the saved predicate registers. The contents of this register shall be preserved by SAL PMI handler
        - GR30-31 are scratch registers available for use.
- FRs: The contents of all floating-point registers are unchanged from the time of the interruption.
- Predicates: The contents of all predicate registers are undefined and available for use.
- BRs: The contents of all branch registers are unchanged, except the following which contain the defined state.
    - BR1 is undefined and available for use.
    - BR0 PAL PMI return address.
- ARs: The contents of all application registers are unchanged from the time of the interruption, except the RSE control register (RSC) and the ITC and RUC counters. The RSC.mode field will be set to 0 (enforced lazy mode) while the other fields in the RSC are unchanged. The ITC register will not be directly modified by PAL, but will continue to count during the execution of the PMI handler. The RUC register will not be directly modified by PAL, but will continue to count during the execution of the PMI handler while the processor is active.
- CFM: The contents of the CFM register are unchanged from the time of the interruption. On resuming from the PMI handler, if SAL has done a cover instruction (e.g., to allow it to use stacked registers itself), then the IFS should contain the value captured by that cover, which will correctly restore CFM when PAL resumes, otherwise CFM should be unchanged.
- RSE: Is in enforced lazy mode, and stacked registers are unchanged from the time of the interruption.

- PSR: PSR.mc, PSR.mfl, PSR.mfh, and PSR.pk are unchanged; all other bits are 0.
- CRs: The contents of all control registers are unchanged from the time of the interruption with the exception of interruption resources, which are described below.
- RRs: The contents of all region registers are unchanged from the time of the interruption.
- PKRs: The contents of all protection key registers are unchanged from the time of the interruption.
- DBR/IBRs: The contents of all breakpoint registers are unchanged from the time of the interruption.
- PMCs/PMDs: The contents of the PMC registers are unchanged from the time of the PMI. The contents of the PMD registers are not modified by PAL code, but may be modified if events it is monitoring are encountered
- Cache: The processor internal cache is not specifically modified by the PMI handler but may be modified due to normal cache activity of running the handler code.
- TLB: The TCs are not modified by the PALE_PMI handler and the TRs are unchanged from the time of the interruption.
- Interruption Resources:
    - IRRs: The contents of IRRs are unchanged from the time of the interruption.
    - IIP and IPSR contain the value of IP and PSR. The IFS.v bit is reset to 0.
    - IFS: The contents of IFS are unchanged from the time of the interruption. On resuming from the PMI handler, if SAL has done a cover instruction (e.g., to allow it to use stacked registers itself), then IFS should contain the value captured by that cover instruction.



V2-I Section 11.6.1.3

11.6.1.3 PAL Interfaces for P-states

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 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 2:395. 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.

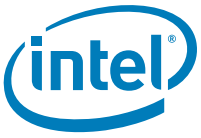
If hardware has the ability to either preempt a previous in-progress P-state transition, or to queue successive P-state requests while the first request is in transition, then the implementation has a pre-emptive policy for P-state request handling. The architecture also allows for a non-preemptive policy 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.

Implementation-specific event conditions may prevent a PAL_SET_PSTATE request from being accepted (e.g., due to a thermal protection mechanism), in which case the PAL procedure returns a status of *transition failure*. Such events are expected to be rare and to happen only in abnormal situations.

It should be noted that platform power-caps do not cause a PAL_SET_PSTATE request to fail. The requested P-state is registered with PAL, and the procedure returns a status of *transition success*.

SCDD: If the logical processor belongs to a software-coordinated dependency domain, the PAL_SET_PSTATE procedure will change the domain parameters resulting in a transition to the requested P-state for all logical processors in that domain.

HCDD: If the logical processor belongs to a hardware-coordinated dependency domain, the PAL_SET_PSTATE procedure will attempt to change the power/performance characteristics for that logical processor. Since the power/performance characteristics for the domain depend on the P-state settings of the other logical processors in the domain, a PAL_SET_PSTATE call on one logical processor may result in either partial or



complete transition to the requested P-state. In case of partial transition (see Figure 11-11, “Computation of performance_index” on page 49 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 based on P-state transition requests on other logical processors. Software can also approximate the behavior of a SCDD by forcing P-state transitions. See the description of the PAL_SET_PSTATE procedure for more details.

HIDD: 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). A value of 100 in P0 represents the minimum processor performance in the P0 state. For example, if the value returned by the procedure is 80, this indicates that the performance of the logical processor over the last time period was 20% lower than the minimum P0 performance. For processors that support variable P-states, it is possible for a processor to report a number greater than 100, representing that the processor is running at a performance level greater than the minimum P0 performance. For example, if the value returned by the processor is 120, it indicates that the performance of the logical processor over the last time period was 20% higher than the minimum P0 performance. The performance index is measured over the time interval since the last PAL_GET_PSTATE call with a type operand of 1. If the processor supports variable P-state performance then the PAL_PROC_SET_FEATURE procedure can be used to enable or disable this feature. Software may choose, on each invocation of the PAL_GET_PSTATE procedure, whether to reset the internal performance measurement logic; resetting the measurement logic initiates a new *performance_index* count, which is reported when the next PAL_GET_PSTATE procedure call is made. A call to PAL_GET_PSTATE with a *type* operand of 1 resets the performance measurement logic.

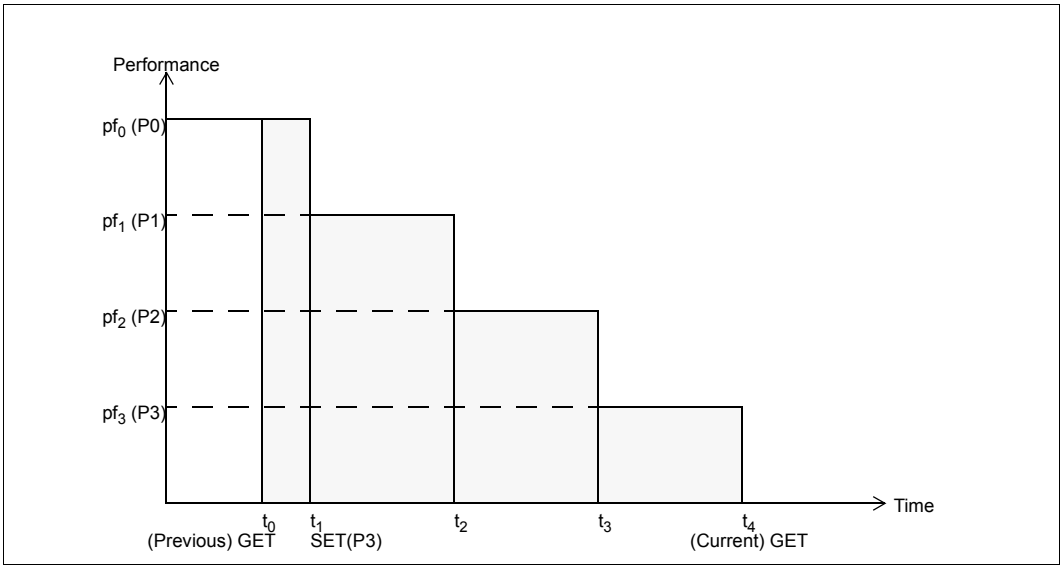
SCDD: If the logical processor belongs to a software-coordinated dependency domain, the performance index returned (for either *type*=0 or 3) corresponds to the target P-state requested by the most recent successful PAL_SET_PSTATE procedure call. No weighted average (*type*=1 or 2) is computed by PAL; calling PAL_GET_PSTATE with *type*=1 or 2 on a SCDD logical processor is undefined.

HCDD: If the logical processor belongs to a hardware-coordinated dependency domain, the performance index returned (*type*=1 or 2) will be a weighted-average sum of the *performance_index* values corresponding to the different P-states that the logical processor was operating in since performance measurement was last reset. 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) =

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

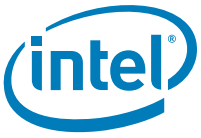
Figure 11-11.Computation of *performance_index*



As seen above, for a HCDD, the PAL_GET_PSTATE procedure allows the caller to get feedback on the dynamic performance of the processor over a software-controlled time period. The caller can use this information to get better system utilization over a subsequent time period by changing the P-state in correlation with the current workload demand. The caller can also use PAL_GET_PSTATE to see the most recent P-state set for this logical processor (*type*=0) and the instantaneous current P-state that the domain parameters are set to (*type*=3). Platform power-caps do not affect either of these return values.

HIDD: If the logical processor belongs to a hardware-independent dependency domain, a weighted-average performance index can be returned by PAL_GET_PSTATE (*type*=1 or 2). Since software could calculate the performance index based on P-states it set, the weighted-average performance index is only of value when factoring in the effect of platform power-caps.

Note that P-state transitions typically do not happen instantaneously. An implementation-specific amount of time is required for a given transition to complete. The computation of the weighted-average *performance_index* may not take into account the fact that transitions of power/performance are gradual, but may be done as though they were instantaneous at the point when the transition starts. The expectation is that any errors in computing the *performance_index* due to non-instantaneous transitions to higher and lower P-states will tend to cancel out, and to the extent that they do not, will be insignificant.



V2-J Section 11.6.1.5

11.6.1.5 Interaction of P-states with HALT State

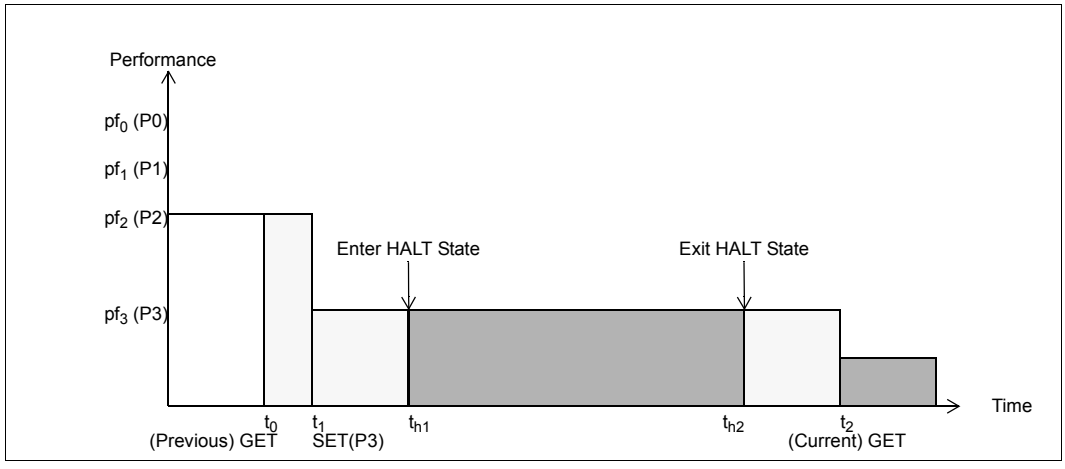
It is possible for a logical processor to enter and exit a HALT state between two consecutive calls to PAL_GET_PSTATE. Since the logical processor is not executing any instructions while in the HALT state, the performance index contribution during this period is essentially 0, and will not be accounted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

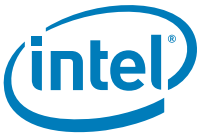
For example, let us assume that the previous PAL_GET_PSTATE procedure was called at time t_0, when the processor was operating in state P2. The previous PAL_SET_PSTATE procedure initiated a transition from P2 to P3 at time t_1. The processor entered HALT state at time t_{h1}, and exited the HALT state 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) =

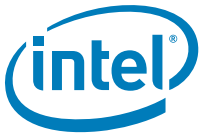
$$\frac{(t_1 - t_0) \times pf_2 + (t_{h1} - t_1) \times pf_3 + (t_2 - t_{h2}) \times pf_3}{(t_2 - t_0) - (t_{h2} - t_{h1})}$$

Figure 11-12. 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.



V2-K Tables 11-7, 11-12 and 11-16

Table 11-7. Processor State Parameter Fields

Field	Bits	Description
rsvd	1:0	Reserved
rz	2	The attempted processor rendezvous was successful if set to 1.
ra	3	A processor rendezvous was attempted if set to 1.
me	4	Distinct multiple errors have occurred, not multiple occurrences of a single error. Software recovery may be possible if error information has not been lost.
mn	5	Min-state save area has been registered with PAL if set to 1.
sy	6	Storage integrity synchronized. A value of 1 indicates that all loads and stores prior to the instruction on which the machine check occurred completed successfully, and that no loads or stores beyond that point occurred. See Table 11-8.
co	7	Continuable. A value of 1 indicates that all in-flight operations from the processor where the machine check occurred were either completed successfully (such as a load), were tagged with an error indication (such as a poisoned store), or were suppressed and will be re-issued if the current instruction stream is restarted. This bit can only be set if the architectural state saved on a machine check is all valid. If this bit is set, then *us* must be cleared to 0, and *ci* must be set to 1. See Table 11-8.
ci	8	Machine check is isolated. A value of 1 indicates that the error has been isolated by the system, it may or may not be recoverable. If 0, the hardware was unable to isolate the error within the CPU and memory hierarchy. The error may have propagated off the system (to persistent storage or the network). If *ci* = 0 then *us* will be set to 1, and *co* and *sy* are cleared to 0. See Table 11-8.
us	9	Uncontained storage damage. A value of 1 indicates the error is contained within the CPU and memory hierarchy, but that some memory locations may be corrupt. If *us* is set to 1, then *co* and *sy* will always be cleared to 0. See Table 11-8.
hd	10	Hardware damage. A value of 1 indicates that as a result of the machine check some non essential hardware is no longer available causing this processor to execute with degraded performance (no functionality has been lost).
tl	11	Trap lost. A value of 1 indicates the machine check occurred after an instruction was executed but before a trap that resulted from the instruction execution could be generated.
mi	12	More information. A value of 1 indicates that more error information about the machine check event is available by making the PAL_MC_ERROR_INFO procedure call.
pi	13	Precise instruction pointer. A value of 1 indicates that the machine logged the instruction pointer to the bundle responsible for generating the machine check.
pm	14	Precise min-state save area. A value of 1 indicates that the min-state save area contains the state of the machine for the instruction responsible for generating the machine check. When this bit is set, the *pi* bit will always be set as well.
dy	15	Processor Dynamic State is valid. (1=valid, 0=not valid) See the PAL_MC_DYNAMIC_STATE procedure call for more information.
in	16	Interruption caused by INIT. (0=machine check, 1=INIT)
rs	17	The RSE is valid. (1=valid, 0=not valid)
cm	18	The machine check has been corrected. (1=corrected, 0=not corrected)
ex	19	A machine check was expected. (1=expected, 0=not expected)
cr	20	Control registers are valid. (1=valid, 0=not valid)

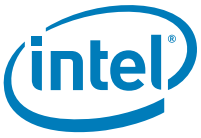


Table 11-7. Processor State Parameter Fields (Continued)

Field	Bits	Description
pc	21	Performance counters are valid. (1=valid, 0=not valid)
dr	22	Debug registers are valid. (1=valid, 0=not valid)
tr	23	Translation registers are valid. (1=valid, 0=not valid)
rr	24	Region registers are valid. (1=valid, 0=not valid)
ar	25	Application registers are valid. (1=valid, 0=not valid)
br	26	Branch registers are valid. (1=valid, 0=not valid)
pr	27	Predicate registers are valid. (1=valid, 0=not valid)
fp	28	Floating-point registers are valid. (1=valid, 0=not valid)
b1	29	Preserved bank one general registers are valid. (1=valid, 0=not valid)
b0	30	Preserved bank zero general registers are valid. (1=valid, 0=not valid)
gr	31	General registers are valid. (1=valid, 0=not valid) (does not include banked registers)
dsize	47:32	Size in bytes of Processor Dynamic State returned by PAL_MC_DYNAMIC_STATE. This value is always a multiple of 8 bytes.
se	48	Shared Error. Machine check corresponds to structure shared by multiple logical processors.
rsvd	58:49	Reserved
cc	59	Cache check. A value of 1 indicates that a cache related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information. This bit must not be set for non-cacheable transaction errors.
tc	60	TLB check. A value of 1 indicates that a TLB related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
bc	61	Bus check. A value of 1 indicates that a bus related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
rc	62	Register file check. A value of 1 indicates that a register file related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
uc	63	Uarch check. A value of 1 indicates that a micro-architectural related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

Table 11-12. Processor State Parameter Fields

Field	Bits	INIT value	Description
rsvd	1:0		Reserved
rz	2	x[a]	The attempted processor rendezvous was successful if set to 1.
ra	3	xa	A processor rendezvous was attempted if set to 1.
me	4	0	Distinct multiple errors have occurred, not multiple occurrences of a single error. Software recovery may be possible if error information has not been lost.
mn	5	xa	Min-state save area has been registered with PAL if set to 1.
sy	6	0	Storage integrity synchronized. A value of 1 indicates that all loads and stores prior to the instruction on which the machine check occurred completed successfully, and that no loads or stores beyond that point occurred. See Table 11-8.
co	7	1	Continuable. A value of 1 indicates that all in-flight operations from the processor where the machine check occurred were either completed successfully (such as a load), were tagged with an error indication (such as a poisoned store), or were suppressed and will be re-issued if the current instruction stream is restarted. This bit can only be set if the architectural state saved on a machine check is all valid. If this bit is set, then *us* must be cleared to 0, and *ci* must be set to 1. See Table 11-8.

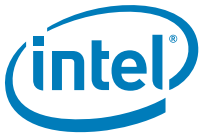


Table 11-12. Processor State Parameter Fields (Continued)

Field	Bits	INIT value	Description
ci	8	1	Machine check is isolated. A value of 1 indicates that the error has been isolated by the system, it may or may not be recoverable. If 0, the hardware was unable to isolate the error within the CPU and memory hierarchy. The error may have propagated off the system (to persistent storage or the network). If *ci* = 0 then *us* will be set to 1, and *co* and *sy* are cleared to 0. See Table 11-8.
us	9	0	Uncontained storage damage. A value of 1 indicates the error is contained within the CPU and memory hierarchy, but that some memory locations may be corrupt. If *us* is set to 1, then *co* and *sy* will always be cleared to 0. See Table 11-8.
hd	10	0	Hardware damage. A value of 1 indicates that as a result of the machine check some non essential hardware is no longer available causing this processor to execute with degraded performance (no functionality has been lost).
tl	11	0	Trap lost. A value of 1 indicates the machine check occurred after an instruction was executed but before a trap that resulted from the instruction execution could be generated.
mi	12	0	More information. A value of 1 indicates that more error information about the machine check event is available by making the PAL_MC_ERROR_INFO procedure call.
pi	13	0	Precise instruction pointer. A value of 1 indicates that the machine logged the instruction pointer to the bundle responsible for generating the machine check.
pm	14	0	Precise min-state save area. A value of 1 indicates that the min-state save area contains the state of the machine for the instruction responsible for generating the machine check. When this bit is set, the *pi* bit will always be set as well.
dy	15	xa	Processor Dynamic State is valid. (1=valid, 0=not valid) See the PAL_MC_DYNAMIC_STATE procedure call for more information.
in	16	1	Interruption caused by INIT. (0=machine check, 1=INIT)
rs	17	xa	The RSE is valid. (1=valid, 0=not valid)
cm	18	0	The machine check has been corrected. (1=corrected, 0=not corrected)
ex	19	0	A machine check was expected. (1=expected, 0=not expected)
cr	20	xa	Control registers are valid. (1=valid, 0=not valid)
pc	21	xa	Performance counters are valid. (1=valid, 0=not valid)
dr	22	xa	Debug registers are valid. (1=valid, 0=not valid)
tr	23	xa	Translation registers are valid. (1=valid, 0=not valid)
rr	24	xa	Region registers are valid. (1=valid, 0=not valid)
ar	25	xa	Application registers are valid. (1=valid, 0=not valid)
br	26	xa	Branch registers are valid. (1=valid, 0=not valid)
pr	27	xa	Predicate registers are valid. (1=valid, 0=not valid)
fp	28	xa	Floating-point registers are valid. (1=valid, 0=not valid)
b1	29	xa	Preserved bank one general registers are valid. (1=valid, 0=not valid)
b0	30	xa	Preserved bank zero general registers are valid. (1=valid, 0=not valid)
gr	31	xa	General registers are valid. (1=valid, 0=not valid) (does not include banked registers)
dsize	47:32	xa	Size in bytes of Processor Dynamic State returned by PAL_MC_DYNAMIC_STATE. This value is always of multiple of 8 bytes
se	48	0	Shared Error. Machine check corresponds to structure shared by multiple logical processors.
rsvd	58:49		Reserved
cc	59	0	Cache check. A value of 1 indicates that a cache related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
tc	60	0	TLB check. A value of 1 indicates that a TLB related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

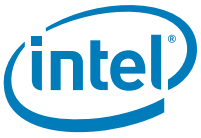


Table 11-12. Processor State Parameter Fields (Continued)

Field	Bits	INIT value	Description
bc	61	0	Bus check. A value of 1 indicates that a bus related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
rc	62	0	Register file check. A value of 1 indicates that a register file related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.
uc	63	0	Uarch check. A value of 1 indicates that a micro-architectural related machine check occurred. See the PAL_MC_ERROR_INFO procedure call for more information.

a. The values of the fields marked with x are set by the PAL INIT handler based on the INIT handling.

Table 11-16. Virtual Processor Descriptor (VPD)

Name	Entries	Offset	Description	Class
vac	1	0	Virtualization Acceleration Control – these control bits enable virtualization acceleration of a particular resource or instruction. See Section 11.7.1.1, “Virtualization Controls” on page 2:331 for details.	Control [always]
vdc	1	8	Virtualization Disable Control – these control bits disable the virtualization of a particular resource or instruction. See Section 11.7.1.1, “Virtualization Controls” on page 2:331 for details.	Control [always]
virt_env_vaddr	1	16	PAL Virtual Environment Buffer Address – this field stores the host virtual address of the virtual environment which the virtual processor belongs to. The value in this field must be the same as the *vbase_addr* field during PAL_VP_INIT_ENV call.	Control [always]
Reserved	29	24	Reserved Area – Reserved for future expansion.	Reserved
vhpi	1	256	Virtual Highest Priority Pending Interrupt – Specifies the current highest priority pending interrupt for the virtual processor. See Table 11-119, “vhpi – Virtual Highest Priority Pending Interrupt” on page 2:495 for details.	Control [a_int]
Reserved	95	264	Reserved Area – Reserved for future expansion.	Reserved
vgr[16-31]	16	1024	Virtual General Registers – Represent the bank 1 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 1, the values in these entries are undefined.	Architectural State [a_bsw]
vbgr[16-31]	16	1152	Virtual Banked General Registers – Represent the bank 0 general registers 16-31 of the virtual processor. When the virtual processor is running and vpsr.bn is 0, the values in these entries are undefined.	Architectural State [a_bsw]
vnat	1	1280	Virtual General Register NaTs – Bits 0-15 represent the NaT values corresponding to vgr16-31, where the NaT bit for vgr16 is in bit 0. Bits 16-63 are don’t cares.	Architectural State [a_bsw]

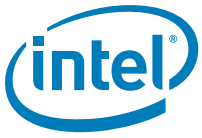
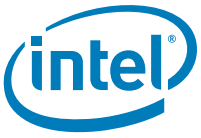
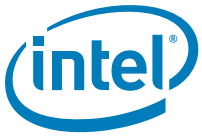


Table 11-16. Virtual Processor Descriptor (VPD) (Continued)

Name	Entries	Offset	Description	Class
vbnat	1	1288	Virtual Banked Register NaTs – Bits 16-31 represent the NaT values corresponding to vbgr16-31, where the NaT bit for vbgr16 is in bit 16. Bits 0-15 and 32-63 are don't cares.	Architectural State [a_bsw]
vcpuid[0-4]	5	1296	Virtual CPUID Registers – Represent cpuid registers 0-4 of the virtual processor. NOTE: vcpuid[0-1] and vcpuid[4]{63:32} must contain the same values as the corresponding values of the logical processor on which this virtual processor is running.	Architectural State [a_from_cpuid]
Reserved	11	1336	Reserved Area – Reserved for future expansion.	Reserved
vpsr	1	1424	Virtual Processor Status Register – Represents the Processor Status Register of the virtual processor.	Architectural State See Table 11-17 for details.
vpr	1	1432	Virtual Predicate Registers – Represents the Predicate Registers of the virtual processor. The bit positions in vpr correspond to predicate registers in the same manner as with the mov predicates instruction. The contents in this field are undefined except at virtualization intercept handoff. The VMM can not rely on the contents in this field to be preserved when the virtual processor is running.	Architectural State [always]
Reserved	76	1440	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state.	Reserved
vcr[0-127]	128	2048	Virtual Control Registers – Represent the control registers of the virtual processor. For the reserved control registers, the corresponding VPD entries are reserved.	Architectural State See Table 11-18 for details.
Reserved	128	3072	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
Reserved	3456	4096	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved
vmm_avail	128	31744	Available for VMM use. This area is ignored by the processor and PAL.	Ignored
Reserved	4096	32768	Reserved Area – Reserved for future expansion. This area may also be used by PAL to hold additional machine-specific processor state	Reserved





V2-L Section 11.7.4.1.3 and 11.7.4.3.5

11.7.4.1.3 Disable VMSW Instruction

The VMSW instruction disable is controlled by the d_vmsw bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, the `vmsw` instruction is disabled on the logical processor. Execution of the `vmsw` instruction, independent of the state of PSR.vm, results in a virtualization intercept.

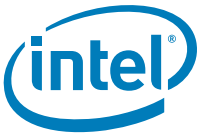
If this control is set to 0, the `vmsw` instruction can be executed by both the VMM and guest without virtualization intercepts, if PSR.it is 1 and the `vmsw` instruction is executed on a page with access rights of 7.

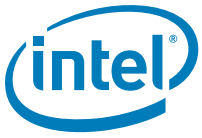
11.7.4.3.5 Disable MOV-to-PMD Virtualization

The MOV-to-PMD[1] virtualization disable is controlled by the d_to_pmd bit in the Virtualization Disable Control (*vdc*) field in the VPD. When this control is set to 1, writes to the performance monitor data registers (PMDs) are not virtualized, and code running with PSR.vm==1 can write these resources directly without any intercepts to the VMM.

If this control is set to 0, writes to the performance monitor data registers with PSR.vm==1 result in virtualization intercepts.

1. The MOV-from-PMD instruction is not virtualized. Hence there is no need to provide optimizations for the MOV-from-PMD instruction.





V2-M Section 11.10

11.10 PAL Procedures

PAL procedures may be called by higher-level firmware and software to obtain information about the identification, configuration, and capabilities of the processor implementation, or to perform implementation-dependent functions such as cache initialization. These procedures access processor implementation-dependent hardware to return information that characterizes and identifies the processor or implements a defined function on that particular processor.

PAL procedures are implemented by a combination of firmware code and hardware. The PAL procedures are defined to be relocatable from the firmware address space. Higher level firmware and software must perform this relocation during the reset flow. The PAL procedures may be called both before and after this relocation occurs, but performance will usually be better after the relocation. In order to ensure no problems occur due to the relocation of the PAL procedures, these procedures are written to be position independent. All references to constant data done by the procedures is done in an IP relative way.

PAL procedures are provided to return information or allow configuration of the following processor features:

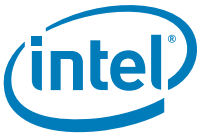
- Cache and memory features supported by the processor
- Processor identification, features, and configuration
- Machine Check Abort handling
- Power state information and management
- Processor self test
- Firmware utilities

PAL procedures are implemented as a single high level procedure, named PAL_PROC, whose first argument is an index which specifies which PAL procedure is being called. Indices are assigned depending on the nature of the PAL procedure being referenced, according to Table 11-48.

Table 11-48.PAL Procedure Index Assignment

Index	Description
0	Reserved; static register calling conventions
1 - 255	Architected procedures; static register calling conventions
256 - 511	Architected procedures; stacked register calling conventions
512 - 767	Implementation-specific procedures; static registers calling conventions
768 - 1023	Implementation-specific procedures; stacked register calling conventions
1024 +	Reserved; static register calling conventions

The assignment of indices for all architected procedures is controlled by this document. The assignment of indices for implementation-specific procedures is controlled by the specific processor for which the procedures are implemented. No implementation-specific procedure calls are required for the correct operation of a processor. No SAL or



operating system code should ever have to call an implementation-specific procedure call for normal activity. They are reserved for diagnostic and bring-up software and the results of such calls may be unpredictable.

Architected procedures may be designated as required or optional. If a procedure is designated as optional, a unique return code will be returned to indicate the procedure is not present in this PAL implementation. It is the caller's responsibility to check for this return code after calling any optional PAL procedure

In addition to the calling conventions described below, PAL procedure calls may be made in physical mode (PSR.it=0, PSR.rt=0, and PSR.dt=0) or virtual mode (PSR.it=1, PSR.rt=1, and PSR.dt=1). All PAL procedures may be called in physical mode. Only those procedures specified later in this chapter may be called in virtual mode. Reserved PAL procedure indices and indices for which no procedure is defined may be called either physically or virtually. (See Section 11.10.2.4, "Unimplemented Procedures".) PAL procedures written to support virtual mode, and the caller of PAL procedures written in virtual mode must obey the restrictions documented in this chapter, otherwise the results of such procedure calls may be unpredictable.

11.10.2.1.1 Static Registers Only

This calling convention is intended for boot time usage before main memory may be available or error recovery situations, where memory or the RSE may not be reliable. All parameters are passed in the lower 32 static general registers. The stacked registers will not be used within the procedure. No memory arguments may be passed as parameters to or from PAL procedures written using the static register calling convention. To avoid RSE activity, static register PAL procedures must be called with the *br.cond* instruction, not the br.call instruction. The caller must explicitly put the return point in BR0 (rp); PAL will branch to this address with a *br.cond* to return. Please refer to Table 11-54 for a detailed list of the general register usage for static registers only calling convention.

11.10.2.2.6 Branch Registers

The conventions for the branch registers follow the *Itanium Software Conventions and Runtime Architecture Guide*. For procedures that use the static register calling conventions, the caller must explicitly put the return point in BR0 (rp); PAL will branch to this address with a *br.cond* to return.

11.10.2.3 Return Buffers

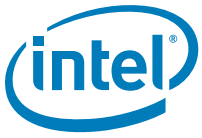
Any addresses passed to PAL procedures as buffers for return parameters must be 8-byte aligned. Unaligned addresses may cause undefined results.

11.10.2.4 Unimplemented Procedures

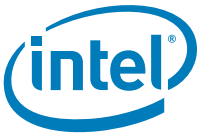
If the caller passes in a PAL procedure index value that is unimplemented (either an index for which no procedure is defined, or an index for which the defined procedure is optional but not implemented on this processor), PAL will return an Unimplemented Procedure (-1) status to the caller.

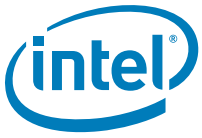
11.10.2.5 Invalid Arguments

The PAL procedure calling conventions specify rules that must be followed. These rules specify certain PSR values, they specify that reserved fields and arguments must be zero filled and specify that values not defined in a range and defined as reserved must not be used.



If the caller of a PAL procedure does not follow these rules, an invalid argument return value may be returned or undefined results may occur during the execution of the procedure.





V2-N PAL_BRAND_INFO

PAL_BRAND_INFO – Provides Processor Branding Information (274)

Purpose: Provides processor branding information.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Not dependent

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 11-57)
address	64-bit pointer to a 128-byte memory buffer 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
-9	Call requires PAL memory buffer

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 11-57 describes the values.

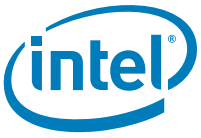
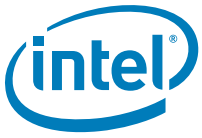


Table 11-57. Processor Brand Information Requested

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.

§



V2-O PAL_HALT_INFO

PAL_HALT_INFO – Get Halt State Information for Power Management (257)

Purpose: Returns information about the processor's power management capabilities.

Calling Conv: Stacked Registers

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

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

Returns:

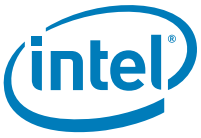
Return Value	Description
status	Return status of the PAL_HALT_INFO procedure.
Reserved	0
Reserved	0
Reserved	0

Status:

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

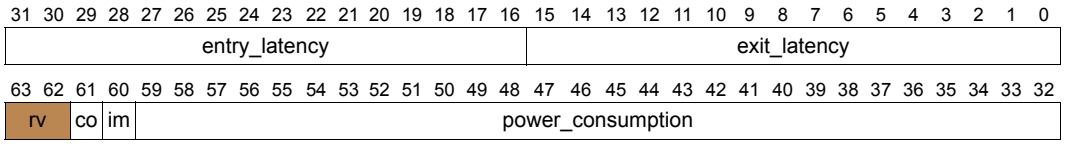
Description: The power information requested is returned in the data buffer referenced by *power_buffer*. Power information is returned about the 8 power states. The low power states are LIGHT_HALT, HALT, plus 6 other low power states. The LIGHT_HALT state is index 0 in the buffer, and the HALT state is index 1. All 8 low power states need not be implemented

The information returned is in the format of Figure 11-14. The information about the HALT states will be in ascending order of the index values



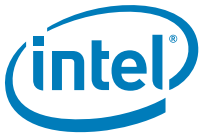
.

Figure 11-14. Layout of *power_buffer* Return Value



- *exit latency* – 16-bit unsigned integer denoting the minimum number of processor cycles to transition to the NORMAL state.
- *entry_latency* – 16-bit unsigned integer denoting the minimum number of processor cycles to transition from the NORMAL state.
- *power_consumption* – 28-bit unsigned integer denoting the typical power consumption of the state, measured in milliwatts.
- *im* – 1-bit field denoting whether this low power state is implemented or not. A value of 1 indicates that the low power state is implemented, a value of 0 indicates that it is not implemented. If this value is 0 then all other fields are invalid.
- *co* – 1-bit field denoting if the low power state maintains cache and TLB coherency. A value of 1 indicates that the low power state keeps the caches and TLBs coherent, a value of 0 indicates that it does not.

The latency numbers given are the minimum number of processor cycles that will be required to transition the states. The maximum or average cannot be determined by PAL due to its dependency on outstanding bus transactions.
For more information on power management, please refer to Section 11.6, “Power Management” on page 2:316.



V2-P PAL_MC_DYNAMIC_STATE

PAL_MC_DYNAMIC_STATE – Returns Dynamic Processor State (24)

Purpose: Returns the Machine Check Dynamic Processor State.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_DYNAMIC_STATE within the list of PAL procedures.
info_type	Unsigned 64-bit value indicating the type of information to return
dy_buffer	64-bit pointer to a memory buffer aligned on an 8-byte boundary.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_MC_DYNAMIC_STATE procedure.
max_size	Maximum size (in bytes) of the data that can be returned by this procedure for this processor family. This value is always a multiple of 8 bytes.
Reserved	0
Reserved	0

Status:

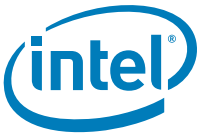
Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error

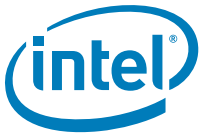
Description: The *info_type* input argument designates the type of information the procedure will return. When *info_type* is 0, the procedure returns the maximum size (in bytes) of processor dynamic state that can be returned for this processor family in the *max_size* return value. The *max_size* return value will always be a multiple of 8 bytes.

When *info_type* is 1, the procedure will copy processor dynamic state into memory pointed to by the input argument *dy_buffer*. This copy will occur using the addressing attributes used to make the procedure call (physical or virtual) and the caller needs to ensure the *dy_buffer* input pointer matches this addressing attribute.

The amount of data returned can vary depending on the state of the machine at the time the procedure is called, and may not always return the maximum size for every call. The amount of data returned is provided in the processor state parameter field *dsize*. The *dsize* parameter will always be a multiple of 8 bytes. Please see Table 11-7 for more information on the processor state parameter. The caller of the procedure needs to ensure that the buffer is large enough to handle the *max_size* that is returned by this procedure.

The contents of the processor dynamic state is implementation dependent. Portions of this information may be cleared by the PAL_MC_CLEAR_LOG procedure. This procedure should be invoked before PAL_MC_CLEAR_LOG to ensure all the data is captured.





V2-Q PAL_MC_ERROR_INFO / PAL_MC_ERROR_INJECT

PAL_MC_ERROR_INFO – Get Processor Error Information (25)

Purpose: Returns the Processor Machine Check Information

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_MC_ERROR_INFO within the list of PAL procedures.
info_index	Unsigned 64-bit integer identifying the error information that is being requested. (See Table 11-86).
level_index	8-byte formatted value identifying the structure to return error information on.(See Figure 11-19).
err_type_index	Unsigned 64-bit integer denoting the type of error information that is being requested for the structure identified in *level_index*.

Returns:

Return Value	Description
status	Return status of the PAL_MC_ERROR_INFO procedure.
error_info	Error information returned. The format of this value is dependant on the input values passed.
inc_err_type	If this value is zero, all the error information specified by *err_type_index* has been returned. If this value is one, more structure-specific error information is available and the caller needs to make this procedure call again with *level_index* unchanged and *err_type_index,* incremented.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-6	Argument was valid, but no error information was available

Description: This procedure returns error information for machine checks as specified by *info_index*, *level_index* and *err_type_index*. Higher level software is informed that additional machine check information is available when the processor state parameter *mi* bit is set to one. See Table 11-7, “Processor State Parameter Fields,” on page 2:299 for more information on the processor state parameter and the *mi* bit description.

The *info_index* argument specifies which error information is being requested. See Table 11-86 for the definition of the *info_index* values.

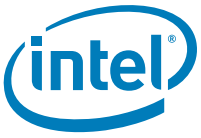


Table 11-86. *info_index* Values

info_index	Error Information Type	Description
0	Processor Error Map	This *info_index* value will return the processor error map. This return value specifies the processor core identification, the processor thread identification, and a bit-map indicating which structure(s) of the processor generated the machine check. This bit-map has the same layout as the *level_index*. A one in the structure bit-map indicates that there is error information available for the structure. The layout of the *level_index* is described in Figure 11-19, “level_index Layout” on page 73.
1	Processor State Parameter	This *info_index* value will return the same processor state parameter that is passed at the PALE_CHECK exit state for a machine check event (provided a valid min-state save area has been registered) or will construct a processor state parameter for a corrected machine check events. This parameter describes the severity of the error and the validity of the processor state when the machine check or CMCI occurred. This procedure will not return a valid PSP for INIT events. The Processor State Parameter is described in Figure 11-11, “Processor State Parameter,” on page 2:299.
2	Structure-specific Error Information	This *info_index* value will return error information specific to a processor structure. The structure is specified by the caller using the *level_index* and *err_type_index* input parameters. The value returned in *error_info* is specific to the structure and type of information requested.

All other values of *info_index* are reserved. When *info_index* is equal to 0 or 1, the *level_index* and *err_type_index* input values are ignored. When *info_index* is equal to 2, the *level_index* and *err_type_index* define the format of the *error_info* return value.

The caller is expected to first make this procedure call with *info_index* equal to zero to obtain the processor error map. This error map informs the caller about the processor core identification, the processor thread identification and indicates which structure(s) caused the machine check. If more than one structure generated a machine check, multiple structure bits will be set. The caller then uses this information to make sub-sequent calls to this procedure for each structure identified in the processor error map to obtain detailed error information.

The *level_index* input argument specifies which processor core, processor thread and structure for which information is being requested. See Table 11-87 on page 73 for the definition of the *level_index* fields. This procedure call can only return information about one processor structure at a time. The caller is responsible for ensuring that only one structure bit in the l*evel_index* input argument is set at a time when retrieving information, otherwise the call will return that an invalid argument was passed.

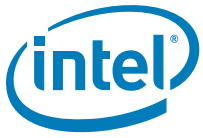


Figure 11-19. *level_index* Layout

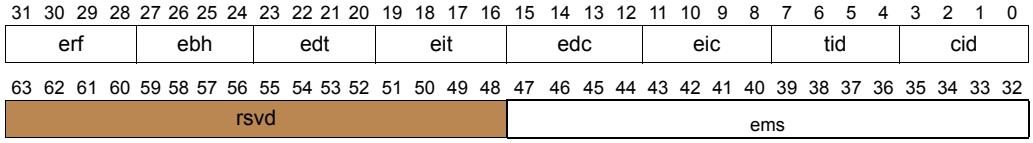


Table 11-87. *level_index* Fields

Field	Bits	Description
cid	3:0	Processor core ID (default is 0 for processors with a single core)
tid	7:4	Logical thread ID (default is 0 for processors that execute a single thread)
eic	11:8	Error information is available for 1st, 2nd, 3rd, and 4th level instruction caches
edc	15:12	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified caches
eit	19:16	Error information is available for 1st, 2nd, 3rd, and 4th level instruction TLB
edt	23:20	Error information is available for 1st, 2nd, 3rd, and 4th level data/unified TLB
ebh	27:24	Error information is available for the 1st, 2nd, 3rd, and 4th level processor bus hierarchy
erf	31:28	Error information is available on register file structures
ems	47:32	Error information is available on micro-architectural structures
rsvd	63:48	Reserved

The convention for levels and hierarchy in the *level_index* field is such that the least significant bit in the error information bit-fields represent the lowest level of the structures hierarchy. For example bit 8 if the *eic* field represents the first level instruction cache.

The *erf* field is 4-bits wide to allow reporting of 4 concurrent register related machine checks at one time. One bit would be set for each error. The *ems* field is 16-bits wide to allow reporting of 16-concurrent micro-architectural structures at one time. There is no significance in the order of these bits. If only one register file related error occurred, it could be reported in any one of the 4-bits.

The *err_type_index* specifies the type of information will be returned in *error_info* for a particular structure. See Table 11-88 for the values of *err_type_index*

Table 11-88. *err_type_index* Values

err_type_index value mod 8	Return Value	Description
0	Structure-specific error information specified by *level_index*	The information returned in *error_info* is dependant on the structure specified in *level_index*. See Table 11-89 for the error_info return formats.
1	Target address	The target address is a 64-bit integer containing the physical address where the data was to be delivered or obtained. The target address also can return the incoming address for external snoops and TLB shoot-downs that generated a machine check. The structure-specific error information informs the caller if there is a valid target address to be returned for the requested structure.
2	Requester identifier	The requester identifier is a 64-bit integer that specifies the bus agent that generated the transaction responsible for generating the machine check. The structure-specific error information informs the caller if there is a valid requester identifier.

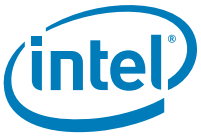


Table 11-88. *err_type_index* Values (Continued)

err_type_index value mod 8	Return Value	Description
3	Responder identifier	The responder identifier is a 64-bit integer that specifies the bus agent that responded to a transaction that was responsible for generating the machine check. The structure-specific error information informs the caller if there is a valid responder identifier.
4	Precise instruction pointer	The precise instruction pointer is a 64-bit virtual address that points to the bundle that contained the instruction responsible for the machine check. The structure-specific error information informs the caller if there is a valid precise instruction pointer.
5-7	Reserved	Reserved

See Table 11-89 for the format of *error_info* when structure-specific information is requested.

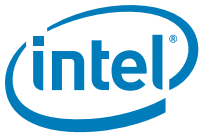
Table 11-89. *error_info* Return Format when *info_index* = 2 and *err_type_index* = 0

level_index Field Input	*error_info* Return Format
eic	cache_check return format
edc	cache_check return format
eit	tlb_check return format
edt	tlb_check return format
ebh	bus_check return format
erf	reg_file_check return format
ems	uarch_check return format

The structure specified by the *level_index* may have the ability to log distinct multiple errors. This can occur if the structure is accessed at the same time by more than one instruction and the processor can log machine check information for each access. To inform the caller of this occurrence, this procedure will return a value of one in the *inc_err_type* return value.

It is important to note, that when the caller sees that the *inc_err_type* return value is one, it should make a sub-sequent call with the *err_type_index* value incremented by 8. If the structure-specific error information returns that there is a valid target address, requester identifier, responder identifier or precise instruction pointer these can be returned as well by incrementing the *err_type_index* value in the same manner. Refer to the following example for more information.

For example, to gather information on the first error of a structure that can log multiple errors, *err_type_index* would be called with the value of 0 first. The caller examines the information returned in *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise instruction pointer available for logging. If there is, it makes sub-sequent calls with *err_type_index* equal to 1, 2, 3 and/or 4 depending on which valid bits are set. Additionally if the *inc_err_type* return value was set to one, the caller knows that this structure logged multiple errors. To get the second error of the structure it sets the *err_type_index* = 8 and the structure-specific information is returned in *error_info*. The caller examines this *error_info* to know if there is a valid target address, requester identifier, responder identifier, or precise



instruction pointer available for logging on the second error. If there is, it makes subsequent calls with *err_type_index* equal to 9, 10, 11, and/or 12 depending on which valid bits are set. The caller continues incrementing the *err_type_index* value in this fashion until the *inc_err_type* return value is zero.

As shown in Table 11-89, the information returned in *error_info* varies based on which structure information is being requested on. The next sections describe the *error_info* return format for the different structures.

Cache_Check Return Format: The cache check return format is returned in *error_info* when the user requests information on any instruction or data/unified caches in the *level_index* input argument. The cache_check return format must be used to report errors in cacheable transactions. These errors may also be reported using the bus_check return format if the bus structures can detect these errors. The cache_check return format is a bit-field that is described in Figure 11-20 and Table 11-90.

Figure 11-20. cache_check Layout

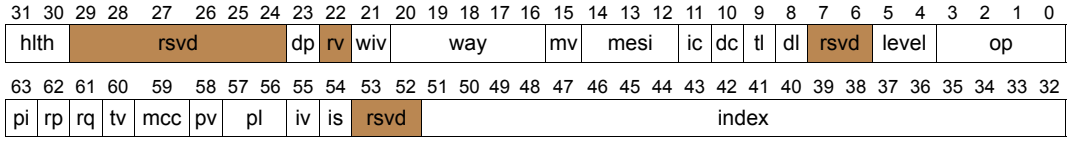


Table 11-90. cache_check Fields

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	An uncorrectable (typically multiple-bit) error was detected and data was poisoned for the corresponding cache line, without any corrupted data being consumed (i.e., no corrupted data has been copied to processor registers).

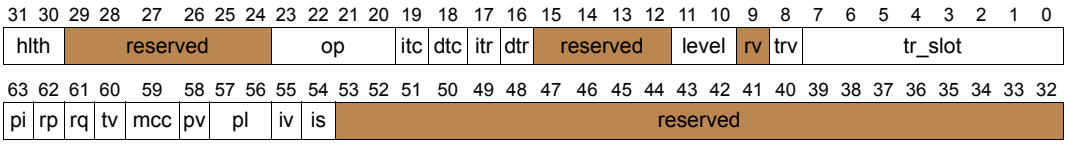
Table 11-90. cache_check Fields (Continued)

Field	Bits	Description
rsvd	29:24	Reserved
hlth	31:30	Health indicator. This field will report if the cache type and level reporting this error supports hardware status tracking and the current status of this cache. 00 – No hardware status tracking is provided for the cache type and level reporting this event. 01 – Status tracking is provided for this cache type and level and the current status is normal status.[a] 10 – Status tracking is provided for the cache type and level and the current status is cautionary.<Superscript>a When a cache reports a cautionary status the "hardware damage" bit of the PSP (see Figure 11-11, “Processor State Parameter,” on page 2:299) will be set as well. 11 – 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.

a. Hardware is tracking the operating status of the structure type and level reporting the error. The hardware reports a "normal" status when the number of entries within a structure reporting repeated corrections is at or below a pre-defined threshold. A "cautionary" status is reported when the number of affected entries exceeds a pre-defined threshold.

TLB_Check Return Format: The tlb_check return format is returned in *error_info* when the user requests information on any instruction or data/unified TLB in the *level_index* input argument. The tlb_check return format is a bit-field that is described in Figure 11-21 and Table 11-91.

Figure 11-21. tlb_check Layout



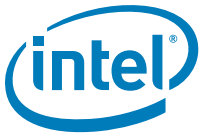


Table 11-91. tlb_check Fields

Field	Bits	Description
tr_slot	7:0	Slot number of the translation register where the failure occurred.
trv	8	The *tr_slot* field in the TLB_check parameter is valid.
rv	9	Reserved
level	11:10	The level of the TLB where the error occurred. A value of 0 indicates the first level of TLB
reserved	15:12	Reserved
dtr	16	Error occurred in the data translation registers
itr	17	Error occurred in the instruction translation registers
dtc	18	Error occurred in data translation cache
itc	19	Error occurred in the instruction translation cache
op	23:20	Type of cache operation that caused the machine check: 0 – unknown 1 – TLB access due to load instruction 2 – TLB access due to store instruction 3 – TLB access due to instruction fetch or instruction prefetch 4 – TLB access due to data prefetch (both hardware and software) 5 – TLB shoot down access 6 – TLB probe instruction (probe, tpa) 7 – move in (VHPT fill) 8 – purge (insert operation that purges entries or a TLB purge instruction) All other values are reserved.
reserved	29:24	Reserved
hlth	31:30	Health indicator. This field will report if the tlb type and level reporting this error supports hardware status tracking and the current status of this tlb. 00 – No hardware status tracking is provided for the tlb type and level reporting this event. 01 – Status tracking is provided for this tlb type and level and the current status is normal.[a] 10 – Status tracking is provided for the tlb type and level and the current status is cautionary.<Superscript>a When a tlb reports a cautionary status the "hardware damage" bit of the PSP (see Figure 11-11, “Processor State Parameter,” on page 2:299) will be set as well. 11 – Reserved
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 TLB_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 TLB_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.

a. Hardware is tracking the operating status of the structure type and level reporting the error. The hardware reports a "normal" status when the number of entries within a structure reporting repeated corrections is at or below a pre-defined threshold. A "cautionary" status is reported when the number of affected entries exceeds a pre-defined threshold.

Bus_Check Return Format: The bus_check return format is returned in *error_info* when the user requests information on any level of hierarchy of the processor bus structures as specified in the *level_index* input argument. The bus_check return format must be used to report errors in uncacheable transactions. These errors must not be reported using the cache_check return format. The bus_check return format is a bit-field that is described in Figure 11-22 and Table 11-92.

Figure 11-22. bus_check Layout

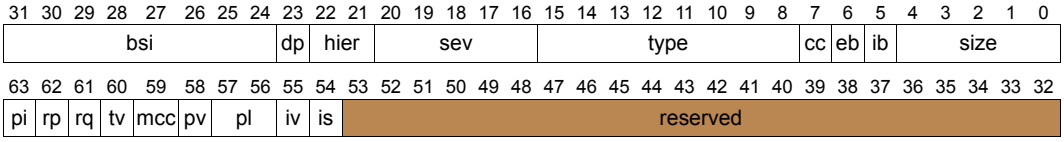


Table 11-92. bus_check Fields

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.

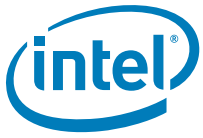


Table 11-92. bus_check Fields (Continued)

Field	Bits	Description
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.
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.

Reg_File_Check Return Format: The reg_file_check return format is returned in *error_info* when the user requests information on any of the registers as specified in the *level_index* input argument. The reg_file_check return format is a bit-field that is described in Figure 11-23 and Table 11-93. When the reg_file_check return format is returned, the target address, the requester identifier and the responder identifier will always be invalid.

Figure 11-23. reg_file_check Layout

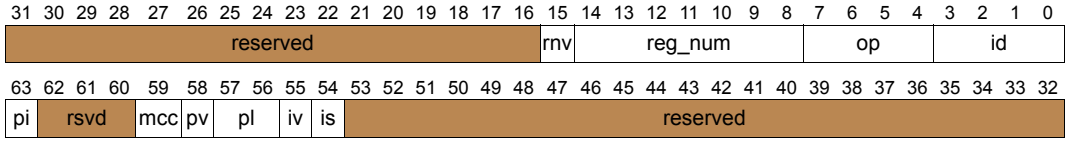


Table 11-93. reg_file_check Fields

Field	Bits	Description
id	3:0	Register file identifier: 0 – unknown/unclassified 1 – General register (bank1) 2 – General register (bank 0) 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
op	7:4	Identifies the operation that caused the machine check 0 – unknown 1 – read 2 – write All other values are processor specific
reg_num	14:8	Identifies the register number that was responsible for generating the machine check

Table 11-93. reg_file_check Fields

Field	Bits	Description
rnv	15	Specifies if the *reg_num* field is valid
reserved	53:16	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 reg_file_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 reg_file_check parameter is valid.
mcc	59	Machine check corrected: This bit is set to one to indicate that the machine check has been corrected.
reserved	62:60	Reserved
pi	63	Precise instruction pointer. This bit is set to one to indicate that a valid precise instruction pointer has been logged.

Uarch_Check Return Format: The uarch_check return format is returned in *error_info* when the user requests information on any of the micro-architectural structures as specified in the *level_index* input argument. The uarch_check return format is a bit-field that is described in Figure 11-24 and Table 11-94.

Figure 11-24. uarch_check Layout

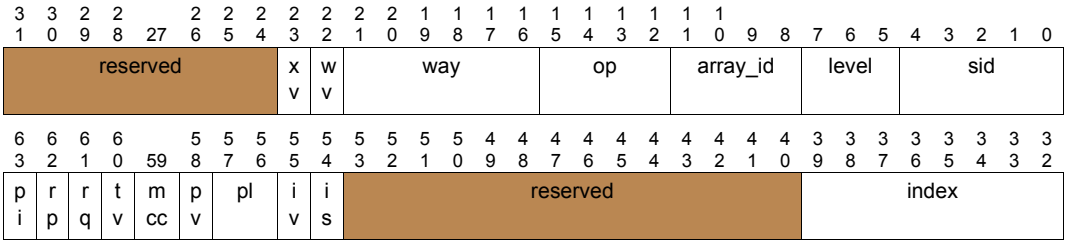


Table 11-94. uarch_check Fields

Field	Bits	Description
sid	4:0	Structure identification. These bits identify the micro-architectural structure where the error occurred. The definition of these bits are implementation specific.
level	7:5	Level of the micro-architectural structure where the error was generated. A value of 0 indicates the first level.
array_id	11:8	Identification of the array in the micro architectural structure where the error was generated. 0 – unknown/unclassified All other values are implementation specific
op	15:12	Type of operation that caused the error 0 – unknown 1 – read or load 2 – write or store All other values are implementation specific
way	21:16	Way of the micro-architectural structure where the error was located.

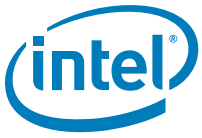
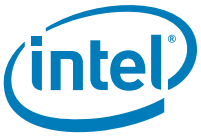


Table 11-94. uarch_check Fields

Field	Bits	Description
wv	22	The *way* field in the uarch_check parameter is valid.
xv	23	The *index* field in the uarch_check parameter is valid.
reserved	31:24	Reserved
index	39:32	Index or set of the micro-architectural structure where the error was located.
reserved	53:40	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.
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.



PAL_MC_ERROR_INJECT – Inject Processor Error (276)

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

Buffer: Dependent

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 pointer to a memory buffer providing additional parameters for the requested error. The address of the 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

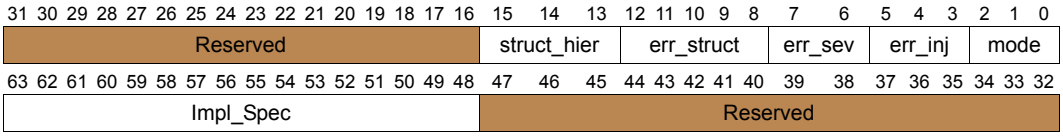
Status:

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.
-9	Call requires PAL memory buffer

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 11-25). 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 11-25. *err_type_info*



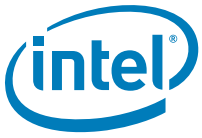
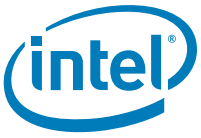


Table 11-95. *err_type_info*

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 2:349. 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	47:16	Reserved
Impl_Spec	63:48	Processor specific error injection capabilities. Please refer to processor specific documentation for additional details.

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_inj*, *err_sev*, *err_struct*, and *struct_hier* 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 (selectable through the *err_inj* field in *err_type_info*). 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.

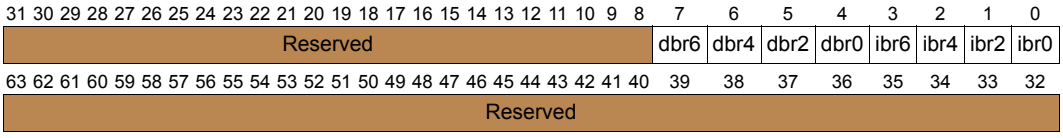
The procedure also provides the ability to set an error injection trigger. In this case, the error injection is delayed until the operation specified by the trigger is encountered and the executing context has the specified privilege level. In the absence of a trigger, the error injection is performed at the time of procedure execution. If an error injection trigger is specified, the mode field in *err_type_info* determines whether the error is injected, or injected and consumed when the trigger operation is encountered. There can be only one outstanding trigger programmed at a time. Subsequent procedure calls that use the trigger functionality will overwrite the previous trigger parameters. Once a trigger is programmed it remains active until either the trigger operation is encountered or software cancels the outstanding trigger via this call. Software can cancel outstanding triggers by specifying *Cancel outstanding trigger* via the *mode* bit in *err_type_info*. The *resources* value returned is all zeroes, indicating that the procedure is no longer using any architectural resources (specified in *resources*) for triggering purposes. 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 11-26).

In order for triggering to work when PAL is using the IBR or DBR registers, certain PSR bits are required to be set. Software needs to ensure that the PSR.db and the PSR.ic bits are set to one when executing the code that it is targeting with the trigger. If either one of these bits are not set, then triggers will not work as defined.

Procedure operation is undefined if software overwrites or modifies the IBR/DBR resources that PAL indicates it is using for a trigger. The IBR/DBR resources that PAL is not using are available for software to program for their own use.

Figure 11-26. *resources* Return Value



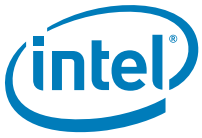


Table 11-96. *resources* Return Value

Field	Bits	Description
ibr0	0	When 1, indicates that IBR0,1 are being used by the procedure for trigger functionality.
ibr2	1	When 1, indicates that IBR2,3 are being used by the procedure for trigger functionality.
ibr4	2	When 1, indicates that IBR4,5 are being used by the procedure for trigger functionality.
ibr6	3	When 1, indicates that IBR6,7 are being used by the procedure for trigger functionality.
dbr0	4	When 1, indicates that DBR0,1 are being used by the procedure for trigger functionality.
dbr2	5	When 1, indicates that DBR2,3 are being used by the procedure for trigger functionality.
dbr4	6	When 1, indicates that DBR4,5 are being used by the procedure for trigger functionality.
dbr6	7	When 1, indicates that DBR6,7 are 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 (cache line, 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*=1 (virtual address provided), then PAL will attempt to locate the cache line as indicated by the input virtual address. If the corresponding cache line cannot be found (the cache line could have been evicted from the cache in the time interval between the procedure call and the search process, or the cache line 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 11-27. *err_struct_info* – Cache

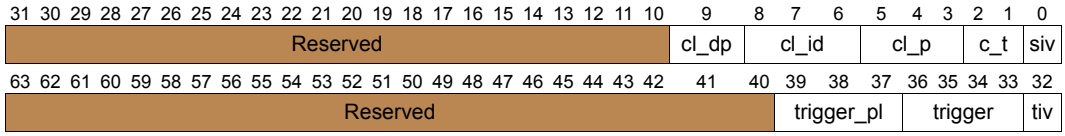


Table 11-97. *err_struct_info* – Cache

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 cache line where the error should be injected: 0 – Reserved 1 – Tag 2 – Data 3 – mesi All other values are reserved.

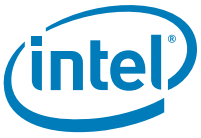


Table 11-97. *err_struct_info* – Cache (Continued)

Field	Bits	Description
cl_id	8:6	Indicates which mechanism is used to identify the cache line 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 cache line 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 cache line for error injection. 3 – *way* and *index* fields provided in *err_data_buffer* should be used to identify the cache line for error injection. All other values are reserved.
cl_dp	9	When 1, indicates that a multiple bit, non-correctable error should be injected in the cache line 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 cache line is cast out (implicit or explicit write-back of the cache line). 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 154. 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 154. 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 11-28. *capabilities* vector for cache

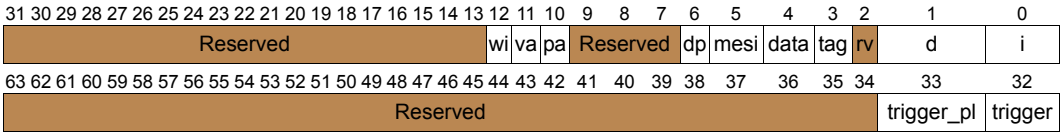


Table 11-98. *capabilities* vector for cache

Field	Bits	Description
i	0	Error injection for instruction caches is supported
d	1	Error injection for data caches is supported
rv	2	Reserved

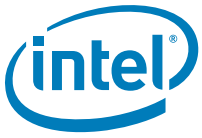


Table 11-98. *capabilities* vector for cache (Continued)

Field	Bits	Description
tag	3	Error injection in *tag* portion of cache line is supported
data	4	Error injection in *data* portion of cache line is supported
mesi	5	Error injection in *mesi* portion of cache line 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
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 needs to be specified for *cache* only if *siv* is 1 or *tiv* is 1, in *err_struct_info*.

Figure 11-29. Buffer pointed to by *err_data_buffer* – Cache

Table 11-99. 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 153.
inj_addr	127:64	64-bit virtual or physical address used to identify the cache line 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 cache line 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 cache line 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

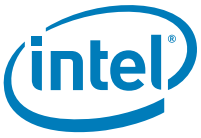


Figure 11-30. *err_struct_info* – TLB

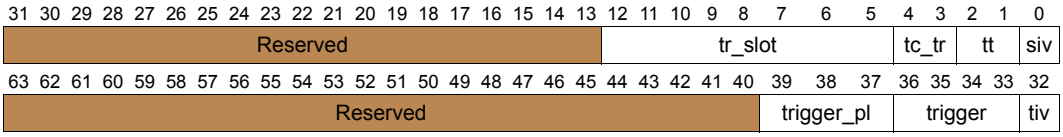


Table 11-100. *err_struct_info* – TLB

Field	Bits	Description
siv	0	When 1, indicates that the structure information fields (*tt, 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 154. 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 154.. 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

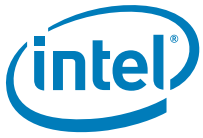


Figure 11-31. *capabilities* vector for TLB

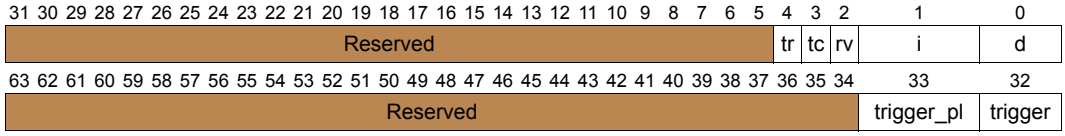


Table 11-101. *capabilities* vector for TLB

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 11-32. Buffer pointed to by *err_data_buffer* – TLB

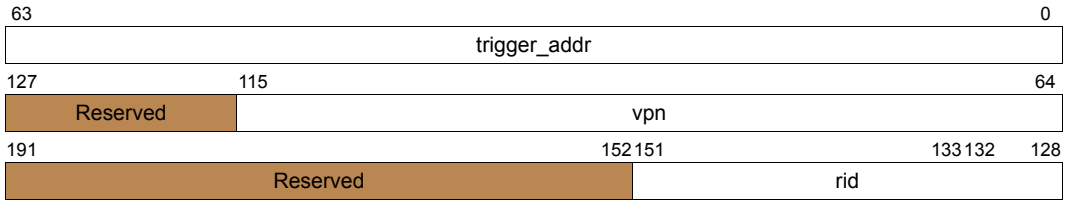
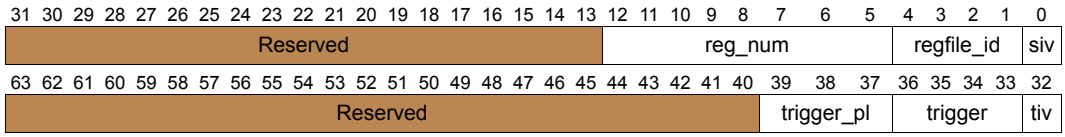


Table 11-102. Buffer pointed to by *err_data_buffer* – TLB

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 153.
vpn	115:64	Indicates the Virtual page number. This field 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 1. *rid* is used in combination with *vpn* to identify the TC entry for error injection.
Reserved	191:152	Reserved

Figure 11-33. *err_struct_info* – Register File



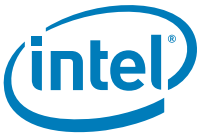


Table 11-103. *err_struct_info* – Register File

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	12:5	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	31:13	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 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 154. 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 154.. 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

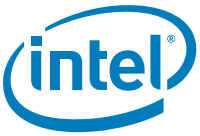


Figure 11-34. *capabilities* Vector for Register File

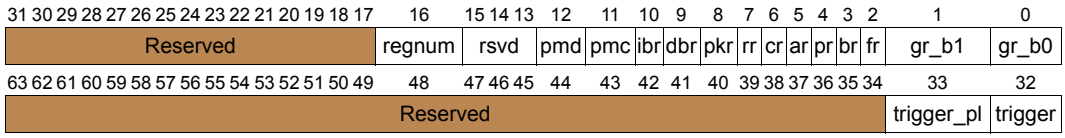


Table 11-104. *capabilities* Vector for Register File

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 11-35. Buffer pointed to by *err_data_buffer* – Register File

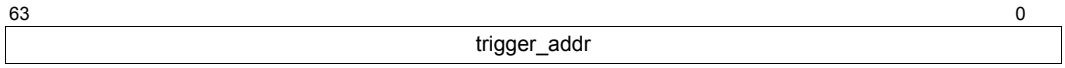


Table 11-105. Buffer pointed to by *err_data_buffer* – Register File

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 153.

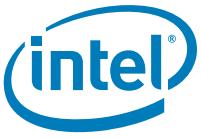


Figure 11-36. *err_struct_info* – Bus/Processor Interconnect

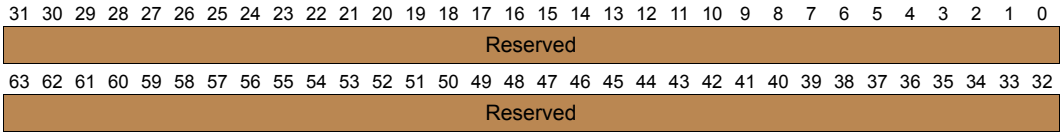


Table 11-106. *err_struct_info* – Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

Figure 11-37. *capabilities* vector for Bus/Processor Interconnect

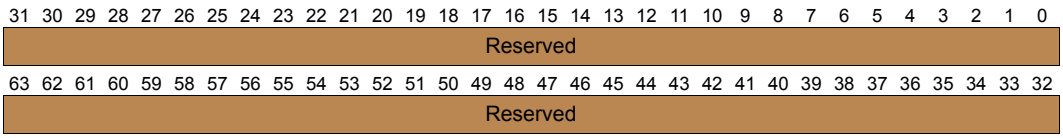


Table 11-107. *capabilities* vector for Bus/Processor Interconnect

Field	Bits	Description
Reserved	63:0	Reserved

err_data_buffer does not need to be specified for *bus/system interconnect*.

V2-R PAL_PERF_MON_INFO

PAL_PERF_MON_INFO – Get Processor Performance Monitor Information (15)

Purpose: Returns Performance Monitor information about what can be counted and how to configure the monitors to count the desired events.

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PERF_MON_INFO within the list of PAL procedures.
pm_buffer	64-bit pointer to a 128-byte memory buffer aligned on an 8-byte boundary.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PERF_MON_INFO procedure.
pm_info	Information about the performance monitors implemented.
Reserved	0
Reserved	0

Status:

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

Description: PAL_PERF_MON_INFO is called to determine the number of performance monitors and the events which can be counted on the performance monitors. For more information on performance monitoring, see Section 7.2, “Performance Monitoring” on page 2:155. *pm_info* is a formatted 64-bit return register, as shown in Figure 11-40.

.

Figure 11-40. Layout of *pm_info* Return Value

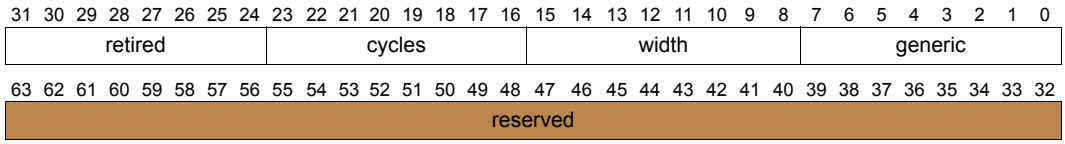


Table 11-105. *pm_info* Fields

Field	Description
generic	Unsigned 8-bit number defining the number of generic PMC/PMD pairs.
width	Unsigned 8-bit number in the range 0:60 defining the number of implemented counter bits.
cycles	Unsigned 8-bit number defining the event type for counting processor cycles.
retired	Unsigned 8-bit number defining the event type for retired instruction bundles.

The *pm_buffer* argument points to a 128-byte memory area where mask information is returned. The layout of *pm_buffer* is shown in Table 11-106.

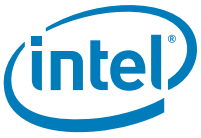


Table 11-106. *pm_buffer* Layout

Offset	Description
0x0	256-bit mask defining which PMC registers are implemented.
0x20	256-bit mask defining which PMD registers are implemented.
0x40	256-bit mask defining which registers can count cycles.
0x60	256-bit mask defining which registers can count retired bundles.

V2-S PAL_VP_INFO

PAL_VP_INFO – PAL Virtual Processor Information (50)

Purpose: Returns information about virtual processor features.

Calling Conv: Static

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VP_INFO within the list of PAL procedures
feature_set	Feature set information is being requested for.
vp_buffer	64-bit pointer to an 8-byte aligned memory buffer (if used).
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_INFO procedure
vp_info	Information about the virtual processor.
vmm_id	Unique identifier for the VMM.
Reserved	0

Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-8	Specified feature_set is not implemented

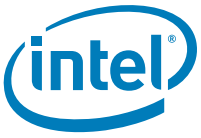
Description: The PAL_VP_INFO procedure call is used to describe virtual processor features.

The *feature_set* input argument for PAL_VP_INFO describes which virtual-processor *feature_set* information is being requested, and is composed of two fields as shown:

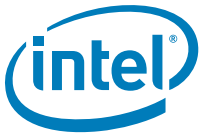
A *vmm_id* of 0 indicates architected feature sets, while others are implementation-specific feature sets. Implementation-specific feature sets are described in VMM-specific documentation.

This procedure will return a -8 if an unsupported *feature_set* argument is passed as an input. The return status is used by the caller to know which feature sets are currently supported on a particular VMM. This procedure always returns unimplemented (-1) when called on physical processors.

For each valid *feature_set*, this procedure returns information about the virtual processor in *vp_info*. Additional information may be returned in the memory buffer pointed to by *vp_buffer*, as needed. Details, for a given implementation-specific *feature_set*, of whether information is returned in the buffer, the size of the buffer, and the representation of this information in the buffer and in *vp_info* are described in VMM-specific documentation.



Architected *feature_set* 0 (*vmm_id* 0, *index* 0) is defined and required to be implemented (if this procedure is implemented), but there are no architected features defined in it yet, and so all bits in *vp_info* are reserved for architected *feature_set* 0. Other architected feature sets (*vmm_id* 0, *index*>0) are undefined, and return -8 (Specified *feature_set* is not implemented). Software can call PAL_VP_INFO with a *feature_set* argument of 0 to get the *vmm_id*, although *vmm_id* is also returned for any other implemented feature sets as well. For *feature_set* 0, the *vp_buffer* argument is ignored.



V2-T PAL_VP_REGISTER

PAL_VP_REGISTER – PAL Register Virtual Processor (269)

Purpose: Register a different host IVT and/or a different optional virtualization intercept handler for the virtual processor specified by *vpd*.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_REGISTER within the list of PAL procedures
vpd	64-bit host-virtual pointer to the Virtual Processor Descriptor (VPD)
host_iva	64-bit host-virtual pointer to the host IVT for the virtual processor
opt_handler	64-bit non-zero host-virtual pointer to an optional handler for virtualization intercepts. See Section 11.7.3, “PAL Intercepts in Virtual Environment” on page 2:334 for details.

Returns:

Return Value	Description
status	Return status of the PAL_VP_REGISTER procedure
Reserved	0
Reserved	0
Reserved	0

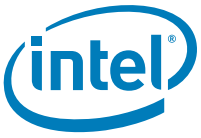
Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_VP_REGISTER registers a different host IVT and/or a different optional virtualization intercept handler specific to the virtual processor specified by *vpd*. On creation of a virtual processor by PAL_VP_CREATE, the VMM specifies a host IVT specific to the virtual processor. This procedure allows the VMM to specify a host IVT different from the one specified during PAL_VP_CREATE.

The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped with either a DTR or DTC. See Section 11.10.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:358 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. The *virt_env_vaddr* parameter in the VPD must be setup with the host virtual address of the PAL virtual environment buffer before calling this procedure.

The *host_iva* parameter specifies the host IVT to handle IVA-based interruptions when this virtual processor is running. The VMM can use the same or different *host_iva* for each virtual processor. The *opt_handler* specifies an optional virtualization intercept handler. If a non-zero value is specified, all virtualization intercepts are delivered to this handler. If a zero value is specified, all virtualization intercepts are delivered to the Virtualization vector in the host IVT. Upon completion of this procedure, the VMM must not relocate the IVT specified by the *host_iva* parameter and/or the virtualization intercept handler specified by the *opt_handler* parameter. The VMM can call this



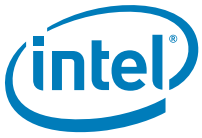
procedure again in case it wishes to associate a different host IVT and/or virtualization intercept handler with the virtual processor.

PAL_VP_REGISTER returns invalid argument on unsupported virtualization optimization combinations in *vpd*. See Section 11.7.4.4, “Virtualization Optimization Combinations” on page 2:348 for details.

This procedure can be used by the VMM to:

- Relocate the host IVT associated with the virtual processor.
- Specify a different optional virtualization intercept handler for the virtual processor.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:447 for details.



V2-U PAL_PLATFORM_ADDR and Table 11-50

PAL_PLATFORM_ADDR – Set Processor Interrupt Block Address and I/O Port Space Address (16)

Purpose: Specifies the physical address of the processor Interrupt Block and I/O Port Space.

Calling Conv: Static Registers Only

Mode: Physical or Virtual

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PLATFORM_ADDR within the list of PAL procedures.
type	Unsigned 64-bit integer specifying the type of block. 0 indicates that the processor interrupt block pointer should be initialized. 1 indicates that the processor I/O block pointer should be initialized.
address	Unsigned 64-bit integer specifying the address to which the processor I/O block or interrupt block shall be set. The address must specify an implemented physical address on the processor model, bit 63 is ignored.
Reserved	0

Returns:

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

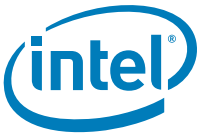
Status:

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

Description: PAL_PLATFORM_ADDR specifies the physical address that the processor shall interpret as accesses to the SAPIC memory or the I/O Port space areas.

The default value for the Interrupt block pointer is 0x00000000 FEE00000. If an alternate address is selected by this call, it must be aligned on a 2 MB boundary, else the procedure will return an error status (-2). Additionally, the address specified must not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary. Some processor implementations may not support relocation of the Interrupt block pointer, and an invalid argument status (-2) will be returned. In this case, the default address space will be used.

The default value for the I/O block pointer is to the beginning of the 64 MB block at the highest physical address supported by the processor. Therefore, its physical address is implementation dependent. If an alternate address is selected by this call, it must be aligned on a 64MB boundary, else the procedure will return an invalid argument status (-2). Additionally, the address specified must not overlay any firmware addresses in the 16 MB region immediately below the 4GB physical address boundary. Some processor



implementations may not support relocation of the I/O block pointer, and an invalid argument status (-2) will be returned. In this case, the default address space will be used.

The Interrupt and I/O Block pointers should be initialized by firmware before any Inter-Processor Interrupt messages or I/O Port accesses. Otherwise the default block pointer values will be used.

If a processor implementation supports relocation of neither the interrupt nor the I/O block pointer, this procedure will not be implemented, and an unimplemented procedure status (-1) will be returned. In this case, the default address spaces will be used.

Table 11-50.PAL Processor Identification, Features, and Configuration Procedures

Procedure	Idx	Class	Conv.	Mode	Buffer	Description
PAL_BRAND_INFO	274	Opt.	Stacked	Both	No	Provides processor branding information.
PAL_BUS_GET_FEATURES	9	Req.	Static	Phys.	No	Return configurable processor bus interface features and their current settings.
PAL_BUS_SET_FEATURES[a]	10	Req.	Static	Phys.	No	Enable or disable configurable features in processor bus interface.
PAL_DEBUG_INFO	11	Req.	Static	Both	No	Return the number of instruction and data breakpoint registers.
PAL_FIXED_ADDR	12	Req.	Static	Both	No	Return the fixed component of a processor's directed address.
PAL_FREQ_BASE	13	Opt.	Static	Both	No	Return the frequency of the output clock for use by the platform, if generated by the processor.
PAL_FREQ_RATIOS	14	Req.	Static	Both	No	Return ratio of processor, bus, and interval time counter to processor input clock or output clock for platform use, if generated by the processor.
PAL_GET_HW_POLICY	48	Opt.	Static	Both	Dep.	Get current hardware resource sharing policy.
PAL_LOGICAL_TO_PHYSICAL	42	Opt.	Static	Both	No	Return information on which logical processors map to a physical processor package.
PAL_PERF_MON_INFO	15	Req.	Static	Both	No	Return the number and type of performance monitors.
PAL_PLATFORM_ADDR[a]	16	Opt.	Static	Both	No	Specify processor interrupt block address and I/O port space address.
PAL_PROC_GET_FEATURES	17	Req.	Static	Phys.	No	Return configurable processor features and their current setting.
PAL_PROC_SET_FEATURES[a]	18	Req.	Static	Phys.	No	Enable or disable configurable processor features.
PAL_REGISTER_INFO	39	Req.	Static	Both	No	Return AR and CR register information.
PAL_RSE_INFO	19	Req.	Static	Both	No	Return RSE information.
PAL_SET_HW_POLICY[a]	49	Opt.	Static	Both	Dep.	Set current hardware resource sharing policy.
PAL_VERSION	20	Req.	Static	Both	No	Return version of PAL code.

a. Calling this procedure may affect resources on multiple processors. Please refer to implementation-specific reference manuals for details.

V2-V PAL_PSTATE_INFO

PAL_PSTATE_INFO – Get Information for Power/Performance States (44)

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

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Not dependent

Arguments:

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

Returns:

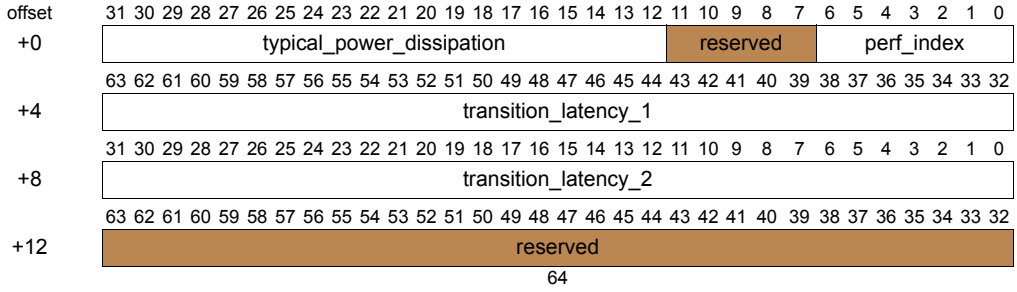
Return Value	Description
status	Return status of the PAL_PSTATE_INFO procedure.
pstate_num	Unsigned integer denoting the number of P-states supported. The maximum value of this field is 16.
dd_info	Dependency domain information
Reserved	0

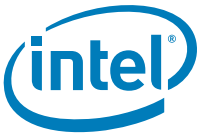
Status:

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 11-41 illustrates the format of the *pstate_buffer*.

Figure 11-41. Layout of *pstate_buffer* Entry

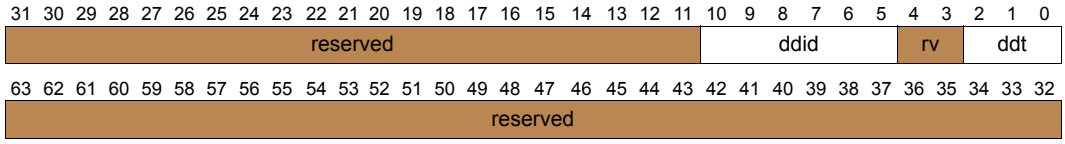




- *typical_power_dissipation* is a 20-bit field denoting the typical processor package power dissipation if all logical processors on the package are placed in this P-state, measured in milliwatts.
- *perf_index* is a 7-bit field denoting the performance index of this P-state, relative to the highest available P-state (P0). This field is enumerated relative to the index of the highest-performing P-state. A value of 100 represents the minimum processor performance in 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 11-42 for *dd_info* layout.

Figure 11-42. Layout of *dd_info* Parameter



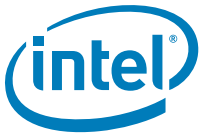
- *ddt* (Dependency Domain Type) is a 3-bit unsigned integer denoting the type of dependency domains that exist on the processor package. The possible values are shown in Table 11-108. See Section 11.6.1, “Power/Performance States (P-states)” on page 2:317 for details of the values in this field.

Table 11-108. Values for *ddt* Field

Value	Description
0	Hardware independent (HIDD)
1	Hardware coordinated (HCDD)
2	Software coordinated (SCDD)
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 2:317.



V2-W PAL_GET/SET_HW_POLICY

PAL_GET_HW_POLICY – Retrieve Current Hardware Resource Sharing Policy (48)

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

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Dependent

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
-9	Call requires PAL memory buffer

Description: This procedure is used to return information on the current hardware resource sharing policy. This procedure can also be used to identify which logical processors (see "PAL_LOGICAL_TO_PHYSICAL – Get Information on Logical to Physical Processor Mappings (42)" on page 2:404 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 that can be returned in the *cur_policy* value are defined in Table 11-80.

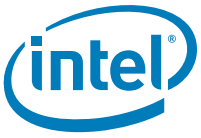


Table 11-80. Hardware policies returned in *cur_policy*

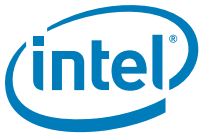
Value	Name	Description
0	Performance	The processor has its hardware resources configured to achieve maximum performance across all logical processors that share hardware with the logical processor the procedure was made on.
1	Fairness	The processor has its hardware resources configured to approximately 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 "PAL_SET_HW_POLICY – Set Current Hardware Resource Sharing Policy (49)" on page 105 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.
5-511		Reserved.
512 and above		Implementation Specific Policy Information. Please refer to processor-specific documentation for information on policies in this range.

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 "PAL_FIXED_ADDR – Get Fixed Geographical Address of Processor (12)" on page 2:390 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 (49)

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

Calling Conv: Static Registers Only

Mode: Physical and Virtual

Buffer: Dependent

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
-9	Call requires PAL memory buffer

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 – Retrieve Current Hardware Resource Sharing Policy (48)" on page 103 for details.

The input argument *policy* selects the hardware policy the caller would like to set. The supported hardware policies are listed in Table 11-116 below. By default the hardware always sets the processor in the performance policy at reset.

Table 11-116. 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.
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.

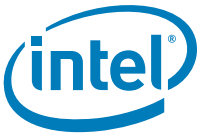


Table 11-116. Processor Hardware Sharing Policies *(Continued)*

Value	Name	Description
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.
4-511		Reserved
512 and above		Implementation Specific Policies. Please refer to processor-specific documentation for information on policies supported in this range.

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.

V2-X PAL_TEST_PROC

PAL_TEST_PROC – Perform a Processor Self-test (258)

Purpose: Performs the second phase of processor self test.

Calling Conv: Stacked Registers

PAL_TEST_PROC may modify some registers marked unchanged in the Stacked Register calling convention. See additional description below.

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_TEST_PROC within the list of PAL procedures.
test_address	Physical address of the memory buffer to be used by processor self-test. The memory attribute of the physical memory buffer must be cacheable (i.e., bit 63 must be zero). See Section 4.4.2 Physical Addressing Memory Attributes for details.
test_info	Input argument specifying the size of the memory buffer passed and the phase of the processor self-test that should be run. See Figure 11-44.
test_params	Input argument specifying the self-test control word and the allowable memory attributes that can be used with the memory buffer. See Figure 11-45.

Returns:

Return Value	Description
status	Return status of the PAL_TEST_PROC procedure.
self-test_state	Formatted 8-byte value denoting the state of the processor after self-test. The format is described in Section 11.2.2.3, “Definition of Self Test State Parameter” on page 2:295.
Reserved	0
Reserved	0

Status:

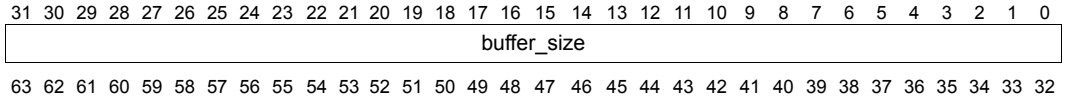
Status Value	Description
1	Call completed without error, but hardware failures occurred during self-test
0	Call completed without error
-2	Invalid argument
-3	Call completed with error

Description: The PAL_TEST_PROC procedure will perform a phase of the processor self-tests as directed by the *test_info* and the *test_control* input parameters.

test_address points to a contiguous memory region to be used by PAL_TEST_PROC. This memory region must be aligned as specified by the alignment return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value. The PAL_TEST_PROC routine requires that the memory has been initialized and that there are no known uncorrected errors in the allocated memory.

The *test_info* input parameter specifies the size of the memory buffer passed to the procedure and which phase of the processor self-test is requested to be run (either phase one or phase two).

Figure 11-44. Layout of *test_info* Argument



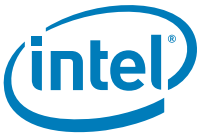
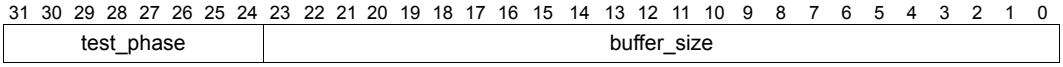


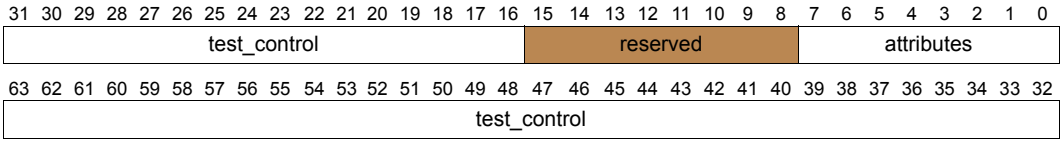
Figure 11-44. Layout of *test_info* Argument



- *buffer_size* indicates the size in bytes of the memory buffer that is passed to this procedure. *buffer_size* must be greater than or equal in size to the *bytes_needed* return value from PAL_TEST_INFO, otherwise this procedure will return with an invalid argument return value.
- *test_phase* defines which phase of the processor self-tests are requested to be run. A value of zero indicates to run phase two of the processor self-tests. Phase two of the processor self-tests are ones that require external memory to execute correctly. A value of one indicates to run phase one of the processor self-tests. Phase one of the processor self-tests are tests run during PALE_RESET and do not depend on external memory to run correctly. When the caller requests to have phase one of the processor self-test run via this procedure call, a memory buffer may be needed to save and restore state as required by the PAL calling conventions. The procedure PAL_TEST_INFO informs the caller about the requirements of the memory buffer.

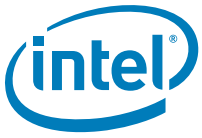
The *test_params* input argument specifies which memory attributes are allowed to be used with the memory buffer passed to this procedure as well as the self-test control word. The self-test control word *test_control* controls the runtime and coverage of the processor self-test phase specified in the *test_phase* parameter.

Figure 11-45. Layout of *test_param* Argument



- *attributes* specifies the memory attributes that are allowed to be used with the memory buffer passed to this procedure. The *attributes* parameter is a vector where each bit represents one of the virtual memory attributes defined by the architecture. The bit field position corresponds to the numeric memory attribute encoding defined in Section 4.4, “Memory Attributes” on page 2:75. The caller is required to support the cacheable attribute for the memory buffer, otherwise an invalid argument will be returned.
- *test_control* is the self-test control word corresponding to the *test_phase* passed. This *test_control* directs the coverage and runtime of the processor self-tests specified by the *test_phase* input argument. Information about the self-test control word can be found in Section 11.2.3, “PAL Self-test Control Word” on page 2:297 and information on if this feature is implemented and the number of bits supported can be obtained by the PAL_TEST_INFO procedure call. If this feature is implemented by the processor, the caller can selectively skip parts of the processor self-test by setting *test_control* bits to a one. If a bit has a zero, this test will be run. The values in the unimplemented bits are ignored. If PAL_TEST_INFO indicated that the self-test control word is not implemented, this procedure will return with an invalid argument status if the caller sets any of the *test_control* bits.

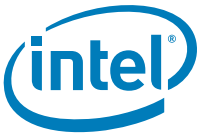
PAL_TEST_PROC will classify the processor after the self-test in one of four states: CATASTROPHIC FAILURE, FUNCTIONALLY RESTRICTED, PERFORMANCE RESTRICTED, or HEALTHY. These processor self-test states are described in Figure 11-9 on page 2:295. If PAL_TEST_PROC returns in the FUNCTIONALLY RESTRICTED or PERFORMANCE RESTRICTED states the *self-test_status* return value can provide



additional information regarding the nature of the failure. In the case of a CATASTROPHIC FAILURE, the procedure does not return.

The procedure will only perform memory accesses to the buffer passed to it using the memory attributes indicated in the *attributes* bit-field. The caller must ensure that the memory region passed to the procedure is in a coherent state.

PAL_TEST_PROC may modify PSR bits or system registers as necessary to test the processor. These bits or registers must be restored upon exit from PAL_TEST_PROC with the exception of the translation caches, which are evicted as a result of testing. PAL_TEST_PROC is free to invalidate all cache contents. If the caller depends on the contents of the cache, they should be flushed before making this call. PAL_TEST_PROC requires that the RSE is set up properly to handle spills and fills to a valid memory location if the contents of the register stack are needed. PAL_TEST_PROC requires that the memory buffer passed to it is not shared with other processors running this procedure in the system at the same time. PAL_TEST_PROC will use this memory region in a non-coherent manner. PAL_TEST_PROC may overwrite floating point registers 32-127 without restoring their values upon exit.



V2-Y PAL_VM_TR_READ

PAL_VM_TR_READ – Read a Translation Register (261)

Purpose: Reads a translation register.

Calling Conv: Stacked Registers

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_VM_TR_READ within the list of PAL procedures.
reg_num	Unsigned 64-bit number denoting which TR to read.
tr_type	Unsigned 64-bit number denoting whether to read an ITR (0) or DTR (1). All other values are reserved.
tr_buffer	Physical address of the 32-byte memory buffer in which translation data is returned.

Returns:

Return Value	Description
status	Return status of the PAL_VM_TR_READ procedure.
TR_valid	Formatted bit vector denoting which fields are valid. See Figure 11-50.
Reserved	0
Reserved	0

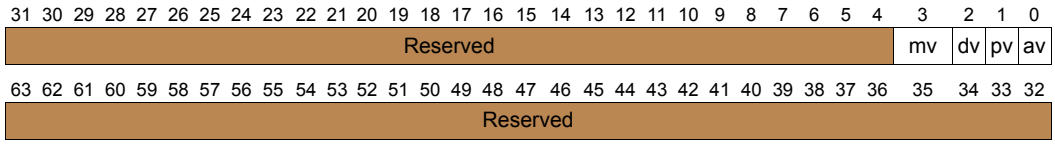
Status:

Status Value	Description
0	Call completed without error.
-2	Invalid argument
-3	Call completed with error.

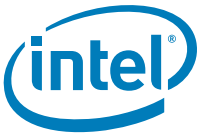
Description: This procedure reads the specified translation register and returns its data in the buffer starting at *tr_buffer*. The format of the data is returned in Translation Insertion Format, as described in Figure 4-5, “Translation Insertion Format,” on page 2:54. In addition, bit 0 of the IFA in Figure 4-5 (an ignored field in the figure) will return whether the translation is valid. If bit 0 is 1, the translation is valid.

Some fields of the translation register returned may be invalid. The validity of these fields is indicated by the return argument *TR_valid*. If these fields are not valid, the caller should ignore the indicated fields when reading the translation register returned in *tr_buffer*.

Figure 11-50. Layout of *TR_valid* Return Value



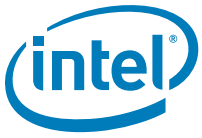
- av – denotes that the access rights field is valid
- pv – denotes that the privilege level field is valid
- dv – denotes that the dirty bit is valid
- mv – denotes that the memory attributes are valid.



A value of 1 denotes a valid field. A value of 0 denotes an invalid field. Any value returned in an invalid field must be ignored.

The *tr_buffer* parameter should be aligned on an 8 byte boundary.

Note: This procedure may have the side effect of flushing all the translation cache entries depending on the implementation.



V2-Z PAL_VP_INIT_ENV

PAL_VP_INIT_ENV – PAL Initialize Virtual Environment (268)

Purpose: Allows a logical processor to enter a virtual environment.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_INIT_ENV within the list of PAL procedures
config_options	64-bit vector of global configuration settings – See Table 11-114. for details
pbase_addr	Host-physical base address of a block of contiguous physical memory for the PAL virtual environment buffer – This memory area must be allocated by the VMM and be 4K aligned. The first logical processor to enter the environment will initialize the physical block for virtualization operations.
vbase_addr	Host-virtual base address of the corresponding physical memory block for the PAL virtual environment buffer – The VMM must maintain the host-virtual to host-physical data and instruction translations in TRs for addresses within the allocated address space. Logical processors in this virtual environment will use this address when transitioning to virtual mode operations.

Returns:

Return Value	Description
status	Return status of the PAL_VP_INIT_ENV procedure
vsa_base	Virtualization Service Address – VSA specifies the virtual base address of the PAL virtualization services in this virtual environment.
Reserved	0
Reserved	0

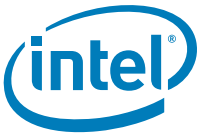
Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: This procedure allows a logical processor to enter a virtual environment. This call must be made after calling PAL_VP_ENV_INFO and before calling other PAL virtualization procedures and services. All of the logical processors in a virtual environment share the same **PAL virtual environment buffer**. The buffer must be 4K aligned. The first logical processor entering the virtual environment initializes the buffer provided by the VMM. Subsequent processors can enter the virtual environment at any time and will not perform initialization to the buffer.

PAL_VP_ENV_INFO must be called before this procedure to determine the configuration options and size requirements for the virtual environment. The VMM is required to maintain the ITR and DTR translations of the PAL virtual environment buffer throughout this procedure. See "PAL_VP_ENV_INFO – PAL Virtual Environment Information (266)" on page 2:474 for more information on PAL_VP_ENV_INFO.

After this procedure, it is optional for the VMM to maintain the TR mapping for the PAL virtual environment buffer. If the TR translations for the buffer are not installed, the



VMM must not make any PAL virtualization service calls; and the VMM must be prepared to handle DTLB faults during any PAL virtualization procedure calls.

Table 11-114 shows the layout of the *config_options* parameter. The *config_options* parameter configures the global configuration options and global virtualization optimizations for all the logical processors in the virtual environment. All logical processors in the virtual environment must specify the same value in the *config_options* parameter during PAL_VP_INIT_ENV, otherwise processor operation is undefined.

Table 11-114. *config_options* – Global Configuration Options

	Field	Bit	Description
Global Configuration Options	initialize	0	If 1, this procedure will initialize the PAL virtual environment buffer for this virtual environment. If 0, this procedure will not initialize the PAL virtual environment buffer. On a multiprocessor system, the VMM must wait until this procedure completes on the first logical processor before calling this procedure on additional logical processors; otherwise processor operation is undefined.
	fr_pmc	1	If 1, for virtualization intercepts the performance counters are disabled by setting PSR.up and pp to 0, see Section 11.7.3.1, “PAL Virtualization Intercept Handoff State” on page 2:335 for details on PSR settings at virtualization intercepts; for all other IVA-based interruptions PSR.pp and up are set according to Interruption State column described in Processor Status Field table described in Table 3-2, “Processor Status Register Fields” on page 2:24. The VMM must have DCR.pp equal to 0 when the *fr_pmc* option is 1, whenever the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. See Section 11.7.2, “Interruption Handling in a Virtual Environment” on page 2:333 and Table 11-21, “IVA Settings after PAL Virtualization-related Procedures and Services” on page 2:334 for details on per-virtual-processor host IVT. If 0, PSR.pp and up are set according to Interruption State column described in Processor Status Field table described in Table 3-2, “Processor Status Register Fields” on page 2:24
	be	2	Big-endian – Indicates the endian setting of the VMM. If 1, the values in the VPD are stored in big-endian format and the PAL services calls are made with PSR.be bit equal to 1. If 0, the values in the VPD are stored in little-endian format and the PAL services calls are made with PSR.be bit equal to 0. The VMM must match DCR.be with the value set in this field when the IVA control register on the logical processor is set to point to the per-virtual-processor host IVT. See Section 11.7.2, “Interruption Handling in a Virtual Environment” on page 2:333 and Table 11-21, “IVA Settings after PAL Virtualization-related Procedures and Services” on page 2:334 for details on per-virtual-processor host IVT.
	Reserved	7:3	Reserved.

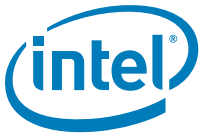
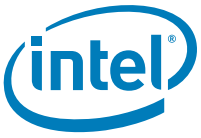


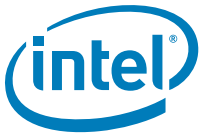
Table 11-114. *config_options* – Global Configuration Options (Continued)

	Field	Bit	Description
Global Virtualization Optimizations	opcode	8	This bit must be set to 1 – opcode information will be provided to the VMM during PAL intercepts within the virtual environment. This opcode may or may not be guaranteed to be the opcode that triggered the intercept. See Table 11-113, "vp_env_info – Virtual Environment Information Parameter" on page 2:474 for details. This procedure returns an error if this bit is not set to 1.
	cause	9	If 1, the causes of virtualization intercepts will be provided to the VMM during PAL intercept handoffs within the virtual environment. No information will be provided if 0. See Section 11.7.3.1, "PAL Virtualization Intercept Handoff State" on page 2:335 for details of virtualization intercept handoffs.
	impl	63	Implementation-specific configuration option. This field is ignored if not implemented. Please refer to processor-specific documentation for details.

The *fr_pmc* bit in the global *config_options* parameter specifies whether the performance counters will be frozen when the Virtualization optimizations specified in the Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*) are running. When a virtual processor is running, the *vac* field in the corresponding VPD specifies whether a certain virtualization accelerations are enabled. If the *fr_pmc* in the virtual environment was also enabled, the performance counters will be frozen when the enabled virtualization optimizations are running. See Section 11.7.4, "Virtualization Optimizations" on page 2:337 for details on Virtualization Acceleration Control (*vac*) and Virtualization Disable Control (*vdc*).

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, "Processor Virtualization" on page 2:44 and "PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)" on page 2:447 for details.





V2-AA PAL_VP_RESTORE

PAL_VP_RESTORE – PAL Restore Virtual Processor (270)

Purpose: Restores virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_RESTORE within the list of PAL procedures.
vpd	64-bit host-virtual pointer to the Virtual Processor Descriptor (VPD).
Reserved	0
Reserved	0

Returns:

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

Status:

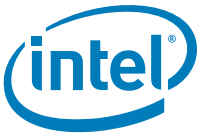
Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

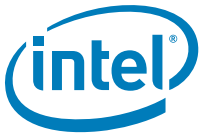
Description: PAL_VP_RESTORE performs an implementation-specific restore operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* and the PAL virtual environment buffer must be mapped by instruction and data translation registers (TR). The instruction and data translation must be maintained until after the next invocation of PAL_VP_SAVE or PAL_VPS_SAVE and a different host IVT is set up by the VMM by writing to the IVA control register. PAL_VP_RESTORE configures the logical processor to run the specified virtual processor by loading implementation-specific virtual processor context from the VPD, and returns control back to the VMM.

This procedure performs an implicit PAL_VPS_SYNC_WRITE; there is no need for the VMM to invoke PAL_VPS_SYNC_WRITE unless the VPD values are modified before resuming the virtual processor. After the procedure, the caller is responsible for restoring all of the architectural state before resuming to the new virtual processor through PAL_VPS_RESUME_NORMAL or PAL_VPS_RESUME_HANDLER.

Upon completion of this procedure, the IVA-based interruptions will be delivered to the host IVT associated with this virtual processor.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:447 for details.





V2-BB PAL_VP_SAVE / PAL_VP_TERMINATE

PAL_VP_SAVE – PAL Save Virtual Processor (271)

Purpose: Saves virtual processor state for the specified *vpd* on the logical processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_SAVE within the list of PAL procedures
vpd	64-bit host-virtual pointer to the Virtual Processor Descriptor (VPD)
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_SAVE procedure
Reserved	0
Reserved	0
Reserved	0

Status:

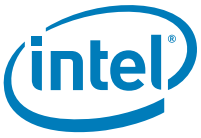
Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: PAL_VP_SAVE performs an implementation-specific save operation of the virtual processor specified by the *vpd* parameter on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR).

This procedure performs an implicit PAL_VPS_SYNC_READ; there is no need for the VMM to invoke PAL_VPS_SYNC_READ to synchronize the implementation-specific control resources before this procedure.

Upon completion of this procedure, the IVA-based interruptions will continue to be delivered to the host IVT associated with this virtual processor. After this procedure, the VMM can setup the IVA control register to use a different host IVT.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:447 for details.



PAL_VP_TERMINATE – PAL Terminate Virtual Processor (272)

Purpose: Terminates operation for the specified virtual processor.

Calling Conv: Stacked Registers

Mode: Virtual

Buffer: Dependent

Arguments:

Argument	Description
index	Index of PAL_VP_TERMINATE within the list of PAL procedures
vpd	64-bit host virtual pointer to the Virtual Processor Descriptor (VPD)
iva	Optional 64-bit host virtual pointer to the IVT when this procedure is done
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_VP_TERMINATE procedure
Reserved	0
Reserved	0
Reserved	0

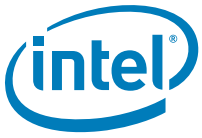
Status:

Status Value	Description
0	Call completed without error
-1	Unimplemented procedure
-2	Invalid argument
-3	Call completed with error
-9	Call requires PAL memory buffer

Description: Terminates operation of the virtual processor specified by *vpd* on the logical processor. The host virtual to host physical translation of the 64K region specified by *vpd* must be mapped by instruction and data translation registers (TR). See Section 11.10.2.1.3, “Making PAL Procedure Calls in Physical or Virtual Mode” on page 2:359 for details on data translation requirements of memory buffer pointers passed as arguments to PAL procedures. All resources allocated for the execution of the virtual machine are freed.

Upon successful execution of PAL_VP_TERMINATE procedure and if the *iva* parameter is non-zero, the IVA control register will contain the value from the *iva* parameter.

This procedure returns unimplemented procedure when virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:447 for details.



V2-CC PAL_PROC_GET / SET FEATURES

PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)

Purpose: Provides information about configurable processor features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PROC_GET_FEATURES within the list of PAL procedures.
Reserved	0
feature_set	Feature set information is being requested for.
Reserved	0

Returns:

Return Value	Description
status	Return status of the PAL_PROC_GET_FEATURES procedure.
features_avail	64-bit vector of features implemented. See Table 11-112.
feature_status	64-bit vector of current feature settings. See Table 11-112.
feature_control	64-bit vector of features controllable by software.

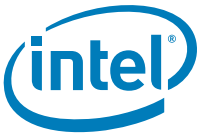
Status:

Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

Description: PAL_PROC_GET_FEATURES and PAL_PROC_SET_FEATURES procedure calls are used together to describe current settings of processor features and to allow modification of some of these processor features.

The *feature_set* input argument for PAL_PROC_GET_FEATURES describes which processor *feature_set* information is being requested. Table 11-112 describes processor *feature_set* zero. The *feature_set* values are split into two categories: architected and implementation-specific. The architected feature sets have values from 0-15. The implementation-specific feature sets are values 16 and above. The architected feature sets are described in this document. The implementation-specific feature sets are described in processor-specific documentation.

This procedure will return an invalid argument if an unsupported architectural *feature_set* is passed as an input. Implementation-specific feature sets will start at 16 and will expand in an ascending order as new implementation-specific feature sets are added. The return *status* is used by the caller to know which implementation-specific feature sets are currently supported on a particular processor.



For each valid *feature_set*, this procedure returns which processor features are implemented in the *features_avail* return argument, the current feature setting is in *feature_status* return argument, and the feature controllability in the *feature_control* return argument. Only the processor features which are implemented and controllable can be changed via PAL_PROC_SET_FEATURES. Features for which features_avail are 0 (unimplemented features) also have features_status and features_control of 0.

In Table 11-112, the *class* field indicates whether a feature is required to be available (*Req.*) or is optional (*Opt.*). The *control* field indicates which features are required to be controllable. *Req.* indicates that the feature must be controllable, *Opt.* indicates that the feature may optionally be controllable, and *No* indicates that the feature cannot be controllable. The *control* field applies only when the feature is available. The sense of the bits is chosen so that for features which are controllable, the default hand-off value at exit from PALE_RESET should be 0. PALE_CHECK and PALE_INIT will not modify these features.

Table 11-112. Processor Features

Bit	Class	Control	Scope	Description
63	Opt.	Req.	May[a]	Enable BERR promotion. When 1, the Bus Error (BERR) signal is promoted to the Bus Initialization (BINIT) signal, and the BINIT pin is asserted on the occurrence of each Bus Error. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
62	Opt.	Req.	May	Enable MCA promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Error signal, and the BERR pin is assert on each occurrence of an MCA. Setting this bit has no effect if BERR signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
61	Opt.	Req.	May	Enable MCA to BINIT promotion. When 1, machine check aborts (MCAs) are promoted to the Bus Initialization signal, and the BINIT pin is assert on each occurrence of an MCA. Setting this bit has no effect if BINIT signalling is disabled. (See PAL_BUS_GET/SET_FEATURES)
60	Opt.	Req.	No[b]	Enable CMCI promotion When 1, Corrected Machine Check Interrupts (CMCI) are promoted to MCAs. They are also further promoted to BERR if bit 39, Enable MCA promotion, is also set and they are promoted to BINIT if bit 38, Enable MCA to BINIT promotion, is also set. This bit has no effect if MCA signalling is disabled (see PAL_BUS_GET/SET_FEATURES)
59	Opt.	Req.	May	Disable Cache. When 0, the processor performs cast outs on cacheable pages and issues and responds to coherency requests normally. When 1, the processor performs a memory access for each reference regardless of cache contents and issues no coherence requests and responds as if the line were not present. Cache contents cannot be relied upon when the cache is disabled. WARNING: Semaphore instructions may not be atomic or may cause Unsupported Data Reference faults if caches are disabled.
58	Opt.	Req.	May	Disable Coherency. When 0, the processor uses normal coherency requests and responses. When 1, the processor answers all requests as if the line were not present.
57	Opt.	Req.	May	Disable Dynamic Power Management (DPM). When 0, the hardware may reduce power consumption by removing the clock input from idle functional units. When 1, all functional units will receive clock input, even when idle.
56	Opt.	Req.	May	Disable a BINIT on internal processor time-out. When 0, the processor may generate a BINIT on an internal processor time-out. When 1, the processor will not generate a BINIT on an internal processor time-out. The event is silently ignored.

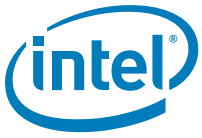


Table 11-112. Processor Features (Continued)

Bit	Class	Control	Scope	Description
55	Opt.	Req.	May	Enable external notification when the processor detects hardware errors caused by environmental factors that could cause loss of deterministic behavior of the processor. When 1, this bit will enable external notification, when 0 external notification is not provided. The type of external notification of these errors is processor-dependent. A loss of processor deterministic behavior is considered to have occurred if these environmentally induced errors cause the processor to deviate from its normal execution and eventually causes different behavior which can be observed at the processor bus pins. Processor errors that do not have this effects (i.e., software induced machine checks) may or may not be promoted depending on the processor implementation.
54	Opt.	Req.	No	Enable the use of the vmsw instruction. When 0, the `vmsw` instruction causes a Virtualization fault when executed at the most privileged level. When 1, this bit will enable normal operation of the `vmsw` instruction. This bit has no effect if virtual machine features are disabled (see bit 40).
53	Opt.	Req.	May	Enable MCA signaling on unconsumed data-poisoning event detection. When 0, a CMCI will be signaled on error detection. When 1, an MCA will be signaled on error detection. Note that the reported error severity depends on which method is chosen for signaling; see Section 11.3.2.3, “Unconsumed Data-Poisoning Event Handling” for details.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.
52	Opt.	Req.	May	Disable P-states. Provides the ability to disable p-states when they are implemented by the processor. When the feature is available and status is 1 or when the feature is not available, the PAL P-state procedures (PAL_PSTATE_INFO, PAL_SET_PSTATE, PAL_GET_PSTATE) will return with a status of -1 (Unimplemented procedure). When the feature is available and the status is 0, the PAL P-state procedures will operate normally.
51:48	N/A	N/A	N/A	Reserved
47	Opt.	Opt.	May	Disable Dynamic branch prediction. When 0, the processor may predict branch targets and speculatively execute, but may not commit results. When 1, the processor must wait until branch targets are known to execute.
46	Opt	Opt.	May	Disable Dynamic Instruction Cache Prefetch. When 0, the processor may prefetch into the caches any instruction which has not been executed, but whose execution is likely. When 1, instructions may not be fetched until needed or hinted for execution. (Prefetch for a hinted branch is allowed even when dynamic instruction cache prefetch is disabled.)
45	Opt.	Opt.	May	Disable Dynamic Data Cache Prefetch. When 0, the processor may prefetch into the caches any data which has not been accessed by instruction execution, but which is likely to be accessed. When 1, no data may be fetched until it is needed for instruction execution or is fetched by an lfetch instruction.
44	Opt.	Req.	No	Disable Spontaneous Deferral. When 1, the processor may optionally defer speculative loads that do not encounter any exception conditions, but that trigger other implementation-dependent conditions (e.g., cache miss). This behavior is gated by the programming model described in Section 5.5.5, “Deferral of Speculative Load Faults” on page 2:105. When 0, spontaneous deferral is disabled.
43	Opt.	Opt.	No	Disable Dynamic Predicate Prediction. When 0, the processor may predict predicate results and execute speculatively, but may not commit results until the actual predicates are known. When 1, the processor shall not execute predicated instructions until the actual predicates are known.

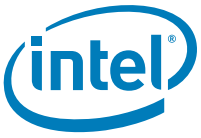


Table 11-112. Processor Features (Continued)

Bit	Class	Control	Scope	Description
42	Opt.	No	RO[c]	XR1 through XR3 implemented. Denotes whether XR1 - XR3 are implemented for machine check recovery. 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.
41	Opt.	No	RO	XIP, XPSR, and XFS implemented. Denotes whether XIP, XPSR, and XFS are implemented for machine check recovery. 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.
40	Opt.	Opt.	No	Virtual Machine features implemented and enabled. When 1, PSR.vm is implemented and virtual machines features are not disabled. When 0 (features_status) and when the corresponding features_avail bit is 1, virtual machines features are implemented but are disabled. When both the features_avail and features_status bits are 0, virtual machine features are not implemented. If implemented and controllable, virtual machine features may be disabled by writing this bit to 0 with PAL_PROC_SET_FEATURES. However, virtual machine features cannot be re-enabled except via a power-on; hence, if virtual machine features are disabled, this bit reads as 0 for both features_status and features_control (but still 1 for features_avail).
39	Opt.	Req.	May	Variable P-state performance: A value of 1 indicates that the processor is optimizing 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.
38	Opt.	No	RO	Simple implementation of unimplemented instruction addresses. Denotes how an unimplemented instruction address is recorded in IIP on an Unimplemented Instruction Address trap or fault. When 1, the full unimplemented address is recorded in IIP; when 0, the address is sign extended (virtual addresses) or zero extended (physical addresses). See Section 3.3.5.3, “Interruption Instruction Bundle Pointer (IIP – CR19)” for details. 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.
37	Opt.	No	RO	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.
36	Opt.	No	RO	Unimplemented instruction address reported as fault. Denotes how the processor reports the detection of unimplemented instruction addresses. When 1, the processor reports an Unimplemented Instruction Address fault on the unimplemented address; when 0, it reports an Unimplemented Instruction Address trap on the previous instruction in program order. 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.
35	Opt.	Req.	May	Disable data speculation and the ALAT. When 1, data speculation checks (`chk.a`) always fail (i.e., always branch to the target address), thus triggering recovery code; check loads (`ld.c`) always re-load the target register. When 0, data speculation works as normal.

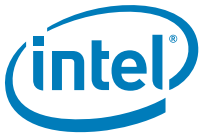


Table 11-112. Processor Features (Continued)

Bit	Class	Control	Scope	Description
34	Opt.	No	RO	Interruption Instruction Bundle interruption registers (IIB0, IIB1) implemented. Denotes whether IIB registers are implemented. 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.
33	Opt.	No	RO	Interval Timer Offset register (ITO) implemented. Denotes whether ITO register is implemented. 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.
32	Opt.	No	RO	Performance Monitoring Count Halted and Virtual Memory Mask control bits implemented. Denotes whether PMC.ch and PMC.vmm have been implemented. 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.
31:0	N/A	N/A	N/A	Reserved

a. May-span-multiple-logical-processors. Readers should refer to implementation-specific document for details.
b. Setting this bit affect logical-processor only.
c. Read-only bit.

PAL_PROC_SET_FEATURES – Set Processor Dependent Features (18)

Purpose: Enables/disables specific processor features.

Calling Conv: Static Registers Only

Mode: Physical

Buffer: Not dependent

Arguments:

Argument	Description
index	Index of PAL_PROC_SET_FEATURES within the list of PAL procedures.
feature_select	64-bit vector denoting desired state of each feature (1=select, 0=non-select).
feature_set	Feature set to apply changes to. See PAL_PROC_GET_FEATURES for more information on feature sets.
Reserved	0

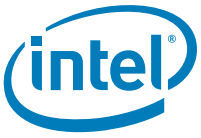
Returns:

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

Status:

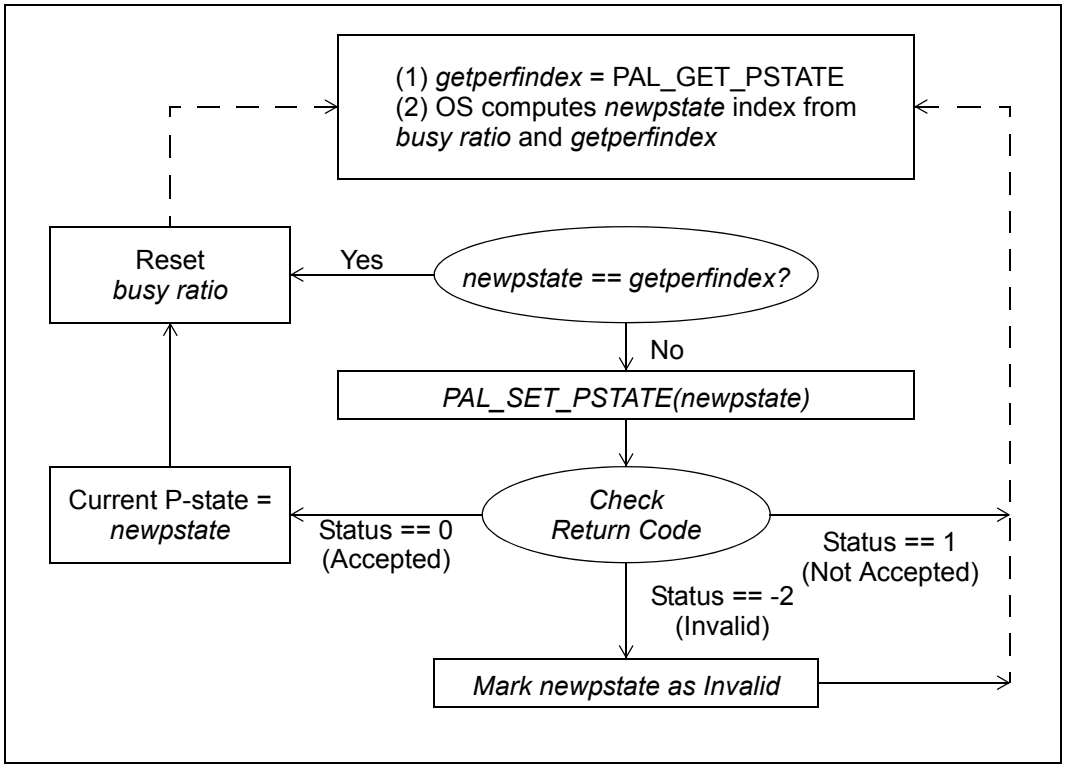
Status Value	Description
1	Call completed without error; The *feature_set* passed is not supported but a *feature_set* of a larger value is supported
0	Call completed without error
-2	Invalid argument
-3	Call completed with error
-8	*feature_set* passed is beyond the maximum *feature_set* supported

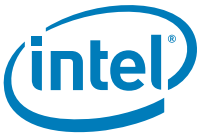
Description: PAL_PROC_GET_FEATURES should be called to ascertain the implemented processor features and their current setting before calling PAL_PROC_SET_FEATURES. The list of possible processor features is defined in Table 11-112. Any attempt to set processor features which cannot be set will be ignored.

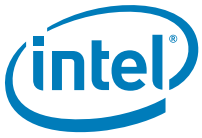


V2-DD Figure 13-6

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







V2-EE PMI Flows

13.3.3 PMI Flows

Processors based on the Itanium architecture implement the Platform Management Interrupt (PMI) to enable platform developers to provide high level system functions, such as power management and security, in a manner that is transparent not only to the application software but also to the operating system.

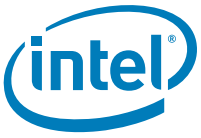
When the processor detects a PMI event it will transfer control to the registered PAL PMI entry point. PAL will set up the hand off state which includes the vector information for the PMI and hand off control to the registered SAL PMI handler. To reduce the PMI overhead time, the PAL PMI handler will not save any processor architectural state to memory. Please see Section 11.5, “Platform Management Interrupt (PMI)” for more information on PAL PMI handling.

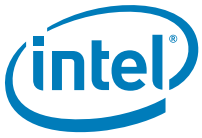
The SAL PMI handler may choose to save some additional register state to SAL allocated memory to handle the specific platform event that generated the PMI.

The OS will not see the PMI events generated by the platform. The platform developer can use PMI interrupts to provide features to differentiate their platform.

PMI handling was designed to be executed with minimal overhead. The SAL firmware code copies the PAL and SAL PMI handlers to RAM during system reset and registers these entry-points with the processor. This code is then run with the cacheable memory attribute to improve performance.

Depending on the implementation and the platform, there may be no special hardware protection of the PMI code's memory area in RAM, and the protection of this code space may be through the OS memory management’s paging mechanism. SAL sets the correct attributes for this memory space and passes this information to the OS through the Memory Descriptor Table from EfiGetMemoryMap() [UEFI].





V3-A br — Branch

br — Branch

Format:

(*qp*) *br.btype.bwh.ph.dh* $target_{25}$	ip_relative_form	B1
(*qp*) *br.btype.bwh.ph.dh* $b_1 = target_{25}$	call_form, ip_relative_form	B3
br.btype.bwh.ph.dh $target_{25}$	counted_form, ip_relative_form	B2
br.ph.dh $target_{25}$	pseudo-op	
(*qp*) *br.btype.bwh.ph.dh* b_2	indirect_form	B4
(*qp*) *br.btype.bwh.ph.dh* $b_1 = b_2$	call_form, indirect_form	B5
br.ph.dh b_2	pseudo-op	

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

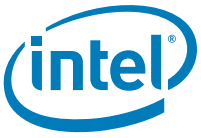
Branches can be either IP-relative, or indirect. For IP-relative branches, the $target_{25}$ operand, in assembly, specifies a label to branch to. This is encoded in the branch instruction as a signed immediate displacement (imm_{21}) between the target bundle and the bundle containing this instruction (imm_{21} = $target_{25}$ - IP >> 4). For indirect branches, the target address is taken from BR b_2.

Table 2-6. Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-rel or Indirect
call	Conditional procedure call	Qualifying predicate	IP-rel or Indirect
ret	Conditional procedure return	Qualifying predicate	Indirect
ia	Invoke IA-32 instruction set	Unconditional	Indirect
cloop	Counted loop branch	Loop count	IP-rel
ctop, cexit	Mod-scheduled counted loop	Loop count and epilog count	IP-rel
wtop, wexit	Mod-scheduled while loop	Qualifying predicate and epilog count	IP-rel

There are two pseudo-ops for unconditional branches. These are encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For the basic branch types, the



branch condition is simply the value of the specified predicate register. These basic branch types are:

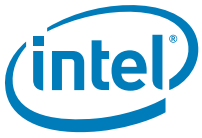
- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR b_1.
  - The values in the DAHRs (if implemented) are pushed onto the DAHS, and the DAHRs revert to default values.
- **return:** If the qualifying predicate is 1, the branch is taken and the following occurs:
  - CFM, EC, and the current privilege level are restored from PFS. (The privilege level is restored only if this does not increase privilege.)
  - The caller's stack frame is restored.
  - If the return lowers the privilege, and PSR.lp is 1, then a Lower-Privilege Transfer trap is taken.
  - The values in the DAHRs (if implemented) are copied from the top level of the DAHS, the DAHS is popped, and the bottom level of the DAHS reverts to default values.
- **ia:** The branch is taken unconditionally, if it is not intercepted by the OS. The effect of the branch is to invoke the IA-32 instruction set (by setting PSR.is to 1) and begin processing IA-32 instructions at the virtual linear target address contained in BR b_2{31:0}. If the qualifying predicate is not PR 0, an Illegal Operation fault is raised. If instruction set transitions are disabled (PSR.di is 1), then a Disabled Instruction Set Transition fault is raised.

  The IA-32 target effective address is calculated relative to the current code segment, i.e. EIP{31:0} = BR b_2{31:0} - CSD.base. The IA-32 instruction set can be entered at any privilege level, provided PSR.di is 0. If PSR.dfh is 1, a Disabled FP Register fault is raised on the target IA-32 instruction. No register bank switch nor change in privilege level occurs during the instruction set transition.

  Software must ensure the code segment descriptor (CSD) and selector (CS) are loaded before issuing the branch. If the target EIP value exceeds the code segment limit or has a code segment privilege violation, an IA_32_Exception(GPFault) is raised on the target IA-32 instruction. For entry into 16-bit IA-32 code, if BR b_2 is not within 64K-bytes of CSD.base a GPFault is raised on the target instruction. EFLAG.rf is unmodified until the successful completion of the first IA-32 instruction. PSR.da, PSR.id, PSR.ia, PSR.dd, and PSR.ed are cleared to zero after `br.ia` completes execution and before the first IA-32 instruction begins execution. EFLAG.rf is not cleared until the target IA-32 instruction successfully completes.

  Software must set PSR properly before branching to the IA-32 instruction set; otherwise processor operation is undefined. See Table 3-2, "Processor Status Register Fields" on page 2:24 for details.

  Software must issue a `mf` instruction before the branch if memory ordering is required between IA-32 processor consistent and Itanium unordered memory



references. The processor does not ensure Itanium-instruction-set-generated writes into the instruction stream are seen by subsequent IA-32 instruction fetches. `br.ia` does not perform an instruction serialization operation. The processor does ensure that prior writes (even in the same instruction group) to GRs and FRs are observed by the first IA-32 instruction. Writes to ARs within the same instruction group as `br.ia` are not allowed, since `br.ia` may implicitly reads all ARs. If an illegal RAW dependency is present between an AR write and `br.ia`, the first IA-32 instruction fetch and execution may or may not see the updated AR value.

IA-32 instruction set execution leaves the contents of the ALAT undefined. Software can not rely on ALAT values being preserved across an instruction set transition. All registers left in the current register stack frame are undefined across an instruction set transition. On entry to IA-32 code, existing entries in the ALAT are ignored. If the register stack contains any dirty registers, an Illegal Operation fault is raised on the `br.ia` instruction. The current register stack frame is forced to zero. To flush the register file of dirty registers, the `flushrs` instruction must be issued in an instruction group preceding the `br.ia` instruction. To enhance the performance of the instruction set transition, software can start the register stack flush in parallel with starting the IA-32 instruction set by 1) ensuring `flushrs` is exactly one instruction group before the `br.ia`, and 2) `br.ia` is in the first B-slot. `br.ia` should always be executed in the first B-slot with a hint of “static-taken” (default), otherwise processor performance will be degraded.

If a `br.ia` causes any Itanium traps (e.g., Single Step trap, Taken Branch trap, or Unimplemented Instruction Address trap), IIP will contain the original 64-bit target IP. (The value will not have been zero extended from 32 bits.)

Another branch type is provided for simple counted loops. This branch type uses the Loop Count application register (LC) to determine the branch condition, and does not use a qualifying predicate:

- **cloop:** If the LC register is not equal to zero, it is decremented and the branch is taken.

In addition to these simple branch types, there are four types which are used for accelerating modulo-scheduled loops (see also Section 4.5.1, “Modulo-scheduled Loop Support” on page 1:75). Two of these are for counted loops (which use the LC register), and two for while loops (which use the qualifying predicate). These loop types use register rotation to provide register renaming, and they use predication to turn off instructions that correspond to empty pipeline stages.

The Epilog Count application register (EC) is used to count epilog stages and, for some while loops, a portion of the prolog stages. In the epilog phase, EC is decremented each time around and, for most loops, when EC is one, the pipeline has been drained, and the loop is exited. For certain types of optimized, unrolled software-pipelined loops, the target of a `br.cexit` or `br.wexit` is set to the next sequential bundle. In this case, the pipeline may not be fully drained when EC is one, and continues to drain while EC is zero.

For these modulo-scheduled loop types, the calculation of whether the branch is taken or not depends on the kernel branch condition (LC for counted types, and the qualifying predicate for while types) and on the epilog condition (whether EC is greater than one or not).

These branch types are of two categories: top and exit. The top types (ctop and wtop) are used when the loop decision is located at the bottom of the loop body and therefore a taken branch will continue the loop while a fall through branch will exit the loop. The

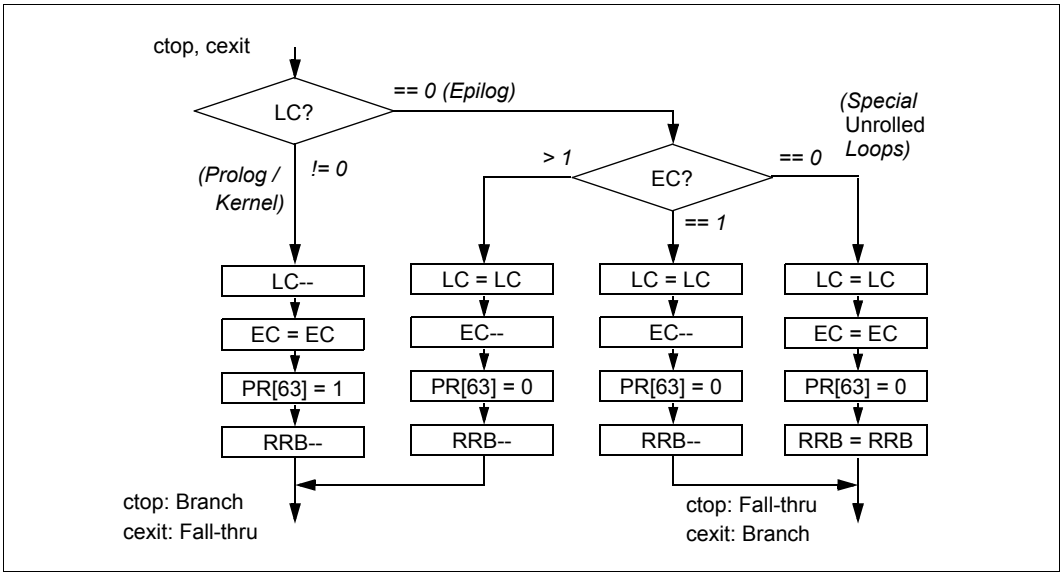
exit types (cexit and wexit) are used when the loop decision is located somewhere other than the bottom of the loop and therefore a fall though branch will continue the loop and a taken branch will exit the loop. The exit types are also used at intermediate points in an unrolled pipelined loop. (For more details, see Section 4.5.1, “Modulo-scheduled Loop Support” on page 1:75).

The modulo-scheduled loop types are:

- **ctop** and **cexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.ctop`, the branch is taken if either LC is non-zero or EC is greater than one. For `br.cexit`, the opposite is true. It is not taken if either LC is non-zero or EC is greater than one and is taken otherwise.

  These branch types also use LC and EC to control register rotation and predicate initialization. During the prolog and kernel phase, when LC is non-zero, LC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero, the epilog phase is entered, and EC counts down. When `br.ctop` or `br.cexit` is executed with LC equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If LC and EC are equal to zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-3.

Figure 2-3. Operation of br.ctop and br.cexit

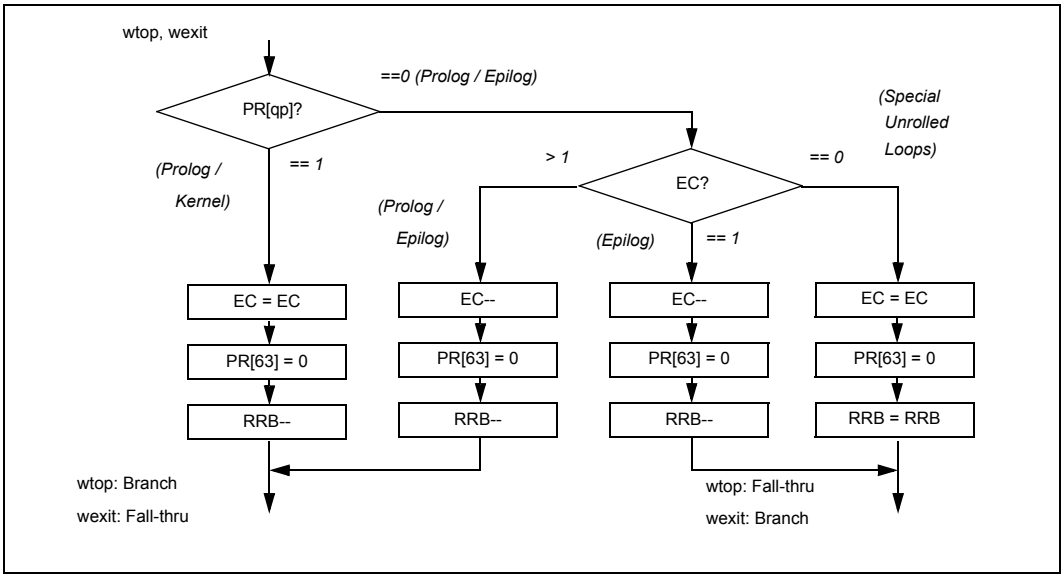


wtop and **wexit:** These branch types behave identically, except in the determination of whether to branch or not. For `br.wtop`, the branch is taken if either the qualifying predicate is one or EC is greater than one. For `br.wexit`, the opposite is true. It is not taken if either the qualifying predicate is one or EC is greater than one, and is taken otherwise.

These branch types also use the qualifying predicate and EC to control register rotation and predicate initialization. During the prolog phase, the qualifying predicate is either zero or one, depending upon the scheme used to program the loop. During the kernel phase, the qualifying predicate is one. During the epilog phase, the qualifying predicate is zero, and EC counts down. When `br.wtop` or `br.wexit` is executed with the qualifying predicate equal to zero and EC equal to one, a final decrement of EC and a final register rotation are done. If the qualifying

predicate and EC are zero, register rotation stops. These other effects are the same for the two branch types, and are described in Figure 2-4.

Figure 2-4. Operation of br.wtop and br.wexit



The loop-type branches (`br.cloop`, `br.ctop`, `br.cexit`, `br.wtop`, and `br.wexit`) are only allowed in instruction slot 2 within a bundle. Executing such an instruction in either slot 0 or 1 will cause an Illegal Operation fault, whether the branch would have been taken or not.

Read after Write (RAW) and Write after Read (WAR) dependency requirements are slightly different for branch instructions. Changes to BRs, PRs, and PFS by non-branch instructions are visible to a subsequent branch instruction in the same instruction group (i.e., a limited RAW is allowed for these resources). This allows for a low-latency compare-branch sequence, for example. The normal RAW requirements apply to the LC and EC application registers, and the RRBs.

Within an instruction group, a WAR dependency on PR 63 is not allowed if both the reading and writing instructions are branches. For example, a `br.wtop` or `br.wexit` may not use PR[63] as its qualifying predicate and PR[63] cannot be the qualifying predicate for any branch preceding a `br.wtop` or `br.wexit` in the same instruction group.

For dependency purposes, the loop-type branches effectively always write their associated resources, whether they are taken or not. The cloop type effectively always writes LC. When LC is 0, a cloop branch leaves it unchanged, but hardware may implement this as a re-write of LC with the same value. Similarly, `br.ctop` and `br.cexit` effectively always write LC, EC, the RRBs, and PR[63]. `br.wtop` and `br.wexit` effectively always write EC, the RRBs, and PR[63].

Values for various branch hint completers are shown in the following tables. Whether Prediction Strategy hints are shown in Table 2-7. Sequential Prefetch hints are shown in Table 2-8. Branch Cache Deallocation hints are shown in Table 2-9. See Section 4.5.2, “Branch Prediction Hints” on page 1:78.

Table 2-7. Branch Whether Hint

bwh Completer	Branch Whether Hint
spnt	Static Not-Taken
sptk	Static Taken
dpnt	Dynamic Not-Taken
dptk	Dynamic Taken

Table 2-8. Sequential Prefetch Hint

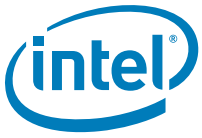
ph Completer	Sequential Prefetch Hint
few or *none*	Few lines
many	Many lines

Table 2-9. Branch Cache Deallocation Hint

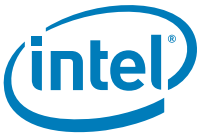
dh Completer	Branch Cache Deallocation Hint
none	Don't deallocate
clr	Deallocate branch information

Operation:

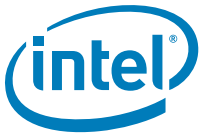
```
if (ip_relative_form)                       // determine branch target
    tmp_IP = IP + sign_ext((imm21 << 4), 25);
else // indirect_form
    tmp_IP = BR[b2];

if (btype != 'ia')                          // for Itanium branches,
    tmp_IP = tmp_IP & ~0xf;                 //  ignore bottom 4 bits of target

lower_priv_transition = 0;

switch (btype) {
    case 'cond':                            // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case 'call':                            // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;              // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;             // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;

            dahr_push_onto_dahs();
        }
```



```
    break;

case 'ret':                          // return restores stack frame
    tmp_taken = PR[qp];
    if (tmp_taken) {
        // tmp_growth indicates the amount to move logical TOP *up*:
        // tmp_growth = sizeof(previous out) - sizeof(current frame)
        // a negative amount indicates a shrinking stack
        tmp_growth = (AR[PFS].pfm.sof - AR[PFS].pfm.sol) - CFM.sof;
        alat_frame_update(-AR[PFS].pfm.sol, 0);
        rse_fatal = rse_restore_frame(AR[PFS].pfm.sol,
                                      tmp_growth, CFM.sof);
        if (rse_fatal) {
        // See Section 6.4, "RSE Operation" on page 2:137
            CFM.sof = 0;
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;
        } else // normal branch return
            CFM = AR[PFS].pfm;

        rse_enable_current_frame_load();
        AR[EC] = AR[PFS].pec;
        if (PSR.cpl u< AR[PFS].ppl) {   // ... and restores privilege
            PSR.cpl = AR[PFS].ppl;
            lower_priv_transition = 1;
        }

        dahr_pop_from_dahs();
    }
    break;

case 'ia':                           // switch to IA mode
    tmp_taken = 1;
    if (PSR.ic == 0 || PSR.dt == 0 || PSR.mc == 1 || PSR.it == 0)
        undefined_behavior();
    if (qp != 0)
        illegal_operation_fault();
    if (AR[BSPSTORE] != AR[BSP])
        illegal_operation_fault();
    if (PSR.di)
        disabled_instruction_set_transition_fault();
    PSR.is = 1;                      // set IA-32 Instruction Set Mode
    CFM.sof = 0;                     //force current stack frame
    CFM.sol = 0;                     //to zero
    CFM.sor = 0;
    CFM.rrb.gr = 0;
    CFM.rrb.fr = 0;
    CFM.rrb.pr = 0;
    rse_invalidate_non_current_regs();
//compute effective instruction pointer
    EIP{31:0} = tmp_IP{31:0} - AR[CSD].Base;

// Note the register stack is disabled during IA-32 instruction
```



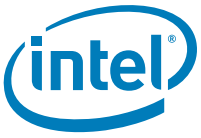
```
// set execution
    break;

case 'cloop':                           // simple counted loop
    if (slot != 2)
        illegal_operation_fault();
    tmp_taken = (AR[LC] != 0);
    if (AR[LC] != 0)
        AR[LC]--;
    break;

case 'ctop':
case 'cexit':                           // SW pipelined counted loop
    if (slot != 2)
        illegal_operation_fault();
    if (btype == 'ctop') tmp_taken =  ((AR[LC] != 0) || (AR[EC] u> 1));
    if (btype == 'cexit')tmp_taken = !((AR[LC] != 0) || (AR[EC] u> 1));
    if (AR[LC] != 0) {
        AR[LC]--;
        AR[EC] = AR[EC];
        PR[63] = 1;
        rotate_regs();
    } else if (AR[EC] != 0) {
        AR[LC] = AR[LC];
        AR[EC]--;
        PR[63] = 0;
        rotate_regs();
    } else {
        AR[LC] = AR[LC];
        AR[EC] = AR[EC];
        PR[63] = 0;
        CFM.rrb.gr = CFM.rrb.gr;
        CFM.rrb.fr = CFM.rrb.fr;
        CFM.rrb.pr = CFM.rrb.pr;
    }
    break;

case 'wtop':
case 'wexit':                           // SW pipelined while loop
    if (slot != 2)
        illegal_operation_fault();
    if (btype == 'wtop') tmp_taken =  (PR[qp] || (AR[EC] u> 1));
    if (btype == 'wexit')tmp_taken = !(PR[qp] || (AR[EC] u> 1));
    if (PR[qp]) {
        AR[EC] = AR[EC];
        PR[63] = 0;
        rotate_regs();
    } else if (AR[EC] != 0) {
        AR[EC]--;
        PR[63] = 0;
        rotate_regs();
    } else {
        AR[EC] = AR[EC];
        PR[63] = 0;
        CFM.rrb.gr = CFM.rrb.gr;
        CFM.rrb.fr = CFM.rrb.fr;
```



```
            CFM.rrb.pr = CFM.rrb.pr;
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                            // set the new value for IP
    if (!impl_uia_fault_supported() &&
        ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
         || (!PSR.it && unimplemented_physical_address(tmp_IP))))
        unimplemented_instruction_address_trap(lower_priv_transition,
                                               tmp_IP);
    if (lower_priv_transition && PSR.lp)
        lower_privilege_transfer_trap();
    if (PSR.tb)
        taken_branch_trap();
}
```

Interruptions: Illegal Operation fault
Disabled Instruction Set Transition fault
Unimplemented Instruction Address trap
Lower-Privilege Transfer trap
Taken Branch trap

Additional Faults on IA-32 target instructions:
IA_32_Exception(GPFault)
Disabled FP Reg Fault if PSR.dfh is 1



V3-B brl — Branch Long

Format:

(*qp*) *brl.btype.bwh.ph.dh* $target_{64}$		X3
(*qp*) *brl.btype.bwh.ph.dh* $b_1 = target_{64}$	call_form	X4
brl.ph.dh $target_{64}$	pseudo-op	

Format:

Description: A branch condition is evaluated, and either a branch is taken, or execution continues with the next sequential instruction. The execution of a branch logically follows the execution of all previous non-branch instructions in the same instruction group. On a taken branch, execution begins at slot 0.

Long branches are always IP-relative. The $target_{64}$ operand, in assembly, specifies a label to branch to. This is encoded in the long branch instruction as an immediate displacement (imm_{60}) between the target bundle and the bundle containing this instruction (imm_{60} = $target_{64}$ - IP >> 4). The L slot of the bundle contains 39 bits of imm_{60}.

Table 2-10. Long Branch Types

btype	Function	Branch Condition	Target Address
cond or *none*	Conditional branch	Qualifying predicate	IP-relative
call	Conditional procedure call	Qualifying predicate	IP-relative

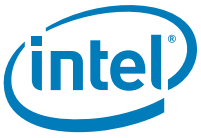
There is a pseudo-op for long unconditional branches, encoded like a conditional branch (*btype* = cond), with the *qp* field specifying PR 0, and with the *bwh* hint of sptk.

The branch type determines how the branch condition is calculated and whether the branch has other effects (such as writing a link register). For all long branch types, the branch condition is simply the value of the specified predicate register:

- **cond:** If the qualifying predicate is 1, the branch is taken. Otherwise it is not taken.
- **call:** If the qualifying predicate is 1, the branch is taken and several other actions occur:
  - The current values of the Current Frame Marker (CFM), the EC application register and the current privilege level are saved in the Previous Function State application register.
  - The caller's stack frame is effectively saved and the callee is provided with a frame containing only the caller's output region.
  - The rotation rename base registers in the CFM are reset to 0.
  - A return link value is placed in BR b_1.
  - The values in the DAHRs (if implemented) are pushed onto the DAHS, and the DAHRs revert to default values.

Read after Write (RAW) and Write after Read (WAR) dependency requirements for long branch instructions are slightly different than for other instructions but are the same as for branch instructions. See page 3:26 for details.

This instruction must be immediately followed by a stop; otherwise its behavior is undefined.



Values for various branch hint completers are the same as for branch instructions. Whether Prediction Strategy hints are shown in Table 2-7 on page 3:27, Sequential Prefetch hints are shown in Table 2-8 on page 3:27, and Branch Cache Deallocation hints are shown in Table 2-9 on page 3:27. See Section 4.5.2, “Branch Prediction Hints” on page 1:78.

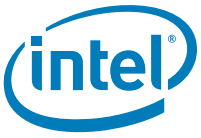
This instruction is not implemented on the Itanium processor, which takes an Illegal Operation fault whenever a long branch instruction is encountered, regardless of whether the branch is taken or not. To support the Itanium processor, the operating system is required to provide an Illegal Operation fault handler which emulates taken and not-taken long branches. Presence of this instruction is indicated by a 1 in the lb bit of CPUID register 4. See Section 3.1.11, “Processor Identification Registers” on page 1:34.

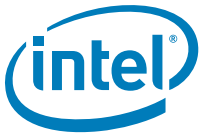
Operation:

```
tmp_IP = IP + (imm60 << 4);                  // determine branch target
if (!followed_by_stop())
    undefined_behavior();
if (!instruction_implemented(BRL))
    illegal_operation_fault();

switch (btype) {
    case 'cond':                             // simple conditional branch
        tmp_taken = PR[qp];
        break;

    case 'call':                             // call saves a return link
        tmp_taken = PR[qp];
        if (tmp_taken) {
            BR[b1] = IP + 16;

            AR[PFS].pfm = CFM;               // ... and saves the stack frame
            AR[PFS].pec = AR[EC];
            AR[PFS].ppl = PSR.cpl;

            alat_frame_update(CFM.sol, 0);
            rse_preserve_frame(CFM.sol);
            CFM.sof -= CFM.sol;              // new frame size is size of outs
            CFM.sol = 0;
            CFM.sor = 0;
            CFM.rrb.gr = 0;
            CFM.rrb.fr = 0;
            CFM.rrb.pr = 0;

            dahr_push_onto_dahs();
        }
        break;
}
if (tmp_taken) {
    taken_branch = 1;
    IP = tmp_IP;                             // set the new value for IP
    if (!impl_uia_fault_supported() &&
        ((PSR.it && unimplemented_virtual_address(tmp_IP, PSR.vm))
         || (!PSR.it && unimplemented_physical_address(tmp_IP))))
        unimplemented_instruction_address_trap(0,tmp_IP);
    if (PSR.tb)
```

```
        taken_branch_trap();
}
```

Interruptions: Illegal Operation fault
Taken Branch trap
Unimplemented Instruction Address trap





V3-C mov — Move Data Access Hint Register

mov — Move Data Access Hint Register

Format: (*qp*) *mov* $dahr_3 = imm_{16}$ M50

Description: The source operand is copied to the destination register.

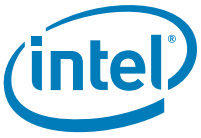
The value in imm_{16} is placed in DAHR $dahr_3$. In processors that do not implement DAHRs, this instruction executes as a nop.

Accesses of the DAHRs are always implicitly serialized. While implicitly serialized, read-after-write and write-after-write dependency violations should be avoided (e.g., setting a DAHR, followed by `br.call` in the same instruction group).

Operation:

```
if (PR[qp]) {
    DAHR[dahr3] = ignored_field_mask(DAHR_TYPE, dahr3, imm16);
}
```

Interruptions: None



V3-D hint — Performance Hint

Format:

(*qp*) hint imm_{21}	pseudo-op	
(*qp*) hint.i imm_{21}	i_unit_form	I18
(*qp*) hint.b imm_{21}	b_unit_form	B9
(*qp*) hint.m imm_{19}	m_unit_form	M49
(*qp*) hint.f imm_{21}	f_unit_form	F16
(*qp*) hint.x imm_{62}	x_unit_form	X5

Description: Provides a performance hint to the processor about the program being executed. It has no effect on architectural machine state, and operates as a `nop` instruction except for its performance effects.

The immediate, imm_{19}, imm_{21} or imm_{62}, specifies the hint. For the x_unit_form, the L slot of the bundle contains the upper 41 bits of imm_{62}.

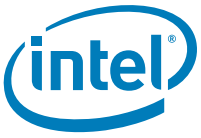
This instruction has five forms, each of which can be executed only on a particular execution unit type. The pseudo-op can be used if the unit type to execute on is unimportant.

Table 2-31. Hint Immediates

imm_{19}, imm_{21} or imm_{62}	Mnemonic	Hint
0x0	@pause	Indicates to the processor that the currently executing stream is waiting, spinning, or performing low priority tasks. This hint can be used by the processor to allocate more resources or time to another executing stream on the same processor. For the case where the currently executing stream is spinning or otherwise waiting for a particular address in memory to change, an advanced load to that address should be done before executing a `hint @pause`; this hint can be used by the processor to resume normal allocation of resources or time to the currently executing stream at the point when some other stream stores to that address.
0x1	@priority	Indicates to the processor that the currently executing stream is performing a high priority task. This hint can be used by the processor to allocate more resources or time to this stream. Implementations will ensure that such increased allocation is only temporary, and that repeated use of this hint will not impair longer-term fairness of allocation.
0x02-0x3f		These values are available for future architected extensions and will execute as a `nop` on all current processors. Use of these values may cause unexpected performance issues on future processors and should not be used.
other		Implementation specific. Performs an implementation-specific hint action. Consult processor model-specific documentation for details.

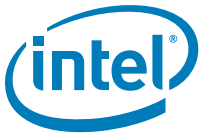
Operation:

```
if (PR[qp]) {
    if (x_unit_form)
        hint = imm62;
    if (m_unit_form)
        hint =imm19;
    else // i_unit_form || b_unit_form || f_unit_form
        hint = imm21;
```



```
        if (is_supported_hint(hint))
            execute_hint(hint);
    }
```

Interruptions: None



V3-E itc — Insert Translation Cache

itc — Insert Translation Cache

Format:

(*qp*) *itc.i* r_2	instruction_form	M41
(*qp*) *itc.d* r_2	data_form	M41

Description: An entry is inserted into the instruction or data translation cache. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}. The processor determines which entry to replace based on an implementation-specific replacement algorithm.

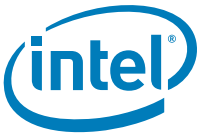
The visibility of the `itc` instruction to externally generated purges (`ptc.g`, `ptc.ga`) must occur before subsequent memory operations. From a software perspective, this is similar to acquire semantics. Serialization is still required to observe the side-effects of a translation being present.

`itc` must be the last instruction in an instruction group; otherwise, its behavior (including its ordering semantics) is undefined.

The TLB is first purged of any overlapping entries as specified by Table 4-1 on page 2:52.

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

To ensure forward progress, software must ensure that PSR.ic remains 0 until `rfi`-ing to the instruction that requires the translation. If psr.ic is to be set to 1 after the itc instruction with an ssm or mov-to-psr instruction, a srlz.i instruction is required between the itc and the instruction that sets psr.ic. This srlz.i instruction must be in a separate instruction group from the one containing the itc, and must be in a separate instruction group from the one containing the instruction that sets psr.ic.



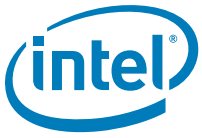
Operation:

```
if (PR[qp]) {
    if (!followed_by_stop())
        undefined_behavior();
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r2].nat)
        register_nat_consumption_fault(0);

    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!impl_check_mov_ifa() &&
            unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(ITC_TYPE);
        tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    } else {                                          // data_form
        tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
        tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
        slot = tlb_replacement_algorithm(DTC_TYPE);
        tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TC);
    }
}
```

Interruptions:

Machine Check abort
Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault
Virtualization fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.



V3-F itr - Insert Translation Register

itr — Insert Translation Register

Format:

(*qp*) *itr.i itr*[r_3] = r_2 — instruction_form — M42

(*qp*) *itr.d dtr*[r_3] = r_2 — data_form — M42

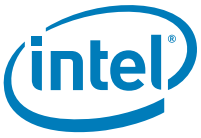
Description: A translation is inserted into the instruction or data translation register specified by the contents of GR r_3. GR r_2 specifies the physical address portion of the translation. ITIR specifies the protection key, page size and additional information. The virtual address is specified by the IFA register and the region register is selected by IFA{63:61}.

As described in Table 4-1, “Purge Behavior of TLB Inserts and Purges” on page 2:52, the TLB is first purged of any entries that overlap with the newly inserted translation. The translation previously contained in the TR slot specified by GR r_3 is not necessarily purged from the processor's TLBs and may remain as a TC entry. To ensure that the previous TR translation is purged, software must use explicit `ptr` instructions before inserting the new TR entry.

This instruction can only be executed at the most privileged level, and when PSR.ic and PSR.vm are both 0.

Operation:

```
if (PR[qp]) {
    if (PSR.ic)
        illegal_operation_fault();
    if (PSR.cpl != 0)
        privileged_operation_fault(0);
    if (GR[r3].nat || GR[r2].nat)
        register_nat_consumption_fault(0);

    slot = GR[r3]{7:0};
    tmp_size = CR[ITIR].ps;
    tmp_va = CR[IFA]{60:0};
    tmp_rid = RR[CR[IFA]{63:61}].rid;
    tmp_va = align_to_size_boundary(tmp_va, tmp_size);

    tmp_tr_type = instruction_form ? ITR_TYPE : DTR_TYPE;

    if (is_reserved_reg(tmp_tr_type, slot))
        reserved_register_field_fault();
    if (is_reserved_field(TLB_TYPE, GR[r2], CR[ITIR]))
        reserved_register_field_fault();
    if (!impl_check_mov_ifa() &&
            unimplemented_virtual_address(CR[IFA], PSR.vm))
        unimplemented_data_address_fault(0);
    if (PSR.vm == 1)
        virtualization_fault();

    if (instruction_form) {
        tlb_must_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
```

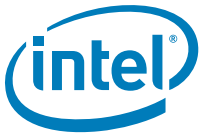


```
            tlb_may_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
            tlb_insert_inst(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
        } else {                                          // data_form
            tlb_must_purge_dtc_entries(tmp_rid, tmp_va, tmp_size);
            tlb_may_purge_itc_entries(tmp_rid, tmp_va, tmp_size);
            tlb_insert_data(slot, GR[r2], CR[ITIR], CR[IFA], tmp_rid, TR);
        }
    }
```

Interruptions: Machine Check abort
Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Unimplemented Data Address fault
Virtualization fault

Serialization: For the instruction_form, software must issue an instruction serialization operation before a dependent instruction fetch access. For the data_form, software must issue a data serialization operation before issuing a data access or non-access reference dependent on the new translation.

Notes: The processor may use invalid translation registers for translation cache entries. Performance can be improved on some processor models by ensuring translation registers are allocated beginning at translation register zero and continuing contiguously upwards. *If, after execution of the itr instruction, PSR.ic is to be set to a 1 with an ssm or mov-to-psr instruction, software must execute a srlz.i instruction before setting PSR.ic=1. If a srlz.i is used, it must be in a separate instruction group from the itr instruction, and must be in a separate instruction group from the instruction that sets psr.ic to 1.*



V3-G ld — Load

ld — Load

Format:

(*qp*) ld*sz*.*ldtype*.*ldhintx* r_1 = [r_3]	no_base_update_form	M1
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], r_2	reg_base_update_form	M2
(*qp*) ld*sz*.*ldtype*.*ldhint* r_1 = [r_3], imm_9	imm_base_update_form	M3
(*qp*) ld16.*ldhintx* r_1, ar.csd = [r_3]	sixteen_byte_form, no_base_update_form	M1
(*qp*) ld16.acq.*ldhintx* r_1, ar.csd = [r_3]	sixteen_byte_form, acquire_form, no_base_update_form	M1
(*qp*) ld8.fill.*ldhintx* r_1 = [r_3]	fill_form, no_base_update_form	M1
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], r_2	fill_form, reg_base_update_form	M2
(*qp*) ld8.fill.*ldhint* r_1 = [r_3], imm_9	fill_form, imm_base_update_form	M3

Description: A value consisting of *sz* bytes is read from memory starting at the address specified by the value in GR r_3. The value is then zero extended and placed in GR r_1. The values of the *sz* completer are given in Table 2-32. The NaT bit corresponding to GR r_1 is cleared, except as described below for speculative loads. The *ldtype* completer specifies special load operations, which are described in Table 2-33.

For the sixteen_byte_form, two 8-byte values are loaded as a single, 16-byte memory read. The value at the lowest address is placed in GR r_1, and the value at the highest address is placed in the Compare and Store Data application register (AR[CSD]). The only load types supported for this sixteen_byte_form are *none* and *acq*.

For the fill_form, an 8-byte value is loaded, and a bit in the UNAT application register is copied into the target register NaT bit. This instruction is used for reloading a spilled register/NaT pair. See Section 4.4.4, “Control Speculation” on page 1:60 for details.

In the base update forms, the value in GR r_3 is added to either a signed immediate value (imm_9) or a value from GR r_2, and the result is placed back in GR r_3. This base register update is done after the load, and does not affect the load address. In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set and no fault is raised. Base register update is not supported for the `ld16` instruction.

Table 2-32. *sz* Completers

sz Completer	Bytes Accessed
1	1 byte
2	2 bytes
4	4 bytes
8	8 bytes

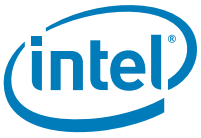


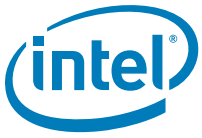
Table 2-33. Load Types

ldtype Completer	Interpretation	Special Load Operation
none	Normal load	
s	Speculative load	Certain exceptions may be deferred rather than generating a fault. Deferral causes the target register's NaT bit to be set. The NaT bit is later used to detect deferral.
a	Advanced load	An entry is added to the ALAT. This allows later instructions to check for colliding stores. If the referenced data page has a non-speculative attribute, the target register and NaT bit is cleared, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
sa	Speculative Advanced load	An entry is added to the ALAT, and certain exceptions may be deferred. Deferral causes the target register's NaT bit to be set, and the processor ensures that no ALAT entry exists for the target register. The absence of an ALAT entry is later used to detect deferral or collision.
c.nc	Check load – no clear	The ALAT is searched for a matching entry. If found, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a load is performed, and an entry is added to the ALAT (unless the referenced data page has a non-speculative attribute, in which case no ALAT entry is allocated).
c.clr	Check load – clear	The ALAT is searched for a matching entry. If found, the entry is removed, no load is done and the target register is unchanged. Regardless of ALAT hit or miss, base register updates are performed, if specified. An implementation may optionally cause the ALAT lookup to fail independent of whether an ALAT entry matches. If not found, a clear check load behaves like a normal load.
c.clr.acq	Ordered check load – clear	This type behaves the same as the unordered clear form, except that the ALAT lookup (and resulting load, if no ALAT entry is found) is performed with acquire semantics.
acq	Ordered load	An ordered load is performed with acquire semantics.
bias	Biased load	A hint is provided to the implementation to acquire exclusive ownership of the accessed cache line.

For more details on ordered, biased, speculative, advanced and check loads see Section 4.4.4, "Control Speculation" on page 1:60 and Section 4.4.5, "Data Speculation" on page 1:63. For more details on ordered loads see Section 4.4.7, "Memory Access Ordering" on page 1:73. See Section 4.4.6, "Memory Hierarchy Control and Consistency" on page 1:69 for details on biased loads. Details on memory attributes are described in Section 4.4, "Memory Attributes" on page 2:75.

For the non-speculative load types, if NaT bit associated with GR r_3 is 1, a Register NaT Consumption fault is taken. For speculative and speculative advanced loads, no fault is raised, and the exception is deferred. For the base-update calculation, if the NaT bit associated with GR r_2 is 1, the NaT bit associated with GR r_3 is set to 1 and no fault is raised.

The value of the *ldhint* completer specifies the locality of the memory access. The values of the *ldhint* completer are given in Table 2-34. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *ldhint*. Prefetch and locality hints do not affect program functionality and may be



ignored by the implementation. See Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69 for details.

Table 2-34. Load Hints for the no-base-update forms

ldhintx Completer	Interpretation
none or d0	Temporal locality, level 1
nt1 or d1	No temporal locality, level 1
d2	Temporal locality, level 1
nta or d3	No temporal locality, all levels
d4	Hint d4
d5	Hint d5
d6	Hint d6
d7	Hint d7

Table 2-35. Load Hints for the base-update forms

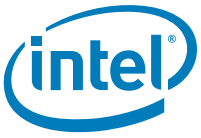
ldhint Completer	Interpretation
none or d0	Temporal locality, level 1
nt1 or d1	No temporal locality, level 1
d2	Temporal locality, level 1
nta or d3	No temporal locality, all levels

In the no_base_update form, the value in GR r_3 is not modified and no prefetch hint is implied.

For the base update forms, specifying the same register address in r_1 and r_3 will cause an Illegal Operation fault.

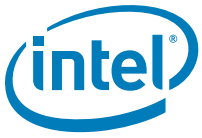
Hardware support for `ld16` instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `ld16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen_byte_form, 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 Section 3.1.11, “Processor Identification Registers” on page 1:34 for details.

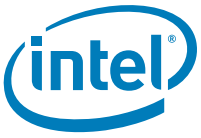


Operation:

```
if (PR[qp]) {
    size = fill_form ? 8 : (sixteen_byte_form ? 16 : sz);

    speculative = (ldtype == 's' || ldtype == 'sa');
    advanced = (ldtype == 'a' || ldtype == 'sa');
    check_clear = (ldtype == 'c.clr' || ldtype == 'c.clr.acq');
    check_no_clear = (ldtype == 'c.nc');
    check = check_clear || check_no_clear;
    acquire = (acquire_form || ldtype == 'acq' || ldtype == 'c.clr.acq');
    otype = acquire ? ACQUIRE : UNORDERED;
    bias = (ldtype == 'bias') ? BIAS : 0 ;
    translate_address = 1;
    read_memory = 1;

    itype = READ;
    if (speculative) itype |= SPEC ;
    if (advanced) itype |= ADVANCE ;
    if (size == 16) itype |= UNCACHE_OPT ;

    if (sixteen_byte_form && !instruction_implemented(LD16))
        illegal_operation_fault();
    if ((reg_base_update_form || imm_base_update_form) && (r1 == r3))
        illegal_operation_fault();
    check_target_register(r1);
    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (reg_base_update_form) {
        tmp_r2 = GR[r2];
        tmp_r2nat = GR[r2].nat;
    }

    if (!speculative && GR[r3].nat)              // fault on NaT address
        register_nat_consumption_fault(itype);
    defer = speculative && (GR[r3].nat || PSR.ed);// defer exception if spec

    if (check && alat_cmp(GENERAL, r1)) {
        translate_address = alat_translate_address_on_hit(ldtype, GENERAL,
r1);
        read_memory = alat_read_memory_on_hit(ldtype, GENERAL, r1);
    }
    if (!translate_address) {
        if (check_clear || advanced)             // remove any old alat entry
            alat_inval_single_entry(GENERAL, r1);
    } else {
        if (!defer) {
            paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                                  &defer);
            spontaneous_deferral(paddr, size, UM.be, mattr, otype,
                                  bias | ldhint, &defer);
            if (!defer && read_memory) {
                if (size == 16) {
                    mem_read_pair(&val, &val_ar, paddr, size, UM.be, mattr,
                                  otype, ldhint);
                }
                else {
```

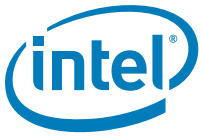


```
                    val = mem_read(paddr, size, UM.be, mattr, otype,
                                   bias | ldhint);
                }
            }
        }
        if (check_clear || advanced)              // remove any old ALAT entry
            alat_inval_single_entry(GENERAL, r1);
        if (defer) {
            if (speculative) {
                GR[r1] = natd_gr_read(paddr, size, UM.be, mattr, otype,
                                      bias | ldhint);
                GR[r1].nat = 1;
            } else {
                GR[r1] = 0;                     // ld.a to sequential memory
                GR[r1].nat = 0;
            }
        } else {                                  // execute load normally
            if (fill_form) {                      // fill NaT on ld8.fill
                bit_pos = GR[r3]{8:3};
                GR[r1] = val;
                GR[r1].nat = AR[UNAT]{bit_pos};
            } else {                              // clear NaT on other types
                if (size == 16) {
                    GR[r1] = val;
                    AR[CSD] = val_ar;
                }
                else {
                    GR[r1] = zero_ext(val, size * 8);
                }
                GR[r1].nat = 0;
            }
            if ((check_no_clear || advanced) && ma_is_speculative(mattr))
                                                  // add entry to ALAT
                alat_write(ldtype, GENERAL, r1, paddr, size);
        }
    }

    if (imm_base_update_form) {                   // update base register
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + tmp_r2;
        GR[r3].nat = GR[r3].nat || tmp_r2nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], ldhint | bias, itype);
}
```



Interruptions:

Illegal Operation fault
Register NaT Consumption fault
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 Access Bit fault
Data Debug fault
Unaligned Data Reference fault
Unsupported Data Reference fault



V3-H lfetch — Line Prefetch

lfetch — Line Prefetch

Format:

(*qp*) lfetch.*lfhint* [r_3]	no_base_update_form	M51
(*qp*) lfetch.fault.*lfhint* [r_3]	no_base_update_form, fault_form	M13
(*qp*) lfetch.*lftype*.*lfhint* [r_3], r_2	reg_base_update_form	M14
(*qp*) lfetch.*lftype*.*lfhint* [r_3], imm_9	imm_base_update_form	M15
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3]	no_base_update_form, exclusive_form	M13
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], r_2	reg_base_update_form, exclusive_form	M14
(*qp*) lfetch.*lftype*.excl.*lfhint* [r_3], imm_9	imm_base_update_form, exclusive_form	M15
(*qp*) lfetch.count.*lfhint* [r_3], cnt_6, $stride_5$	counted_form	M52

Description: The line containing the address specified by the value in GR r_3 is moved to the highest level of the data memory hierarchy. The value of the *lfhint* modifier specifies the locality of the memory access; see Section 4.4, “Memory Access Instructions” on page 1:59 for details. The mnemonic values of *lfhint* are given in Table 2-39.

The behavior of the memory read is also determined by the memory attribute associated with the accessed page. See Chapter 4, “Addressing and Protection” in Volume 2. Line size is implementation dependent but must be a power of two greater than or equal to 32 bytes. In the exclusive form, the cache line is allowed to be marked in an exclusive state. This qualifier is used when the program expects soon to modify a location in that line. If the memory attribute for the page containing the line is not cacheable, then no reference is made.

The completer, *lftype*, specifies whether or not the instruction raises faults normally associated with a regular load. Table 2-38 defines these two options.

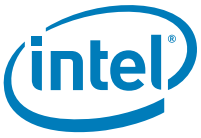
Table 2-38. *lftype* Mnemonic Values

lftype Mnemonic	Interpretation
none	No faults are raised
fault	Raise faults

In the base update forms, after being used to address memory, the value in GR r_3 is incremented by either the sign-extended value in imm_9 (in the imm_base_update_form) or the value in GR r_2 (in the reg_base_update_form). In the reg_base_update_form, if the NaT bit corresponding to GR r_2 is set, then the NaT bit corresponding to GR r_3 is set – no fault is raised.

In the reg_base_update_form and the imm_base_update_form, if the NaT bit corresponding to GR r_3 is clear, then the address specified by the value in GR r_3 after the post-increment acts as a hint to implicitly prefetch the indicated cache line. This implicit prefetch uses the locality hints specified by *lfhint*. The implicit prefetch does not affect program functionality, does not raise any faults, and may be ignored by the implementation.

In the no_base_update_form, the value in GR r_3 is not modified and no implicit prefetch hint is implied.



If the NaT bit corresponding to GR r_3 is set then the state of memory is not affected. In the reg_base_update_form and imm_base_update_form, the post increment of GR r_3 is performed and prefetch is hinted as described above.

`lfetch` instructions, like hardware prefetches, are not orderable operations, i.e., they have no order with respect to prior or subsequent memory operations.

Table 2-39. *lfhint* Mnemonic Values

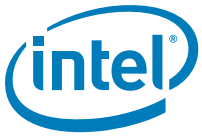
lfhint Mnemonic	Interpretation
none or d0	Temporal locality, level 1
nt1 or d1	No temporal locality, level 1
nt2 or d2	No temporal locality, level 2
nta or d3	No temporal locality, all levels
d4	Hint d4
d5	Hint d5
d6	Hint d6
d7	Hint d7

A faulting `lfetch` to an unimplemented address results in an Unimplemented Data Address fault. A non-faulting `lfetch` to an unimplemented address does not take the fault and will not issue a prefetch request, but, if specified, will perform a register post-increment.

Both the non-faulting and the faulting forms of lfetch can be used speculatively. The purpose of raising faults on the faulting form is to allow the operating system to resolve problems with the address to the extent that it can do so relatively quickly. If problems with the address cannot be resolved quickly, the OS simply returns to the program, and forces the data prefetch to be skipped over.

Specifically, if a faulting `lfetch` takes any of the listed faults (other than Illegal Operation fault), the operating system must handle this fault to the extent that it can do so relatively quickly and invisibly to the interrupted program. If the fault cannot be handled quickly or cannot be handled invisibly (e.g., if handling the fault would involve terminating the program), the OS must return to the interrupted program, skipping over the data prefetch. This can easily be done by setting the IPSR.ed bit to 1 before executing an `rfi` to go back to the process, which will allow the `lfetch.fault` to perform its base register post-increment (if specified), but will suppress any prefetch request and hence any prefetch-related fault. Note that the OS can easily identify that a faulting `lfetch` was the cause of the fault by observing that ISR.na is 1, and ISR.code{3:0} is 4. The one exception to this is the Illegal Operation fault, which can be caused by an `lfetch.fault` if base register post-increment is specified, and the base register is outside of the current stack frame, or is GR0. Since this one fault is not related to the prefetch aspect of `lfetch.fault`, but rather to the base update portion, Illegal Operation faults on `lfetch.fault` should be handled the same as for any other instruction.

In the counted_form, multiple prefetch operations may optionally be generated. The cnt_6 operand specifies the number of prefetches to be done, and can be any number in the range 1 to 64; it is encoded as cnt_6-1 in the instruction. The $stride_5$ operand specifies the offset to be added to the address to generate each subsequent prefetch. The $stride_5$ operand must be a multiple of 64, and can be any number in the range -16*64, -15*64, ... -1*64, 0, 1*64, ... 15*64; it is encoded as $stride_5$/64 in the instruction. If cnt_6 is 1, the $stride_5$ operands is ignored. In implementations that do not implement counted prefetch, this instruction behaves the same as an `lfetch` with *lftype* of none.

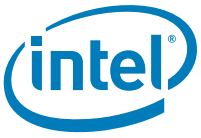


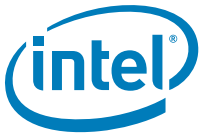
Operation:

```
if (PR[qp]) {
    itype = READ|NON_ACCESS;
    itype |= (lftype == 'fault' || fault_form) ? LFETCH_FAULT : LFETCH;

    if (reg_base_update_form || imm_base_update_form)
        check_target_register(r3);

    if (itype & LFETCH_FAULT) {        // faulting form
        if (GR[r3].nat && !PSR.ed)     // fault on NaT address
            register_nat_consumption_fault(itype);
    }

    excl_hint = (exclusive_form) ? EXCLUSIVE : 0;

    if (!GR[r3].nat && !PSR.ed) {// faulting form already faulted if r3 is nat
        paddr = tlb_translate(GR[r3], 1, itype, PSR.cpl, &mattr, &defer);
        if (!defer)
            mem_promote(paddr, mattr, lfhint | excl_hint);
    }

    if (counted_form && impl_lfetch_count()) {
        for (vaddr=GR[r3]+stride5, count=cnt6-1; count>0; count--) {
            paddr = tlb_translate(vaddr, 1, itype, PSR.cpl, &mattr, &defer);
            if (!defer)
                mem_promote(paddr, mattr, lfhint | excl_hint);
        }
    }

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = GR[r3].nat;
    } else if (reg_base_update_form) {
        GR[r3] = GR[r3] + GR[r2];
        GR[r3].nat = GR[r2].nat || GR[r3].nat;
    }

    if ((reg_base_update_form || imm_base_update_form) && !GR[r3].nat)
        mem_implicit_prefetch(GR[r3], lfhint | excl_hint, itype);
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
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 Access Bit fault
Data Debug fault





V3-I mov — Move Indirect Register

mov — Move Indirect Register

Format:

(*qp*) mov r_1 = *ireg*[r_3] — from_form M43
(*qp*) mov *ireg*[r_3] = r_2 — to_form M42

Description: The source operand is copied to the destination register.

For move from indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg* (see Table 2-41 below). The indexed register is read and its value is copied into GR r_1.

For move to indirect register, GR r_3 is read and the value used as an index into the register file specified by *ireg*. GR r_2 is read and its value copied into the indexed register.

Table 2-41. Indirect Register File Mnemonics

ireg	Register File
cpuid	Processor Identification Register
dahr	Data Access Hint Register
dbr	Data Breakpoint Register
ibr	Instruction Breakpoint Register
pkr	Protection Key Register
pmc	Performance Monitor Configuration Register
pmd	Performance Monitor Data Register
rr	Region Register

For all register files other than the region registers, bits {7:0} of GR r_3 are used as the index. For region registers, bits {63:61} are used. The remainder of the bits are ignored. Access to machine specific registers is implementation dependent, all bits may be used as a index.

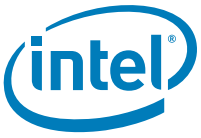
Instruction and data breakpoint, performance monitor configuration, protection key, machine specific, and region registers can only be accessed at the most privileged level. Performance monitor data registers can only be written at the most privileged level.

The CPU identification registers can only be read. There is no to_form of this instruction.

The DAHR registers can only be read with this instruction. See "mov — Move Data Access Hint Register" on page 3:180 instruction page for how to write these registers. DAHRs are unprivileged.

For move to protection key register, the processor ensures uniqueness of protection keys by checking new valid protection keys against all protection key registers. If any matching keys are found, duplicate protection keys are invalidated.

Apart from the PMC and PMD register files, access of a non-existent register results in a Reserved Register/Field fault. All accesses to the implementation-dependent portion of



PMC and PMD register files result in implementation dependent behavior but do not fault.

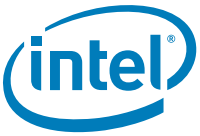
Modifying a region register or a protection key register which is being used to translate:

- the executing instruction stream when PSR.it == 1, or
- the data space for an eager RSE reference when PSR.rt == 1

is an undefined operation.

Operation:

```
if (PR[qp]) {
    if (ireg == RR_TYPE)
        tmp_index = GR[r3]{63:61};
    else // all other register types
        tmp_index = GR[r3]{7:0};

    if (from_form) {
        if (ireg == DAHR_TYPE && !instruction_implemented(DAHR))
            illegal_operation_fault();
        check_target_register(r1);

        if (PSR.cpl != 0 && !(ireg == PMD_TYPE || ireg == CPUID_TYPE || ireg ==
DAHR_TYPE))
            privileged_operation_fault(0);

        if (GR[r3].nat)
            register_nat_consumption_fault(0);

        if (is_reserved_reg(ireg, tmp_index))
            reserved_register_field_fault();

        if (PSR.vm == 1 && !(ireg == PMD_TYPE || ireg == DAHR_TYPE))
            virtualization_fault();

        if (ireg == PMD_TYPE) {
            if ((PSR.cpl != 0) && ((PSR.sp == 1) ||
                 (tmp_index > 3 &&
                 tmp_index <= IMPL_MAXGENERIC_PMCPMD &&
                 PMC[tmp_index].pm == 1)))
                GR[r1] = 0;
            else
                GR[r1] = pmd_read(tmp_index);
        } else
            switch (ireg) {
                case CPUID_TYPE:  GR[r1] = CPUID[tmp_index]; break;
                case DAHR_TYPE:   GR[r1] = DAHR[tmp_index]; break;
                case DBR_TYPE:    GR[r1] = DBR[tmp_index]; break;
                case IBR_TYPE:    GR[r1] = IBR[tmp_index]; break;
                case PKR_TYPE:    GR[r1] = PKR[tmp_index]; break;
                case PMC_TYPE:    GR[r1] = pmc_read(tmp_index); break;
                case RR_TYPE:     GR[r1] = RR[tmp_index]; break;
            }
        GR[r1].nat = 0;
    } else {          // to_form
        if (PSR.cpl != 0)
            privileged_operation_fault(0);
```



```
            if (GR[r2].nat || GR[r3].nat)
                register_nat_consumption_fault(0);

            if (is_reserved_reg(ireg, tmp_index)
                || ireg == CPUID_TYPE
                || ireg == DAHR_TYPE
                || is_reserved_field(ireg, tmp_index, GR[r2]))
                reserved_register_field_fault();
            if (PSR.vm == 1)
                virtualization_fault();
            if (ireg == PKR_TYPE && GR[r2]{0} == 1) { // writing valid prot key
                if ((tmp_slot = tlb_search_pkr(GR[r2]{31:8})) != NOT_FOUND)
                    PKR[tmp_slot].v = 0; // clear valid bit of matching key reg
            }
            tmp_val = ignored_field_mask(ireg, tmp_index, GR[r2]);
            switch (ireg) {
                case DBR_TYPE:    DBR[tmp_index] = tmp_val; break;
                case IBR_TYPE:    IBR[tmp_index] = tmp_val; break;
                case PKR_TYPE:    PKR[tmp_index] = tmp_val; break;
                case PMC_TYPE:    pmc_write(tmp_index, tmp_val); break;
                case PMD_TYPE:    pmd_write(tmp_index, tmp_val); break;
                case RR_TYPE:     RR[tmp_index]= tmp_val; break;
            }
        }
    }
```

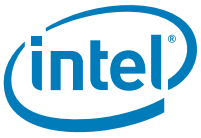
Interruptions: Illegal Operation fault
Privileged Operation fault
Register NaT Consumption fault
Reserved Register/Field fault
Virtualization fault

Serialization: For move to data breakpoint registers, software must issue a data serialize operation before issuing a memory reference dependent on the modified register.

For move to instruction breakpoint registers, software must issue an instruction serialize operation before fetching an instruction dependent on the modified register.

For move to protection key, region, performance monitor configuration, and performance monitor data registers, software must issue an instruction or data serialize operation to ensure the changes are observed before issuing any dependent instruction.

To obtain improved accuracy, software can issue an instruction or data serialize operation before reading the performance monitors.



V3-J st — Store

st — Store

Format:

(*qp*) *stsz.sttype.sthintx* [r_3] = r_2	normal_form, no_base_update_form	M4
(*qp*) *stsz.sttype.sthint* [r_3] = r_2, imm_9	normal_form, imm_base_update_form	M5
(*qp*) *st16.sttype.sthintx* [r_3] = r_2, ar.csd	sixteen_byte_form, no_base_update_form	M4
(*qp*) *st8.spill.sthintx* [r_3] = r_2	spill_form, no_base_update_form	M4
(*qp*) *st8.spill.sthint* [r_3] = r_2, imm_9	spill_form, imm_base_update_form	M5

Description: A value consisting of the least significant *sz* bytes of the value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The values of the *sz* completer are given in Table 2-32 on page 3:151. The *sttype* completer specifies special store operations, which are described in Table 2-51. If the NaT bit corresponding to GR r_3 is 1, or in sixteen_byte_form or normal_form, if the NaT bit corresponding to GR r_2 is 1, a Register NaT Consumption fault is taken.

In the sixteen_byte_form, two 8-byte values are stored as a single, 16-byte atomic memory write. The value in GR r_2 is written to memory starting at the address specified by the value in GR r_3. The value in the Compare and Store Data application register (AR[CSD]) is written to memory starting at the address specified by the value in GR r_3 plus 8.

In the spill_form, an 8-byte value is stored, and the NaT bit corresponding to GR r_2 is copied to a bit in the UNAT application register. This instruction is used for spilling a register/NaT pair. See Section 4.4.4, “Control Speculation” on page 1:60 for details.

In the imm_base_update form, the value in GR r_3 is added to a signed immediate value (imm_9) and the result is placed back in GR r_3. This base register update is done after the store, and does not affect the store address, nor the value stored (for the case where r_2 and r_3 specify the same register). Base register update is not supported for the `st16` instruction.

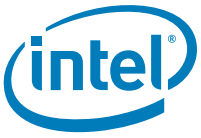
Table 2-51. Store Types

sttype Completer	Interpretation	Special Store Operation
none	Normal store	
rel	Ordered store	An ordered store is performed with release semantics.

For more details on ordered stores see Section 4.4.7, “Memory Access Ordering” on page 1:73.

The ALAT is queried using the physical memory address and the access size, and all overlapping entries are invalidated.

The value of the *sthint* completer specifies the locality of the memory access. The values of the *sthint* completer are given in Table 2-51. A prefetch hint is implied in the base update forms. The address specified by the value in GR r_3 after the base update acts as a hint to prefetch the indicated cache line. This prefetch uses the locality hints specified by *sthint*. See Section 4.4.6, “Memory Hierarchy Control and Consistency” on page 1:69.



Hardware support for `st16` instructions that reference a page that is neither a cacheable page with write-back policy nor a NaTPage is optional. On processor models that do not support such `st16` accesses, an Unsupported Data Reference fault is raised when an unsupported reference is attempted.

For the sixteen_byte_form, 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 Section 3.1.11, "Processor Identification Registers" on page 1:34 for details.

Table 2-52. Store Hints for the no-base-update forms

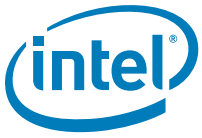
sthintx Completer	Interpretation
none or d0	Temporal locality, level 1
d1	Temporal locality, level 1
d2	Temporal locality, level 1
nta or d3	No temporal locality, all levels
d4	Hint d4
d5	Hint d5
d6	Hint d6
d7	Hint d7

Table 2-53. Store Hints for the base-update forms

sthint Completer	Interpretation
none or d0	Temporal locality, level 1
d1	Temporal locality, level 1
d2	Temporal locality, level 1
nta or d3	No temporal locality, all levels

Operation:

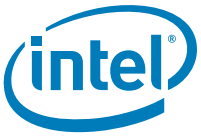
```
if (PR[qp]) {
    size = spill_form ? 8 : (sixteen_byte_form ? 16 : sz);
    itype = WRITE;
    if (size == 16) itype |= UNCACHE_OPT;
    otype = (sttype == 'rel') ? RELEASE : UNORDERED;

    if (sixteen_byte_form && !instruction_implemented(ST16))
        illegal_operation_fault();
    if (imm_base_update_form)
        check_target_register(r3);
    if (GR[r3].nat || ((sixteen_byte_form || normal_form) && GR[r2].nat))
        register_nat_consumption_fault(WRITE);

    paddr = tlb_translate(GR[r3], size, itype, PSR.cpl, &mattr,
                          &tmp_unused);
    if (spill_form && GR[r2].nat) {
        natd_gr_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    }
    else {
        if (sixteen_byte_form)
            mem_write16(GR[r2], AR[CSD], paddr, UM.be, mattr, otype, sthint);
        else
            mem_write(GR[r2], paddr, size, UM.be, mattr, otype, sthint);
    }
```

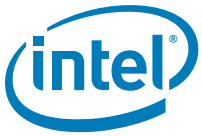


```
    if (spill_form) {
        bit_pos = GR[r3]{8:3};
        AR[UNAT]{bit_pos} = GR[r2].nat;
    }

    alat_inval_multiple_entries(paddr, size);

    if (imm_base_update_form) {
        GR[r3] = GR[r3] + sign_ext(imm9, 9);
        GR[r3].nat = 0;
        mem_implicit_prefetch(GR[r3], sthint, WRITE);
    }
}
```

Interruptions:

Illegal Operation fault
Register NaT Consumption fault
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





V3-K Chapter 3 Pseudo-Code Functions

Pseudo-Code Functions 3

This chapter contains a table of all pseudo-code functions used on the Itanium instruction pages.

Table 3-1. Pseudo-code Functions

Function	Operation
*xxx*_fault(parameters ...)	There are several fault functions. Each fault function accepts parameters specific to the fault, e.g., exception code values, virtual addresses, etc. If the fault is deferred for speculative load exceptions the fault function will return with a deferral indication. Otherwise, fault routines do not return and terminate the instruction sequence.
*xxx*_trap(parameters ...)	There are several trap functions. Each trap function accepts parameters specific to the trap, e.g., trap code values, virtual addresses, etc. Trap routines do not return.
acceptance_fence()	Ensures prior data memory references to uncached ordered-sequential memory pages are “accepted” before subsequent data memory references are performed by the processor.
alat_cmp(rtype, raddr)	Returns a one if the implementation finds an ALAT entry which matches the register type specified by `rtype` and the register address specified by `raddr`, else returns zero. This function is implementation specific. Note that an implementation may optionally choose to return zero (indicating no match) even if a matching entry exists in the ALAT. This provides implementation flexibility in designing fast ALAT lookup circuits.
alat_frame_update(delta_bof, delta_sof)	Notifies the ALAT of a change in the bottom of frame and/or size of frame. This allows management of the ALAT’s tag bits or other management functions it might need.
alat_inval()	Invalidate all entries in the ALAT.
alat_inval_multiple_entries(paddr, size)	The ALAT is queried using the physical memory address specified by `paddr` and the access size specified by `size`. All matching ALAT entries are invalidated. No value is returned.
alat_inval_single_entry(rtype, rega)	The ALAT is queried using the register type specified by `rtype` and the register address specified by `rega`. At most one matching ALAT entry is invalidated. No value is returned.
alat_read_memory_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should perform a memory access (requires prior address translation); returns a zero otherwise.
alat_translate_address_on_hit(ldtype, rtype, raddr)	Returns a one if the implementation requires that the requested check load should translate the source address and take associated faults; returns a zero otherwise.
alat_write(ldtype, rtype, raddr, paddr, size)	Allocates a new ALAT entry or updates an existing entry using the load type specified by `ldtype`, the register type specified by `rtype`, the register address specified by `raddr`, the physical memory address specified by `paddr`, and the access size specified by `size`. No value is returned. This function guarantees that at most only one ALAT entry exists for a given `raddr`. Based on the load type `ldtype`, if a `ld.c.nc`, `ldf.c.nc`, or `ldfp.c.nc` instruction's `raddr` matches an existing ALAT entry's register tag, but the instruction's `size` and/or `paddr` are different than that of the existing entry's, then this function may either preserve the existing entry, or invalidate it and write a new entry with the instruction's specified `size` and `paddr`.
align_to_size_boundary(vaddr, size)	Returns `vaddr` aligned to the boundary specified by `size`.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
branch_predict(wh, ih, ret, target, tag)	Implementation-dependent routine which updates the processor's branch prediction structures.
check_branch_implemented(check_type)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether a failing check instruction causes a branch (TRUE), or a Speculative Operation fault (FALSE). The result may be different for different types of check instructions: CHKS_GENERAL, CHKS_FLOAT, CHKA_GENERAL, CHKA_FLOAT. In addition, the result may depend on other implementation-dependent parameters.
check_probe_virtualization_fault(type, cpl)	If implemented, this function may raise virtualization faults for specific probe instructions. Please refer to the instruction page for probe instruction for details.
check_target_register(r1)	If the `r1` argument specifies an out-of-frame stacked register (as defined by CFM) or `r1` specifies GR0, an Illegal Operation fault is delivered, and this function does not return.
check_target_register_sof(r1, newsof)	If the `r1` argument specifies an out-of-frame stacked register (as defined by the `newsof` argument) or `r1` specifies GR0, an Illegal Operation fault is delivered and this function does not return.
concatenate2(x1, x2)	Concatenates the lower 32 bits of the 2 arguments, and returns the 64-bit result.
concatenate4(x1, x2, x3, x4)	Concatenates the lower 16 bits of the 4 arguments, and returns the 64-bit result.
concatenate8(x1, x2, x3, x4, x5, x6, x7, x8)	Concatenates the lower 8 bits of the 8 arguments, and returns the 64-bit result.
dahr_dahs_revert_to_default()	All DAHRs and all elements at all levels of the DAHS revert to default values.
dahr_pop_from_dahs()	The elements in the top stack level of the Data Access Hint Stack (DAHS) are copied into the DAHRs, and the elements in the stack are popped up one level, with the elements in the bottom stack level reverting to default values.
dahr_push_to_dahs()	The elements in the Data Access Hint Stack (DAHS) are pushed down one level (the elements in the bottom stack level are lost), the values in the DAHRs are copied into the elements in the top stack level, and then the DAHRs revert to default values.
data_serialize()	Ensures all prior register updates with side-effects are observed before subsequent execution and data memory references are performed.
deliver_unmasked_pending_interrupt()	This implementation-specific function checks whether any unmasked external interrupts are pending, and if so, transfers control to the external interrupt vector.
execute_hint(hint)	Executes the hint specified by `hint`.
fadd(fp_dp, fr2)	Adds a floating-point register value to the infinitely precise product and return the infinitely precise sum, ready for rounding.
fcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcmp` instruction.
fcvt_fx_exception_fault_check(fr2, signed_form, trunc_form, sf *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fcvt.fx`, `fcvt.fxu`, `fcvt.fx.trunc` and `fcvt.fxu.trunc` instructions. It propagates NaNs.
fma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fma` instruction. It propagates NaNs and special IEEE results.
fminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `famax`, `famin`, `fmax`, and `fmin` instructions.
fms_fnma_exception_fault_check(f2, f3, f4, pc, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fms` and `fnma` instructions. It propagates NaNs and special IEEE results.
fmul(fr3, fr4)	Performs an infinitely precise multiply of two floating-point register values.
followed_by_stop()	Returns TRUE if the current instruction is followed by a stop; otherwise, returns FALSE.
fp_check_target_register(f1)	If the specified floating-point register identifier is 0 or 1, this function causes an illegal operation fault.
fp_decode_fault(tmp_fp_env)	Returns floating-point exception fault code values for ISR.code.
fp_decode_traps(tmp_fp_env)	Returns floating-point trap code values for ISR.code.
fp_equal(fr1, fr2)	IEEE standard equality relationship test.

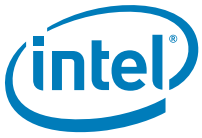


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
fp_fr_to_mem_format(freg, size)	Converts a floating-point value in register format to floating-point memory format. It assumes that the floating-point value in the register has been previously rounded to the correct precision which corresponds with the `size` parameter.
fp_ieee_recip(num, den)	Returns the true quotient for special sets of operands, or an approximation to the reciprocal of the divisor to be used in the software divide algorithm.
fp_ieee_recip_sqrt(root)	Returns the true square root result for special operands, or an approximation to the reciprocal square root to be used in the software square root algorithm.
fp_is_nan(freg)	Returns true when floating register contains a NaN.
fp_is_nan_or_inf(freg)	Returns true if the floating-point exception_fault_check functions returned a IEEE fault disabled default result or a propagated NaN.
fp_is_natval(freg)	Returns true when floating register contains a NaTVal
fp_is_normal(freg)	Returns true when floating register contains a normal number.
fp_is_pos_inf(freg)	Returns true when floating register contains a positive infinity.
fp_is_qnan(freg)	Returns true when floating register contains a quiet NaN.
fp_is_snan(freg)	Returns true when floating register contains a signalling NaN.
fp_is_unorm(freg)	Returns true when floating register contains an unnormalized number.
fp_is_unsupported(freg)	Returns true when floating register contains an unsupported format.
fp_less_than(fr1, fr2)	IEEE standard less-than relationship test.
fp_lesser_or_equal(fr1, fr2)	IEEE standard less-than or equal-to relationship test
fp_mem_to_fr_format(mem, size)	Converts a floating-point value in memory format to floating-point register format.
fp_normalize(fr1)	Normalizes an unnormalized fp value. This function flushes to zero any unnormal values which can not be represented in the register file
fp_raise_fault(tmp_fp_env)	Checks the local instruction state for any faulting conditions which require an interruption to be raised.
fp_raise_traps(tmp_fp_env)	Checks the local instruction state for any trapping conditions which require an interruption to be raised.
fp_reg_bank_conflict(f1, f2)	Returns true if the two specified FRs are in the same bank.
fp_reg_disabled(f1, f2, f3, f4)	Check for possible disabled floating-point register faults.
fp_reg_read(freg)	Reads the FR and gives canonical double-extended denormals (and pseudo-denormals) their true mathematical exponent. Other classes of operands are unaltered.
fp_unordered(fr1, fr2)	IEEE standard unordered relationship
fp_update_fpsr(sf, tmp_fp_env)	Copies a floating-point instruction's local state into the global FPSR.
fp_update_psr(dest_freg)	Conditionally sets PSR.mfl or PSR.mfh based on dest_freg.
fpcmp_exception_fault_check(f2, f3, frel, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcmp` instruction.
fpcvt_exception_fault_check(f2, signed_form, trunc_form, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpcvt.fx`, `fpcvt.fxu`, `fpcvt.fx.trunc`, and `fpcvt.fxu.trunc` instructions. It propagates NaNs.
fpma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpma` instruction. It propagates NaNs and special IEEE results.
fpminmax_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpmin`, `fpmax`, `fpamin` and `fpamax` instructions.
fpms_fpnma_exception_fault_check(f2, f3, f4, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `fpms` and `fpnma` instructions. It propagates NaNs and special IEEE results.
fprcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprcpa` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.
fprsqrta_exception_fault_check(f3, sf, *tmp_fp_env, *limits_check)	Checks for all floating-point faulting conditions for the `fprsqrta` instruction. It propagates NaNs and special IEEE results. It also indicates operand limit violations.

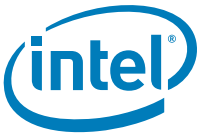


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
frcpa_exception_fault_check(f2, f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frcpa` instruction. It propagates NaNs and special IEEE results.
frsqrta_exception_fault_check(f3, sf, *tmp_fp_env)	Checks for all floating-point faulting conditions for the `frsqrta` instruction. It propagates NaNs and special IEEE results
ignored_field_mask(regclass, reg, value)	Boolean function that returns value with bits cleared to 0 corresponding to ignored bits for the specified register and register type.
impl_check_mov_itir()	Implementation-specific function that returns TRUE if ITIR is checked for reserved fields and encodings on a mov to ITIR instruction.
impl_check_mov_psr_l(gr)	Implementation-specific function to check bits {63:32} of `gr` corresponding to reserved fields of the PSR for Reserved Register/Field fault.
impl_check_tlb_itir()	Implementation-specific function that returns TRUE if all fields of ITIR are checked for reserved encodings on a TLB insert instruction regardless of whether the translation is present.
impl_ia32_ar_reserved_ignored(ar3)	Implementation-specific function which indicates how the reserved and ignored fields in the specified IA-32 application register, `ar3`, behave. If it returns FALSE, the reserved and/or ignored bits in the specified application register can be written, and when read they return the value most-recently written. If it returns TRUE, attempts to write a non-zero value to a reserved field in the specified application register cause a Reserved Register/Field fault, and reads return 0; writing to an ignored field in the specified application register is ignored, and reads return the constant value defined for that field.
impl_iib()	Implementation-specific function which indicates whether Interruption Instruction Bundle registers (IIB0-1) are implemented.
impl_itir_cwi_mask()	Implementation-specific function that either returns the value passed to it or the value passed to it masked with zeros in bit positions {63:32} and/or {1:0}.
impl_ruc()	Implementation-specific function which indicates whether Resource Utilization Counter (RUC) application register is implemented.
impl_uia_fault_supported()	Implementation-specific function that either returns TRUE if the processor reports unimplemented instruction addresses with an Unimplemented Instruction Address fault, and returns FALSE if the processor reports them with an Unimplemented Instruction Address trap.
implemented_vm()	Returns TRUE if the processor implements the PSR.vm bit (regardless of whether virtual machine features are enabled or disabled).
instruction_implemented(inst)	Implementation-dependent routine which returns TRUE or FALSE, depending on whether `inst` is implemented.
instruction_serialize()	Ensures all prior register updates with side-effects are observed before subsequent instruction and data memory references are performed. Also ensures prior SYNC.i operations have been observed by the instruction cache.
instruction_synchronize()	Synchronizes the instruction and data stream for Flush Cache operations. This function ensures that when prior Flush Cache operations are observed by the local data cache they are observed by the local instruction cache, and when prior Flush Cache operations are observed by another processor's data cache they are observed within the same processor's instruction cache.
is_finite(freg)	Returns true when floating register contains a finite number.
is_ignored_reg(regnum)	Boolean function that returns true if `regnum` is an ignored application register, otherwise false.
is_inf(freg)	Returns true when floating register contains an infinite number.
is_interruption_cr(regnum)	Boolean function that returns true if `regnum` is one of the Interruption Control registers (see Section 3.3.5, "Interruption Control Registers" on page 2:36), otherwise false.
is_kernel_reg(ar_addr)	Returns a one if `ar_addr` is the address of a kernel register application register
is_read_only_reg(rtype, raddr)	Returns a one if the register addressed by `raddr` in the register bank of type `rtype` is a read only register.

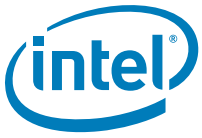


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
is_reserved_field(regclass, arg2, arg3)	Returns true if the specified data would write a one in a reserved field.
is_reserved_reg(regclass, regnum)	Returns true if register `regnum` is reserved in the `regclass` register file.
is_supported_hint(hint)	Returns true if the implementation supports the specified `hint`. This function may depend on factors other than the `hint` value, such as which execution unit it is executed on or the slot number the instruction was encoded in.
itlb_ar()	Returns the page access rights from the ITLB for the page addressed by the current IP, or INVALID_AR if PSR.it is 0.
make_icache_coherent(paddr)	The cache line addressed by the physical address `paddr` is flushed in an implementation-specific manner that ensures that the instruction cache is coherent with the data caches.
mem_flush(paddr)	The line addressed by the physical address `paddr` is invalidated in all levels of the memory hierarchy above memory and written back to memory if it is inconsistent with memory.
mem_flush_pending_stores()	The processor is instructed to start draining pending stores in write coalescing and write buffers. This operation is a hint. There is no indication when prior stores have actually been drained.
mem_implicit_prefetch(vaddr, hint, type)	Moves the line addressed by `vaddr` to the location of the memory hierarchy specified by `hint`. This function is implementation dependent and can be ignored. The `type` allows the implementation to distinguish prefetches for different instruction types.
mem_promote(paddr, mtype, hint)	Moves the line addressed by `paddr` to the highest level of the memory hierarchy conditioned by the access hints specified by `hint`. Implementation dependent and can be ignored.
mem_read(paddr, size, border, mattr, otype, hint)	Returns the `size` bytes starting at the physical memory location specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE.
mem_read_pair(*low_value, *high_value, paddr, size, border, mattr, otype, hint)	Reads the `size` / 2 bytes of memory starting at the physical memory address specified by `paddr` into `low_value`, and the `size` / 2 bytes of memory starting at the physical memory address specified by (`paddr` + `size` / 2) into `high_value`, with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or ACQUIRE. No value is returned.
mem_write(value, paddr, size, border, mattr, otype, hint)	Writes the least significant `size` bytes of `value` into memory starting at the physical memory address specified by `paddr` with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_write16(gr_value, ar_value, paddr, border, mattr, otype, hint)	Writes the 8 bytes of `gr_value` into memory starting at the physical memory address specified by `paddr`, and the 8 bytes of `ar_value` into memory starting at the physical memory address specified by (`paddr` + 8), with byte order specified by `border`, memory attributes specified by `mattr`, and access hint specified by `hint`. `otype` specifies the memory ordering attribute of this access, and must be UNORDERED or RELEASE. No value is returned.
mem_xchg(data, paddr, size, byte_order, mattr, otype, hint)	Returns size bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. After the read, the least significant `size` bytes of data are written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and must be ACQUIRE.

Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
mem_xchg_add(add_val, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. The least significant `size` bytes of the sum of the value read from memory and `add_val` is then written to `size` bytes in memory starting at the physical address specified by `paddr`. The read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg_cond(cmp_val, data, paddr, size, byte_order, mattr, otype, hint)	Returns `size` bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val,` then the least significant `size` bytes of data are written to `size bytes` in memory starting at the physical address specified by `paddr`. If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by `mattr` and the byte ordering in memory is specified by `byte_order`. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
mem_xchg16_cond(cmp_val, gr_data, ar_data, paddr, byte_order, mattr, otype, hint)	Returns 8 bytes from memory starting at the physical address specified by `paddr`. The read is conditioned by the locality hint specified by `hint`. If the value read from memory is equal to `cmp_val`, then the 8 bytes of `gr_data` are written to 8 bytes in memory starting at the physical address specified by (`paddr` & ~0x8), and the 8 bytes of `ar_data` are written to 8 bytes in memory starting at the physical address specified by ((`paddr` & ~0x8) + 8). If the write is performed, the read and write are performed atomically. Both the read and the write are conditioned by the memory attribute specified by mattr and the byte ordering in memory is specified by `byte_order`. The byte ordering only affects the ordering of bytes within each of the 8-byte values stored. `otype` specifies the memory ordering attribute of this access, and has the value ACQUIRE or RELEASE.
ordering_fence()	Ensures prior data memory references are made visible before future data memory references are made visible by the processor.
partially_implemented_ip()	Implementation-dependent routine which returns TRUE if the implementation, on an Unimplemented Instruction Address trap, writes IIP with the sign-extended virtual address or zero-extended physical address for what would have been the next value of IP. Returns FALSE if the implementation, on this trap, simply writes IIP with the full address which would have been the next value of IP.
pending_virtual_interrupt()	Check for unmasked pending virtual interrupt.
pr_phys_to_virt(phys_id)	Returns the virtual register id of the predicate from the physical register id, `phys_id` of the predicate.
rotate_regs()	Decrements the Register Rename Base registers, effectively rotating the register files. CFM.rrb.gr is decremented only if CFM.sor is non-zero.
rse_enable_current_frame_load()	If the RSE load pointer (RSE.BSPLoad) is greater than AR[BSP], the `RSE.CFLE` bit is set to indicate that mandatory RSE loads are allowed to restore registers in the current frame (in no other case does the RSE spill or fill registers in the current frame). This function does not perform mandatory RSE loads. This procedure does not cause any interruptions.
rse_ensure_regs_loaded(number_of_bytes)	All registers and NaT collections between `AR[BSP]` and `(AR[BSP]-number_of_bytes)` which are not already in stacked registers are loaded into the register stack with mandatory RSE loads. If the number of registers to be loaded is greater than `RSE.N_STACK_PHYS` an Illegal Operation fault is raised. All registers starting with backing store address (AR[BSP] - 8) and decrementing down to and including backing store address (AR[BSP] - number_of_bytes) are made part of the dirty partition. With exception of the current frame, all other stacked registers are made part of the invalid partition. Note that `number_of_bytes` may be zero. The resulting sequence of RSE loads may be interrupted. Mandatory RSE loads may cause an interruption; see Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_invalidate_non_current_regs()	All registers outside the current frame are invalidated.

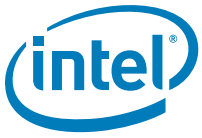


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
rse_load(type)	Restores a register or NaT collection from the backing store (load_address = RSE.BspLoad - 8). If load_address{8:3} is equal to 0x3f then a NaT collection is loaded into a NaT dispersal register. (dispersal register may not be the same as AR[RNAT].) If load_address{8:3} is not equal to 0x3f then the register RSE.LoadReg - 1 is loaded and the NaT bit for that register is set to dispersal_register{load_address{8:3}}. If the load is successful RSE.BspLoad is decremented by 8. If the load is successful and a register was loaded RSE.LoadReg is decremented by 1 (possibly wrapping in the stacked registers). The load moves a register from the invalid partition to the current frame if RSE.CFLE is 1, or to the clean partition if RSE.CFLE is 0. For mandatory RSE loads, type is MANDATORY. Mandatory RSE loads may cause interruptions. See Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_new_frame(current_frame_size, new_frame_size)	A new frame is defined without changing any register renaming. The new frame size is completely defined by the new_frame_size parameter (successive calls are not cumulative). If new_frame_size is larger than current_frame_size and the number of registers in the invalid and clean partitions is less than the size of frame growth then mandatory RSE stores are issued until enough registers are available. The resulting sequence of RSE stores may be interrupted. Mandatory RSE stores may cause interruptions; see Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_preserve_frame(preserved_frame_size)	The number of registers specified by preserved_frame_size are marked to be preserved by the RSE. Register renaming causes the preserved_frame_size registers after GR[32] to be renamed to GR[32].AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored.
rse_restore_frame(preserved_sol, growth, current_frame_size)	The first two parameters define how the current frame is about to be updated by a branch return or rfi: preserved_sol defines how many registers need to be restored below RSE.BOF; growth defines by how many registers the top of the current frame will grow (growth will generally be negative). The number of registers specified by preserved_sol are marked to be restored. Register renaming causes the preserved_sol registers before GR[32] to be renamed to GR[32].AR[BSP] is updated to contain the backing store address where the new GR[32] will be stored. If the number of dirty and clean registers is less than preserved_sol then mandatory RSE loads must be issued before the new current frame is considered valid. This function does not perform mandatory RSE loads. This function returns TRUE if the preserved frame grows beyond the invalid and clean regions into the dirty region. In this case the third argument, current_frame_size, is used to force the returned to frame to zero (see Section 6.5.5, “Bad PFS used by Branch Return” on page 2:143).
rse_store(type)	Saves a register or NaT collection to the backing store (store_address = AR[BSPSTORE]). If store_address{8:3} is equal to 0x3f then the NaT collection AR[RNAT] is stored. If store_address{8:3} is not equal to 0x3f then the register RSE.StoreReg is stored and the NaT bit from that register is deposited in AR[RNAT]{store_address{8:3}}. If the store is successful AR[BSPSTORE] is incremented by 8. If the store is successful and a register was stored RSE.StoreReg is incremented by 1 (possibly wrapping in the stacked registers). This store moves a register from the dirty partition to the clean partition. For mandatory RSE stores, type is MANDATORY. Mandatory RSE stores may cause interruptions. See Table 6-6, “RSE Interruption Summary” on page 6-145.
rse_update_internal_stack_pointers(new_store_pointer)	Given a new value for AR[BSPSTORE] (new_store_pointer) this function computes the new value for AR[BSP]. This value is equal to new_store_pointer plus the number of dirty registers plus the number of intervening NaT collections. This means that the size of the dirty partition is the same before and after a write to AR[BSPSTORE]. All clean registers are moved to the invalid partition.
sign_ext(value, pos)	Returns a 64 bit number with bits pos-1 through 0 taken from value and bit pos-1 of value replicated in bit positions pos through 63. If pos is greater than or equal to 64, value is returned.

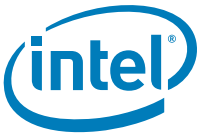


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
spontaneous_deferral(paddr, size, border, mattr, otype, hint, *defer)	Implementation-dependent routine which optionally forces `*defer` to TRUE if all of the following are true: spontaneous deferral is enabled, spontaneous deferral is permitted by the programming model, and the processor determines it would be advantageous to defer the speculative load (e.g., based on a miss in some particular level of cache).
spontaneous_deferral_enabled()	Implementation-dependent routine which returns TRUE or FALSE, depending on whether spontaneous deferral of speculative loads is enabled or disabled in the processor.
tlb_access_key(vaddr, itype)	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.
tlb_broadcast_purge(rid, vaddr, size, type)	Sends a broadcast purge DTC and ITC transaction to other processors in the multiprocessor coherency domain, where the region identifier (`rid`), virtual address (`vaddr`) and page size (`size`) specify the translation entry to purge. The operation waits until all processors that receive the purge have completed the purge operation. The purge type (`type`) specifies whether the ALAT on other processors should also be purged in conjunction with the TC.
tlb_enter_privileged_code()	This function determines the new privilege level for epc from the TLB entry for the page containing this instruction. If the page containing the epc instruction has execute-only page access rights and the privilege level assigned to the page is higher than (numerically less than) the current privilege level, then the current privilege level is set to the privilege level field in the translation for the page containing the epc instruction.
tlb_grant_permission(vaddr, type, pl)	Returns a boolean indicating if read, write access is granted for the specified virtual memory address (`vaddr`) and privilege level (`pl`). The access type (`type`) specifies either read or write. The following faults are checked:: • 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 If a fault is generated, this function does not return.
tlb_insert_data(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the DTLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true the entry is placed in the TR section, otherwise the TC section.
tlb_insert_inst(slot, pte0, pte1, vaddr, rid, tr)	Inserts an entry into the ITLB, at the specified `slot` number. `pte0`, `pte1` compose the translation. `vaddr` and `rid` specify the virtual address and region identifier for the translation. If `tr` is true, the entry is placed in the TR section, otherwise the TC section.
tlb_may_purge_dtc_entries(rid, vaddr, size)	May locally purge DTC entries that match the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.

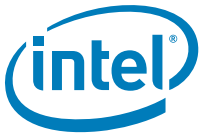


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
tlb_may_purge_itc_entries(rid, vaddr, size)	May locally purge ITC entries that match the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). May also invalidate entries that partially overlap the parameters. The extent of purging is implementation dependent. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_dtc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTC entries matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr{63:61}` (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing DTR translation, an implementation may generate a machine check abort.
tlb_must_purge_dtr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, DTR entries matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr{63:61}` (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_must_purge_itc_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITC entry matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr`{63:61} (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache. If the specified purge values overlap with an existing ITR translation, an implementation may generate a machine check abort.
tlb_must_purge_itr_entries(rid, vaddr, size)	Purges all local, possibly overlapping, ITR entry matching the specified region identifier (`rid`), virtual address (`vaddr`) and page size (`size`). `vaddr`{63:61} (VRN) is ignored in the purge, i.e all entries that match `vaddr`{60:0} must be purged regardless of the VRN bits. If the purge size is not supported, an implementation may generate a machine check abort or over purge the translation cache up to and including removal of all entries from the translation cache.
tlb_purge_translation_cache(loop)	Removes 1 to N translations from the local processor's ITC and DTC. The number of entries removed is implementation specific. The parameter `loop` is used to generate an implementation-specific purge parameter.
tlb_replacement_algorithm(tlb)	Returns the next ITC or DTC slot number to replace. Replacement algorithms are implementation specific. `tlb` specifies to perform the algorithm on the ITC or DTC.
tlb_search_pkr(key)	Searches for a valid protection key register with a matching protection `key`. The search algorithm is implementation specific. Returns the PKR register slot number if found, otherwise returns Not Found.

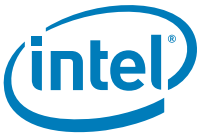


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
tlb_translate(vaddr, size, type, cpl, *attr, *defer)	Returns the translated data physical address for the specified virtual memory address (`vaddr`) when translation enabled; otherwise, returns `vaddr`. `size` specifies the size of the access, `type` specifies the type of access (e.g., read, write, advance, spec). `cpl` specifies the privilege level for access checking purposes. `*attr` returns the mapped physical memory attribute. If any fault conditions are detected and deferred, tlb_translate returns with `*defer` set. If a fault is generated but the fault is not deferred, tlb_translate does not return. tlb_translate checks the following faults: • 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
tlb_translate_nonaccess(vaddr, type)	Returns the translated data physical address for the specified virtual memory address (`vaddr`). `type` specifies the type of access (e.g., `FC`, `TPA`). If a fault is generated, tlb_translate_nonaccess does not return. The following faults are checked: • Unimplemented Data Address fault • Virtualization fault (`tpa` only) • 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)
tlb_vhpt_hash(vrn, vaddr61, rid, size)	Generates a VHPT entry address for the specified virtual region number (`vrn`) and 61-bit virtual offset (`vaddr61`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_hash hashes `vaddr`, `rid` and `size` parameters to produce a hash index. The hash index is then masked based on PTA.size and concatenated with PTA.base to generate the VHPT entry address. The long format hash is implementation specific.
tlb_vhpt_tag(vaddr, rid, size)	Generates a VHPT tag identifier for the specified virtual address (`vaddr`), region identifier (`rid`) and page size (`size`). Tlb_vhpt_tag hashes the `vaddr`, rid and `size` parameters to produce translation identifier. The tag in conjunction with the hash index is used to uniquely identify translations in the VHPT. Tag generation is implementation specific. All processor models tag function must guarantee that bit 63 of the generated tag is zero (ti bit).
undefined()	Returns an undefined 64-bit value.
undefined_behavior()	Causes undefined processor behavior. Extent of undefined behavior is described in Section 3.5, “Undefined Behavior” on page 1:44.

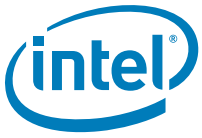
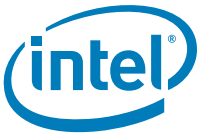


Table 3-1. Pseudo-code Functions (Continued)

Function	Operation
unimplemented_physical_address(paddr)	Return TRUE if the presented physical address is unimplemented on this processor model; FALSE otherwise. This function is model specific.
unimplemented_virtual_address(vaddr, vm)	Return TRUE if the presented virtual address is unimplemented on this processor model; FALSE otherwise. If vm is 1, one additional bit of virtual address is treated as unimplemented. This function is model specific.
vm_disabled()	Returns TRUE if the processor implements the PSR.vm bit and virtual machine features are disabled. See Section 3.4, “Processor Virtualization” on page 2:44 in SDM and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:448 in SDM for details.
vmsw_disabled()	Returns TRUE if the processor implements the PSR.vm bit and the `vmsw` instruction is disabled. See Section 3.4, “Processor Virtualization” on page 2:44 in SDM and “PAL_PROC_GET_FEATURES – Get Processor Dependent Features (17)” on page 2:448 in SDM for details.
zero_ext(value, pos)	Returns a 64 bit unsigned number with bits `pos`-1 through 0 taken from `value` and zeroes in bit positions `pos` through 63. If `pos` is greater than or equal to 64, `value` is returned.

§



V3-L Chapter 4 Instruction Formats

Instruction Formats 4

Each Itanium instruction is categorized into one of six types; each instruction type may be executed on one or more execution unit types. Table 4-1 lists the instruction types and the execution unit type on which they are executed:

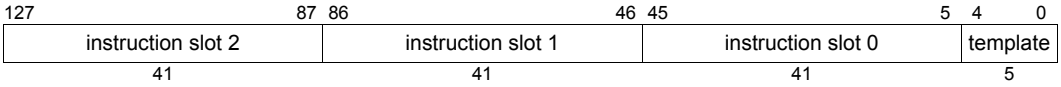
Table 4-1. Relationship between Instruction Type and Execution Unit Type

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit/B-unit[a]

a. L+X Major Opcodes 0 - 7 execute on an I-unit. L+X Major Opcodes 8 - F execute on a B-unit.

Three instructions are grouped together into 128-bit sized and aligned containers called **bundles**. Each bundle contains three 41-bit **instruction slots** and a 5-bit template field. The format of a bundle is depicted in Figure 4-1.

Figure 4-1. Bundle Format



The template field specifies two properties: stops within the current bundle, and the mapping of instruction slots to execution unit types. Not all combinations of these two properties are allowed - Table 4-2 indicates the defined combinations. The three rightmost columns correspond to the three instruction slots in a bundle; listed within each column is the execution unit type controlled by that instruction slot for each encoding of the template field. A double line to the right of an instruction slot indicates that a stop occurs at that point within the current bundle. See “Instruction Encoding Overview” on page 38 for the definition of a stop. Within a bundle, execution order proceeds from slot 0 to slot 2. Unused template values (appearing as empty rows in Table 4-2) are reserved and cause an Illegal Operation fault.

Extended instructions, used for long immediate integer and long branch instructions, occupy two instruction slots. Depending on the major opcode, extended instructions execute on a B-unit (long branch/call) or an I-unit (all other L+X instructions).

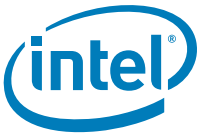


Table 4-2. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a]
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

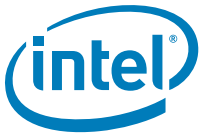
a. The MLX template was formerly called MLI, and for compatibility, the X slot may encode break.i and nop.i in addition to any X-unit instruction.

4.1 Format Summary

All instructions in the instruction set are 41 bits in length. The leftmost 4 bits (40:37) of each instruction are the major opcode. Table 4-3 shows the major opcode assignments for each of the 5 instruction types — ALU (A), Integer (I), Memory (M), Floating-point (F), and Branch (B). Bundle template bits are used to distinguish among the 4 columns, so the same major op values can be reused in each column.

Unused major ops (appearing as blank entries in Table 4-3) behave in one of four ways:

- Ignored major ops (white entries in Table 4-3) execute as `nop` instructions.



- Reserved major ops (light gray in the gray scale version of Table 4-3, brown in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 major ops (dark gray in the gray scale version of Table 4-3, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a nop instruction if 0.
- Reserved if PR[qp] is 1 B-unit major ops (medium gray in the gray scale version of Table 4-3, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a nop instruction if 0. These differ from the Reserved if PR[qp] is 1 major ops (purple) only in their RAW dependency behavior (see "RAW Dependency Table" on page 374).

Table 4-3. Major Opcode Assignments

Major Op (Bits 40:37)	Instruction Type				
	I/A	M/A	F	B	L+X
0	Misc 0	Sys/Mem Mgmt 0	FP Misc 0	Misc/Indirect Branch 0	Misc 0
1	1	Sys/Mem Mgmt 1	FP Misc 1	Indirect Call 1	1
2	2	2	2	Indirect Predict/Nop 2	2
3	3	3	3	3	3
4	Deposit 4	Int Ld +Reg/getf 4	FP Compare 4	IP-relative Branch 4	4
5	Shift/Test Bit 5	Int Ld/St +Imm 5	FP Class 5	IP-rel Call 5	5
6	6	FP Ld/St +Reg/setf 6	6	6	movl 6
7	MM Mpy/Shift 7	FP Ld/St +Imm 7	7	IP-relative Predict 7	7
8	ALU/MM ALU 8	ALU/MM ALU 8	fma 8	e 8	8
9	Add Imm_{22} 9	Add Imm_{22} 9	fma 9	e 9	9
A	A	A	fms A	e A	A
B	B	B	fms B	e B	B
C	Compare C	Compare C	fnma C	e C	Long Branch C
D	Compare D	Compare D	fnma D	e D	Long Call D
E	Compare E	Compare E	fselect/xma E	e E	E
F	F	F	F	e F	F

Table 4-4 on page 3:184 summarizes all the instruction formats. The instruction fields are color-coded for ease of identification, as described in Table 4-6 on page 3:186. A color version of this chapter is available for those heavily involved in working with the instruction encodings.
The instruction field names, used throughout this chapter, are described in Table 4-6 on page 3:186. The set of special notations (such as whether an instruction is privileged) are listed in Table 4-7 on page 3:187. These notations appear in the "Instruction" column of the opcode tables.

Most instruction containing immediates encode those immediates in more than one instruction field. For example, the 14-bit immediate in the Add Imm_{14} instruction (format A4) is formed from the imm_{7b}, imm_{6d}, and s fields. Table 4-80 on page 3:259 shows how the immediates are formed from the instruction fields for each instruction which has an immediate.

Table 4-4. Instruction Format Summary

		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU	A1	8					x_{2a}		v_e		x_4			x_{2b}		r_3							r_2							r_1									qp				
Shift L and Add	A2	8					x_{2a}		v_e		x_4			ct_{2d}		r_3							r_2							r_1									qp				
ALU Imm_8	A3	8				s	x_{2a}		v_e		x_4			x_{2b}		r_3							imm_{7b}							r_1									qp				
Add Imm_{14}	A4	8				s	x_{2a}		v_e		imm_{6d}					r_3							imm_{7b}							r_1									qp				
Add Imm_{22}	A5	9				s	imm_{9d}									imm_{5c}					r_3		imm_{7b}							r_1									qp				
Compare	A6	C - E				t_b	x_2		t_a		p_2					r_3							r_2						c	p_1								qp				
Compare to Zero	A7	C - E				t_b	x_2		t_a		p_2					r_3							0						c	p_1								qp				
Compare Imm_8	A8	C - E				s	x_2		t_a		p_2					r_3							imm_{7b}						c	p_1								qp				
MM ALU	A9	8				z_a	x_{2a}		z_b		x_4			x_{2b}		r_3							r_2							r_1									qp				
MM Shift and Add	A10	8				z_a	x_{2a}		z_b		x_4			ct_{2d}		r_3							r_2							r_1									qp				
MM Multiply Shift	I1	7				z_a	x_{2a}		z_b	v_e	ct_{2d}			x_{2b}		r_3							r_2							r_1									qp				
MM Mpy/Mix/Pack	I2	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		r_3							r_2							r_1									qp				
MM Mux1	I3	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}					mbt_{4c}				r_2							r_1									qp				
MM Mux2	I4	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		mht_{8c}							r_2							r_1									qp				
Shift R Variable	I5	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		r_3							r_2							r_1									qp				
MM Shift R Fixed	I6	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		r_3							$count_{5b}$							r_1									qp				
Shift L Variable	I7	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		r_3							r_2							r_1									qp				
MM Shift L Fixed	I8	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}					$ccount_{5c}$				r_2							r_1									qp				
Bit Strings	I9	7				z_a	x_{2a}		z_b	v_e	x_{2c}			x_{2b}		r_3							0							r_1									qp				
Shift Right Pair	I10	5					x_2		x		$count_{6d}$					r_3							r_2							r_1									qp				
Extract	I11	5					x_2		x		len_{6d}					r_3							pos_{6b}					y		r_1									qp				
Dep.Z	I12	5					x_2		x		len_{6d}					y	$cpos_{6c}$						r_2							r_1									qp				
Dep.Z Imm_8	I13	5				s	x_2		x		len_{6d}					y	$cpos_{6c}$						imm_{7b}							r_1									qp				
Deposit Imm_1	I14	5				s	x_2		x		len_{6d}					r_3							$cpos_{6b}$							r_1									qp				
Deposit	I15	4				$cpos_{6d}$						len_{4d}				r_3							r_2							r_1									qp				
Test Bit	I16	5				t_b	x_2		t_a		p_2					r_3							pos_{6b}					y	c	p_1								qp				
Test NaT	I17	5				t_b	x_2		t_a		p_2					r_3						x						y	c	p_1								qp				
Nop/Hint	I18	0				i	x_3				x_6					y	imm_{20a}																					qp				
Break	I19	0				i	x_3				x_6						imm_{20a}																					qp				
Int Spec Check	I20	0				s	x_3				imm_{13c}												r_2							imm_{7a}									qp				
Move to BR	I21	0					x_3				$timm_{9c}$									ih	x	wh	r_2						p	pbtv		b_1						qp				
Move from BR	I22	0					x_3				x_6																	b_2		r_1									qp				
Move to Pred	I23	0				s	x_3					$mask_{8c}$											r_2							$mask_{7a}$									qp				
Move to Pred Imm_{44}	I24	0				s	x_3				imm_{27a}																												qp				
Move from Pred/IP	I25	0					x_3				x_6																			r_1									qp				
Move to AR	I26	0					x_3				x_6					ar_3							r_2																qp				
Move to AR Imm_8	I27	0				s	x_3				x_6					ar_3							imm_{7b}																qp				
Move from AR	I28	0					x_3				x_6					ar_3														r_1									qp				
Sxt/Zxt/Czx	I29	0					x_3				x_6					r_3														r_1									qp				
Test Feature	I30	5				t_b	x_2		t_a		p_2					0						x	imm_{5b}					y	c	p_1								qp				
Int Load	M1	4				m					x_6			hint	x	r_3						h								r_1									qp				
Int Load +Reg	M2	4				m					x_6			hint	x	r_3							r_2							r_1									qp				
Int Load +Imm	M3	5				s					x_6			hint	i	r_3							imm_{7b}							r_1									qp				
Int Store	M4	4				m					x_6			hint	x	r_3							r_2						h									qp				
Int Store +Imm	M5	5				s					x_6			hint	i	r_3							r_2							imm_{7a}									qp				
FP Load	M6	6				m					x_6			hint	x	r_3						h								f_1									qp				
FP Load +Reg	M7	6				m					x_6			hint	x	r_3							r_2							f_1									qp				
FP Load +Imm	M8	7				s					x_6			hint	i	r_3							imm_{7b}							f_1									qp				
FP Store	M9	6				m					x_6			hint	x	r_3							f_2						h									qp				
FP Store +Imm	M10	7				s					x_6			hint	i	r_3							f_2							imm_{7a}									qp				
FP Load Pair	M11	6				m					x_6			hint	x	r_3							f_2							f_1									qp				
FP Load Pair +Imm	M12	6				m					x_6			hint	x	r_3							f_2							f_1									qp				
Line Prefetch	M13	6				m					x_6			hint	x	r_3														h									qp				
Line Prefetch +Reg	M14	6				m					x_6			hint	x	r_3							r_2						h									qp				
Line Prefetch +Imm	M15	7				s					x_6			hint	i	r_3							imm_{7b}						h									qp				
(Cmp &) Exchg	M16	4				m					x_6			hint	x	r_3							r_2							r_1									qp				
Fetch & Add	M17	4				m					x_6			hint	x	r_3												s	i_{2b}	r_1									qp				
Set FR	M18	6				m					x_6				x								r_2							f_1									qp				
Get FR	M19	4				m					x_6				x								f_2							r_1									qp				
		40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 4-4. Instruction Format Summary (Continued)

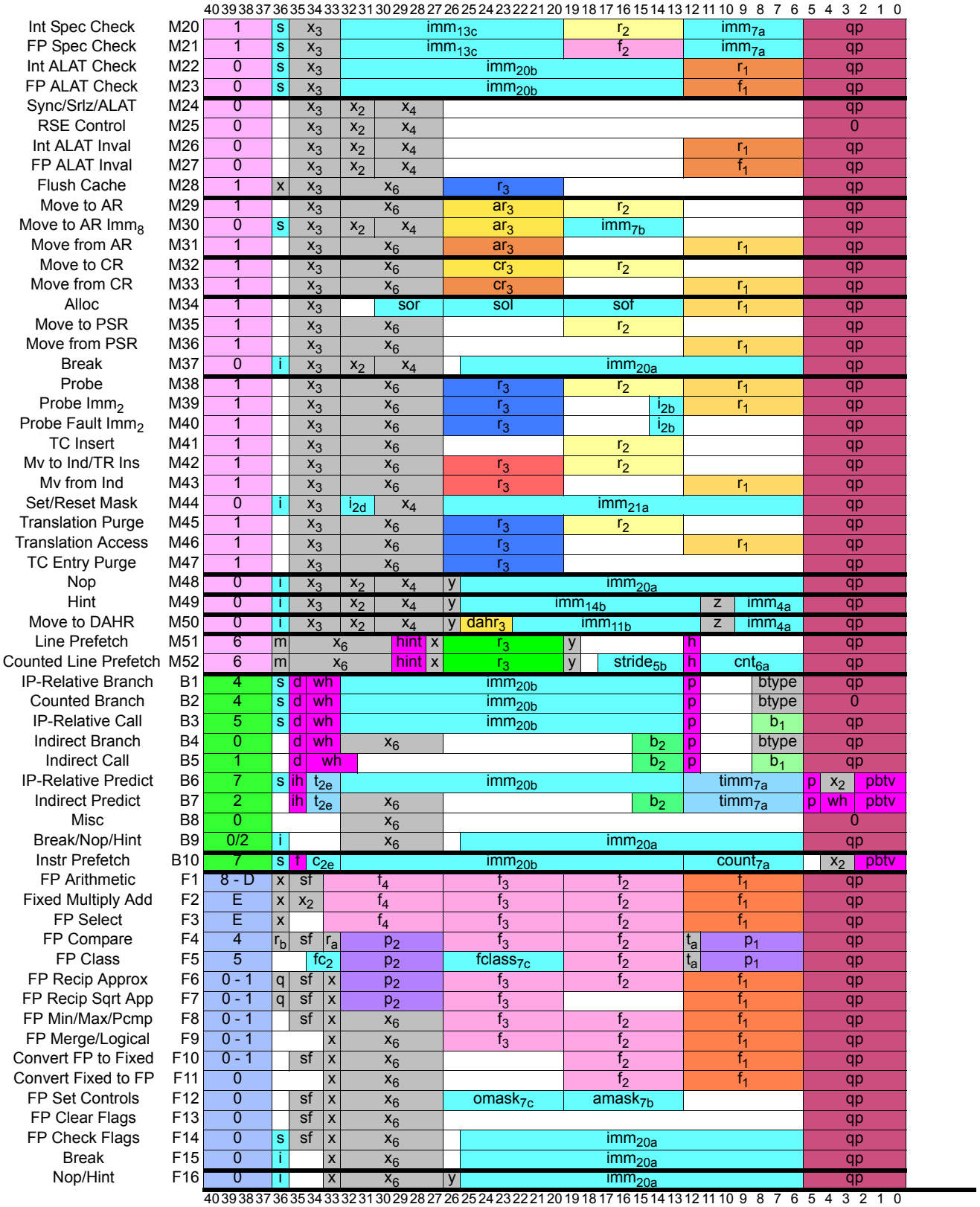


Table 4-4. Instruction Format Summary (Continued)

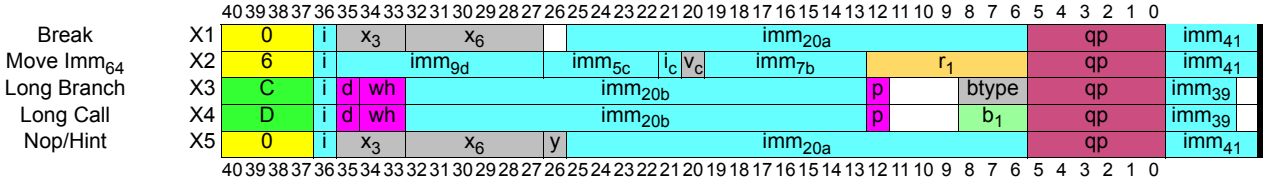


Table 4-5. Instruction Field Color Key

Field & Color	
ALU Instruction	Opcode Extension
Integer Instruction	Opcode Hint Extension
Memory Instruction	Immediate
Branch Instruction	Indirect Source
Floating-point Instruction	Predicate Destination
Integer Source	Integer Destination
Memory Source	Memory Source & Destination
Shift Source	Shift Immediate
Special Register Source	Special Register Destination
Floating-point Source	Floating-point Destination
Branch Source	Branch Destination
Address Source	Branch Tag Immediate
Qualifying Predicate	Reserved Instruction
Ignored Field/Instruction	Reserved Inst if PR[qp] is 1
	Reserved B-type Inst if PR[qp] is 1

Table 4-6. Instruction Field Names

Field Name	Description
ar_3	application register source/target
b_1, b_2	branch register source/target
btype	branch type opcode extension
c	complement compare relation opcode extension
$ccount_{5c}$	multimedia shift left complemented shift count immediate
$count_{5b}$, $count_{6d}$	multimedia shift right/shift right pair shift count immediate
$cpos_x$	deposit complemented bit position immediate
cr_3	control register source/target
ct_{2d}	multimedia multiply shift/shift and add shift count immediate
d	branch cache deallocation hint opcode extension
$dahr_3$	data access hint register target
f_n	floating-point register source/target
fc_2, $fclass_{7c}$	floating-point class immediate
hint	memory reference hint opcode extension
i, i_{2b}, i_{2d}, imm_x	immediate of length 1, 2, or *x*
ih	branch importance hint opcode extension

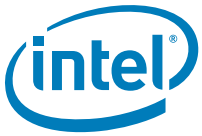


Table 4-6. Instruction Field Names (Continued)

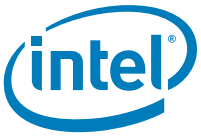
Field Name	Description
len_{4d}, len_{6d}	extract/deposit length immediate
m	memory reference post-modify opcode extension
$mask_x$	predicate immediate mask
mbt_{4c}, mht_{8c}	multimedia mux1/mux2 immediate
p	sequential prefetch hint opcode extension
p_1, p_2	predicate register target
pos_{6b}	test bit/extract bit position immediate
q	floating-point reciprocal/reciprocal square-root opcode extension
qp	qualifying predicate register source
r_n	general register source/target
s	immediate sign bit
sf	floating-point status field opcode extension
sof, sol, sor	alloc size of frame, size of locals, size of rotating immediates
t_a, t_b	compare type opcode extension
t_{2e}, $timm_x$	branch predict tag immediate
v_x	reserved opcode extension field
wh	branch whether hint opcode extension
x, x_n	opcode extension of length 1 or *n*
y	extract/deposit/test bit/test NaT/hint opcode extension
z_a, z_b	multimedia operand size opcode extension

Table 4-7. Special Instruction Notations

Notation	Description
e	instruction ends an instruction group when taken, or for Reserved if PR[qp] is 1 (cyan) encodings and non-branch instructions with a qualifying predicate, when its PR[qp] is 1, or for Reserved (brown) encodings, unconditionally
f	instruction must be the first instruction in an instruction group and must either be in instruction slot 0 or in instruction slot 1 of a template having a stop after slot 0
i	instruction is allowed in the I slot of an MLI template
l	instruction must be the last in an instruction group
p	privileged instruction
t	instruction is only allowed in instruction slot 2

The remaining sections of this chapter present the detailed encodings of all instructions. The “A-Unit Instruction encodings” are presented first, followed by the “I-Unit Instruction Encodings” on page 198, “M-Unit Instruction Encodings” on page 211, “B-Unit Instruction Encodings” on page 240, “F-Unit Instruction Encodings” on page 247, and “X-Unit Instruction Encodings” on page 256. Within each section, the instructions are grouped by function, and appear with their instruction format in the same order as in Table 4-4, “Instruction Format Summary” on page 3-184. The opcode extension fields are briefly described and tables present the opcode extension assignments. Unused instruction encodings (appearing as blank entries in the opcode extensions tables) behave in one of four ways:

- Ignored instructions (white color entries in the tables) execute as `nop` instructions.



- Reserved instructions (light gray color in the gray scale version of the tables, brown color in the color version) cause an Illegal Operation fault.
- Reserved if PR[qp] is 1 instructions (dark gray in the gray scale version of the tables, purple in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0.
- Reserved if PR[qp] is 1 B-unit instructions (medium gray in the gray scale version of the tables, cyan in the color version) cause an Illegal Operation fault if the predicate register specified by the qp field of the instruction (bits 5:0) is 1 and execute as a `nop` instruction if 0. These differ from the Reserved if PR[qp] is 1 instructions (purple) only in their RAW dependency behavior (see “RAW Dependency Table” on page 374).

Some processors may implement the Reserved if PR[qp] is 1 (purple) and Reserved if PR[qp] is 1 B-unit (cyan) encodings in the L+X opcode space as Reserved (brown). These encodings appear in the L+X column of Table 4-3 on page 3:183, and in Table 4-75 on page 3:257, Table 4-76 on page 3:257, Table 4-77 on page 3:258, and Table 4-78 on page 3:258. On processors which implement these encodings as Reserved (brown), the operating system is required to provide an Illegal Operation fault handler which emulates them as Reserved if PR[qp] is 1 (cyan/purple) by decoding the reserved opcodes, checking the qualifying predicate, and returning to the next instruction if PR[qp] is 0.
Constant 0 fields in instructions must be 0 or undefined operation results. The undefined operation may include checking that the constant field is 0 and causing an Illegal Operation fault if it is not. If an instruction having a constant 0 field also has a qualifying predicate (qp field), the fault or other undefined operation must not occur if PR[qp] is 0. For constant 0 fields in instruction bits 5:0 (normally used for qp), the fault or other undefined operation may or may not depend on the PR addressed by those bits.
Ignored (white space) fields in instructions should be coded as 0. Although ignored in this revision of the architecture, future architecture revisions may define these fields as hint extensions. These hint extensions will be defined such that the 0 value in each field corresponds to the default hint. It is expected that assemblers will automatically set these fields to zero by default.
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.

4.2 A-Unit Instruction Encodings

4.2.1 Integer ALU

All integer ALU instructions are encoded within major opcode 8 using a 2-bit opcode extension field in bits 35:34 (x_{2a}) and most have a second 2-bit opcode extension field in bits 28:27 (x_{2b}), a 4-bit opcode extension field in bits 32:29 (x_4), and a 1-bit reserved opcode extension field in bit 33 (v_e). Table 4-8 shows the 2-bit x_{2a} and 1-bit v_e assignments, Table 4-9 shows the integer ALU 4-bit+2-bit assignments, and Table 4-12 on page 3:194 shows the multimedia ALU 1-bit+2-bit assignments (which also share major opcode 8).

Table 4-8. Integer ALU 2-bit+1-bit Opcode Extensions

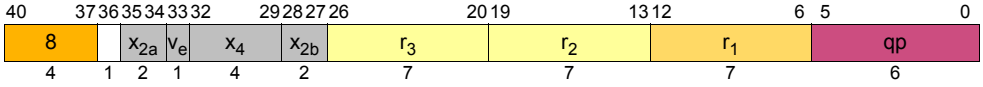
Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	
		0	1
8	0	Integer ALU 4-bit+2-bit Ext (Table 4-9)	
	1	Multimedia ALU 1-bit+2-bit Ext (Table 4-12)	
	2	adds – imm_{14} A4	
	3	addp4 – imm_{14} A4	

Table 4-9. Integer ALU 4-bit+2-bit Opcode Extensions

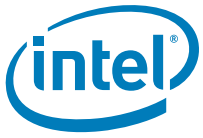
Opcode Bits 40:37	x_{2a} Bits 35:34	v_e Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
				0	1	2	3
8	0	0	0	add A1	add +1 A1		
			1	sub -1 A1	sub A1		
			2	addp4 A1			
			3	and A1	andcm A1	or A1	xor A1
			4	shladd A2			
			5				
			6	shladdp4 A2			
			7				
			8				
			9		sub – imm_8 A3		
			A				
			B	and – imm_8 A3	andcm – imm_8 A3	or – imm_8 A3	xor – imm_8 A3
			C				
			D				
			E				
			F				

4.2.1.1 Integer ALU – Register-Register

A1

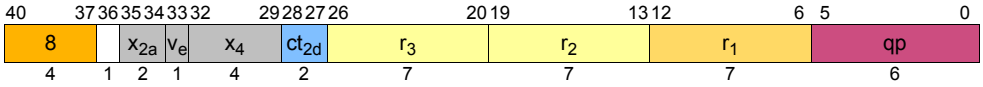


Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
add	$r_1 = r_2, r_3$	8	0	0	0	0
	$r_1 = r_2, r_3, 1$					1
sub	$r_1 = r_2, r_3$				1	1
	$r_1 = r_2, r_3, 1$					0
addp4	$r_1 = r_2, r_3$				2	0
and					3	0
andcm						1
or						2
xor						3



4.2.1.2 Shift Left and Add

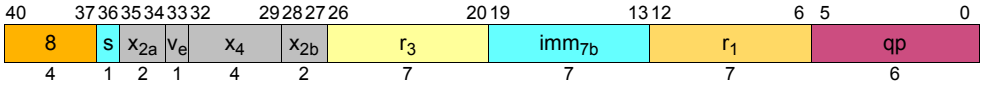
A2



Instruction	Operands	Opcode	Extension		
			x_{2a}	v_e	x_4
shladd shladdp4	$r_1 = r_2$, $count_2$, r_3	8	0	0	4 6

4.2.1.3 Integer ALU – Immediate$_8$-Register

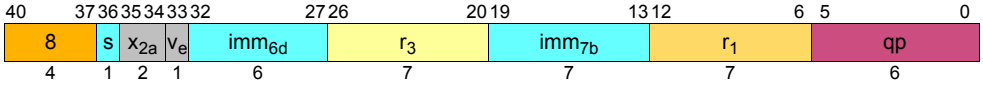
A3



Instruction	Operands	Opcode	Extension			
			x_{2a}	v_e	x_4	x_{2b}
sub	$r_1 = imm_8$, r_3	8	0	0	9	1
and					B	0
andcm						1
or						2
xor						3

4.2.1.4 Add Immediate$_{14}$

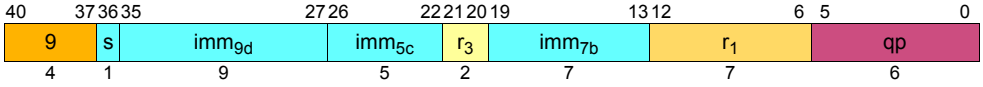
A4



Instruction	Operands	Opcode	Extension	
			x_{2a}	v_e
adds addp4	$r_1 = imm_{14}$, r_3	8	2 3	0

4.2.1.5 Add Immediate$_{22}$

A5



Instruction	Operands	Opcode
addl	$r_1 = imm_{22}$, r_3	9

4.2.2 Integer Compare

The integer compare instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and three 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), and 12 (c), as shown in Table 4-10. The integer compare immediate instructions are encoded within major opcodes C - E using a 2-bit opcode extension field (x_2) in bits 35:34 and two 1-bit opcode extension fields in bits 33 (t_a) and 12 (c), as shown in Table 4-11.

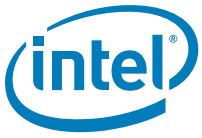


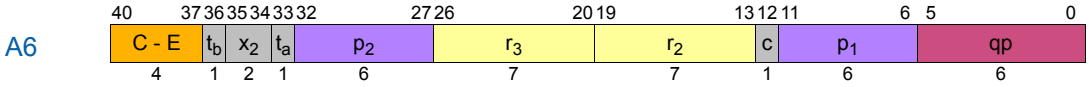
Table 4-10.Integer Compare Opcode Extensions

x_2 Bits 35:34	t_b Bit 36	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
				C	D	E
0	0	0	0	cmp.lt A6	cmp.ltu A6	cmp.eq A6
			1	cmp.lt.unc A6	cmp.ltu.unc A6	cmp.eq.unc A6
		1	0	cmp.eq.and A6	cmp.eq.or A6	cmp.eq.or.andcm A6
			1	cmp.ne.and A6	cmp.ne.or A6	cmp.ne.or.andcm A6
	1	0	0	cmp.gt.and A7	cmp.gt.or A7	cmp.gt.or.andcm A7
			1	cmp.le.and A7	cmp.le.or A7	cmp.le.or.andcm A7
		1	0	cmp.ge.and A7	cmp.ge.or A7	cmp.ge.or.andcm A7
			1	cmp.lt.and A7	cmp.lt.or A7	cmp.lt.or.andcm A7
1	0	0	0	cmp4.lt A6	cmp4.ltu A6	cmp4.eq A6
			1	cmp4.lt.unc A6	cmp4.ltu.unc A6	cmp4.eq.unc A6
		1	0	cmp4.eq.and A6	cmp4.eq.or A6	cmp4.eq.or.andcm A6
			1	cmp4.ne.and A6	cmp4.ne.or A6	cmp4.ne.or.andcm A6
	1	0	0	cmp4.gt.and A7	cmp4.gt.or A7	cmp4.gt.or.andcm A7
			1	cmp4.le.and A7	cmp4.le.or A7	cmp4.le.or.andcm A7
		1	0	cmp4.ge.and A7	cmp4.ge.or A7	cmp4.ge.or.andcm A7
			1	cmp4.lt.and A7	cmp4.lt.or A7	cmp4.lt.or.andcm A7

Table 4-11.Integer Compare Immediate Opcode Extensions

x_2 Bits 35:34	t_a Bit 33	c Bit 12	Opcode Bits 40:37		
			C	D	E
2	0	0	cmp.lt – imm_8 A8	cmp.ltu – imm_8 A8	cmp.eq – imm_8 A8
		1	cmp.lt.unc – imm_8 A8	cmp.ltu.unc – imm_8 A8	cmp.eq.unc – imm_8 A8
	1	0	cmp.eq.and – imm_8 A8	cmp.eq.or – imm_8 A8	cmp.eq.or.andcm – imm_8 A8
		1	cmp.ne.and – imm_8 A8	cmp.ne.or – imm_8 A8	cmp.ne.or.andcm – imm_8 A8
3	0	0	cmp4.lt – imm_8 A8	cmp4.ltu – imm_8 A8	cmp4.eq – imm_8 A8
		1	cmp4.lt.unc – imm_8 A8	cmp4.ltu.unc – imm_8 A8	cmp4.eq.unc – imm_8 A8
	1	0	cmp4.eq.and – imm_8 A8	cmp4.eq.or – imm_8 A8	cmp4.eq.or.andcm – imm_8 A8
		1	cmp4.ne.and – imm_8 A8	cmp4.ne.or – imm_8 A8	cmp4.ne.or.andcm – imm_8 A8

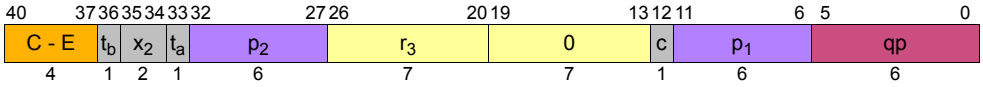
4.2.2.1 Integer Compare – Register-Register



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.lt	$p_1, p_2 = r_2, r_3$	C	0	0	0	0
cmp.ltu		D				
cmp.eq		E				
cmp.lt.unc		C				1
cmp.ltu.unc		D				
cmp.eq.unc		E				
cmp.eq.and		C			1	0
cmp.eq.or		D				
cmp.eq.or.andcm		E				
cmp.ne.and		C				1
cmp.ne.or		D				
cmp.ne.or.andcm		E				
cmp4.lt		C	1	0	0	0
cmp4.ltu		D				
cmp4.eq		E				
cmp4.lt.unc		C				1
cmp4.ltu.unc		D				
cmp4.eq.unc		E				
cmp4.eq.and		C			1	0
cmp4.eq.or		D				
cmp4.eq.or.andcm		E				
cmp4.ne.and		C				1
cmp4.ne.or		D				
cmp4.ne.or.andcm		E				

4.2.2.2 Integer Compare to Zero – Register

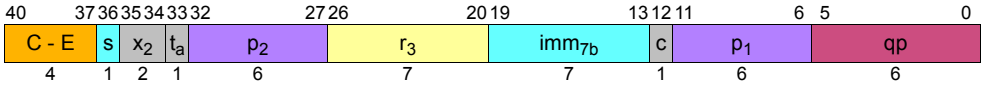
A7



Instruction	Operands	Opcode	Extension			
			x_2	t_b	t_a	c
cmp.gt.and	p_1, p_2 = r0, r_3	C	0	1	0	0
cmp.gt.or		D				
cmp.gt.or.andcm		E				
cmp.le.and		C				1
cmp.le.or		D				
cmp.le.or.andcm		E				
cmp.ge.and		C			1	0
cmp.ge.or		D				
cmp.ge.or.andcm		E				
cmp.lt.and		C				1
cmp.lt.or		D				
cmp.lt.or.andcm		E				
cmp4.gt.and		C	1		0	0
cmp4.gt.or		D				
cmp4.gt.or.andcm		E				
cmp4.le.and		C				1
cmp4.le.or		D				
cmp4.le.or.andcm		E				
cmp4.ge.and		C			1	0
cmp4.ge.or		D				
cmp4.ge.or.andcm		E				
cmp4.lt.and		C				1
cmp4.lt.or		D				
cmp4.lt.or.andcm		E				

4.2.2.3 Integer Compare – Immediate-Register

A8



Instruction	Operands	Opcode	Extension		
			x_2	t_a	c
cmp.lt	$p_1, p_2 = imm_8, r_3$	C	2	0	0
cmp.ltu		D			
cmp.eq		E			
cmp.lt.unc		C			1
cmp.ltu.unc		D			
cmp.eq.unc		E			
cmp.eq.and		C		1	0
cmp.eq.or		D			
cmp.eq.or.andcm		E			
cmp.ne.and		C			1
cmp.ne.or		D			
cmp.ne.or.andcm		E			
cmp4.lt		C	3	0	0
cmp4.ltu		D			
cmp4.eq		E			
cmp4.lt.unc		C			1
cmp4.ltu.unc		D			
cmp4.eq.unc		E			
cmp4.eq.and		C		1	0
cmp4.eq.or		D			
cmp4.eq.or.andcm		E			
cmp4.ne.and		C			1
cmp4.ne.or		D			
cmp4.ne.or.andcm		E			

4.2.3 Multimedia

All multimedia ALU instructions are encoded within major opcode 8 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 2-bit opcode extension field in bits 35:34 (x_{2a}) as shown in Table 4-12. The multimedia ALU instructions also have a 4-bit opcode extension field in bits 32:29 (x_4), and a 2-bit opcode extension field in bits 28:27 (x_{2b}) as shown in Table 4-13 on page 3:195.

Table 4-12.Multimedia ALU 2-bit+1-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	
8	1	0	0	Multimedia ALU Size 1 (Table 4-13)
			1	Multimedia ALU Size 2 (Table 4-14)
		1	0	Multimedia ALU Size 4 (Table 4-15)
			1	

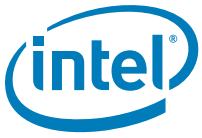


Table 4-13.Multimedia ALU Size 1 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	0	0	padd1 A9	padd1.sss A9	padd1.uuu A9	padd1.uus A9
				1	psub1 A9	psub1.sss A9	psub1.uuu A9	psub1.uus A9
				2			pavg1 A9	pavg1.raz A9
				3			pavgsub1 A9	
				4				
				5				
				6				
				7				
				8				
				9	pcmp1.eq A9	pcmp1.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

Table 4-14.Multimedia ALU Size 2 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	0	1	0	padd2 A9	padd2.sss A9	padd2.uuu A9	padd2.uus A9
				1	psub2 A9	psub2.sss A9	psub2.uuu A9	psub2.uus A9
				2			pavg2 A9	pavg2.raz A9
				3			pavgsub2 A9	
				4	pshladd2 A10			
				5				
				6	pshradd2 A10			
				7				
				8				
				9	pcmp2.eq A9	pcmp2.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

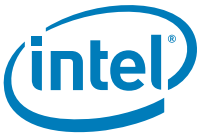
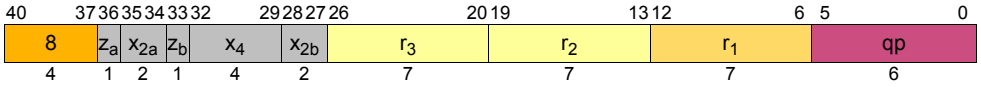


Table 4-15.Multimedia ALU Size 4 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_{2a} Bits 35:34	z_a Bit 36	z_b Bit 33	x_4 Bits 32:29	x_{2b} Bits 28:27			
					0	1	2	3
8	1	1	0	0	padd4 A9			
				1	psub4 A9			
				2				
				3				
				4				
				5				
				6				
				7				
				8				
				9	pcmp4.eq A9	pcmp4.gt A9		
				A				
				B				
				C				
				D				
				E				
				F				

4.2.3.1 Multimedia ALU

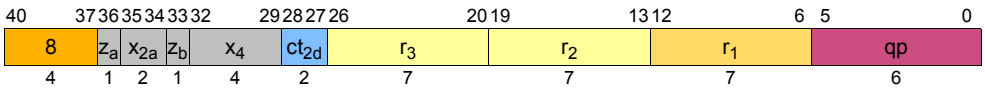
A9



Instruction	Operands	Opcode	Extension				
			x_{2a}	z_a	z_b	x_4	x_{2b}
padd1	$r_1 = r_2, r_3$	8	1	0	0	0	0
padd2					1		
padd4				1	0		
padd1.sss				0	0	0	1
padd2.sss					1		
padd1.uuu				0	0		2
padd2.uuu					1		
padd1.uus				0	0		3
padd2.uus					1		
psub1				0	0	1	0
psub2					1		
psub4				1	0		
psub1.sss				0	0		1
psub2.sss					1		
psub1.uuu				0	0		2
psub2.uuu					1		
psub1.uus				0	0		3
psub2.uus					1		
pavg1				0	0	2	2
pavg2					1		
pavg1.raz				0	0		3
pavg2.raz					1		
pavgsub1				0	0	3	2
pavgsub2					1		
pcmp1.eq				0	0	9	0
pcmp2.eq					1		
pcmp4.eq				1	0		
pcmp1.gt				0	0		1
pcmp2.gt					1		
pcmp4.gt				1	0		

4.2.3.2 Multimedia Shift and Add

A10



Instruction	Operands	Opcode	Extension			
			x_{2a}	z_a	z_b	x_4
pshladd2	$r_1 = r_2, count_2, r_3$	8	1	0	1	4
pshradd2						6

4.3 I-Unit Instruction Encodings

4.3.1 Multimedia and Variable Shifts

All multimedia multiply/shift/max/min/mix/mux/pack/unpack and variable shift instructions are encoded within major opcode 7 using two 1-bit opcode extension fields in bits 36 (z_a) and 33 (z_b) and a 1-bit reserved opcode extension in bit 32 (v_e) as shown in Table 4-16. They also have a 2-bit opcode extension field in bits 35:34 (x_{2a}) and a 2-bit field in bits 29:28 (x_{2b}) and most have a 2-bit field in bits 31:30 (x_{2c}) as shown in Table 4-17.

Table 4-16.Multimedia and Variable Shift 1-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	
			0	1
7	0	0	Multimedia Size 1 (Table 4-17)	
		1	Multimedia Size 2 (Table 4-18)	
	1	0	Multimedia Size 4 (Table 4-19)	
		1	Variable Shift (Table 4-20)	

Table 4-17.Multimedia Opcode 7 Size 1 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	0	0	0	0				
					1				
					2				
					3				
				1	0				
					1				
					2				
					3				
				2	0		unpack1.h I2	mix1.r I2	
					1	pmin1.u I2	pmax1.u I2		
					2		unpack1.l I2	mix1.l I2	
					3			psad1 I2	
				3	0				
					1				
					2			mux1 I3	
					3				

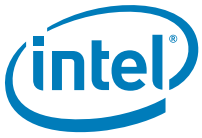


Table 4-18.Multimedia Opcode 7 Size 2 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	0	1	0	0	0	pshr2.u – var I5	pshl2 – var I7		
					1	pmpyshr2.u I1			
					2	pshr2 – var I5			
					3	pmpyshr2 I1			
				1	0				
					1	pshr2.u – fixed I6		popcnt I9	clz I9
					2				
					3	pshr2 – fixed I6			
				2	0	pack2.uss I2	unpack2.h I2	mix2.r I2	
					1				pmpy2.r I2
					2	pack2.sss I2	unpack2.l I2	mix2.l I2	
					3	pmin2 I2	pmax2 I2		pmpy2.l I2
				3	0				
					1		pshl2 – fixed I8		
					2			mux2 I4	
					3				

Table 4-19.Multimedia Opcode 7 Size 4 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	0	0	0	0	pshr4.u – var I5	pshl4 – var I7		
					1				mpy4 I2
					2	pshr4 – var I5			
					3				mpyshl4 I2
				1	0				
					1	pshr4.u – fixed I6			
					2				
					3	pshr4 – fixed I6			
				2	0		unpack4.h I2	mix4.r I2	
					1				
					2	pack4.sss I2	unpack4.l I2	mix4.l I2	
					3				
				3	0				
					1		pshl4 – fixed I8		
					2				
					3				

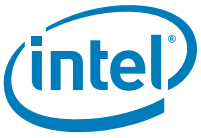
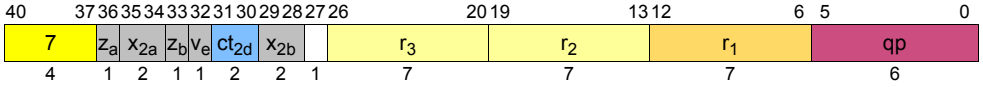


Table 4-20.Variable Shift Opcode 7 2-bit Opcode Extensions

Opcode Bits 40:37	z_a Bit 36	z_b Bit 33	v_e Bit 32	x_{2a} Bits 35:34	x_{2b} Bits 29:28	x_{2c} Bits 31:30			
						0	1	2	3
7	1	1	0	0	0	shr.u – var I5	shl – var I7		
					1				
					2	shr – var I5			
					3				
				1	0				
					1				
					2				
					3				
				2	0				
					1				
					2				
					3				
				3	0				
					1				
					2				
					3				

4.3.1.1 Multimedia Multiply and Shift

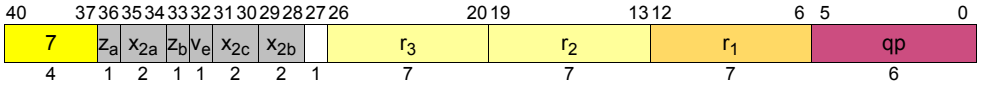
I1



Instruction	Operands	Opcode	Extension				
			z_a	z_b	v_e	x_{2a}	x_{2b}
pmpyshr2 pmpyshr2.u	$r_1 = r_2, r_3, count_2$	7	0	1	0	0	3 1

4.3.1.2 Multimedia Multiply/Mix/Pack/Unpack

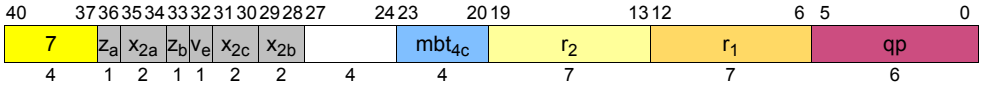
I2



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mpy4 mpyshl4	$r_1 = r_2, r_3$	7	1	0	0	0	1 3	3
pmpy2.r pmpy2.l			0	1		2	1 3	3
mix1.r mix2.r mix4.r			0 0 1	0 1 0			0	2
mix1.l mix2.l mix4.l			0 0 1	0 1 0			2	
pack2.uss			0	1			0	0
pack2.sss pack4.sss			0 1	1 0			2	
unpack1.h unpack2.h unpack4.h			0 0 1	0 1 0			0	1
unpack1.l unpack2.l unpack4.l			0 0 1	0 1 0			2	
pmin1.u pmax1.u			0	0			1	0 1
pmin2 pmax2			0	1			3	0 1
psad1			0	0			3	2

4.3.1.3 Multimedia Mux1

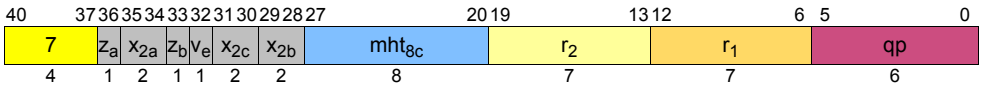
I3



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux1	$r_1 = r_2, mbtype_4$	7	0	0	0	3	2	2

4.3.1.4 Multimedia Mux2

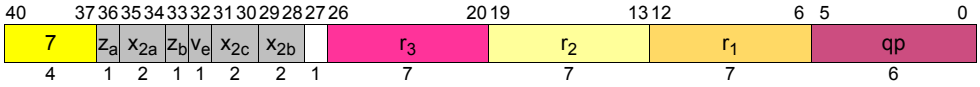
I4



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
mux2	$r_1 = r_2, mhtype_8$	7	0	1	0	3	2	2

4.3.1.5 Shift Right – Variable

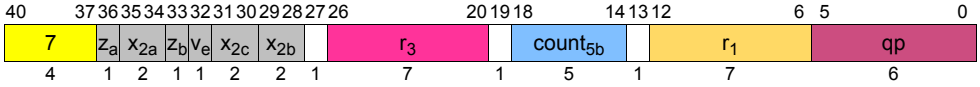
I5



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, r_2$	7	0	1	0	0	2	0
pshr4			1	0				
shr			1	1				
pshr2.u			0	1			0	
pshr4.u			1	0				
shr.u			1	1				

4.3.1.6 Multimedia Shift Right – Fixed

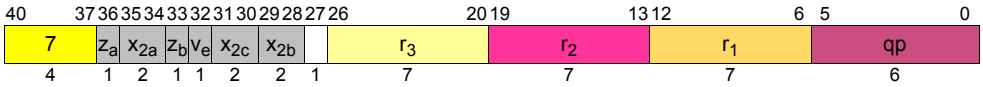
I6



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshr2	$r_1 = r_3, count_5$	7	0	1	0	1	3	0
pshr4			1	0				
pshr2.u			0	1			1	
pshr4.u			1	0				

4.3.1.7 Shift Left – Variable

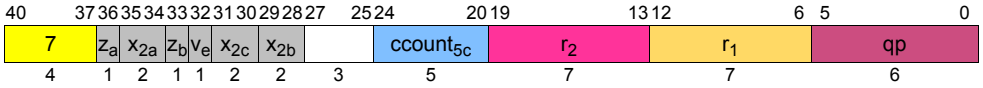
I7



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, r_3$	7	0	1	0	0	0	1
pshl4			1	0				
shl			1	1				

4.3.1.8 Multimedia Shift Left – Fixed

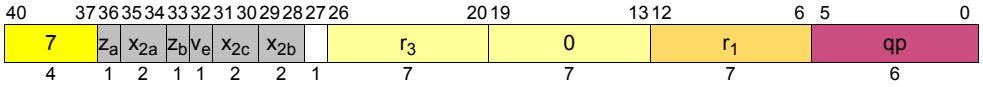
I8



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
pshl2	$r_1 = r_2, count_5$	7	0	1	0	3	1	1
pshl4			1	0				

4.3.1.9 Bit Strings

I9



Instruction	Operands	Opcode	Extension					
			z_a	z_b	v_e	x_{2a}	x_{2b}	x_{2c}
popcnt	$r_1 = r_3$	7	0	1	0	1	1	2
clz								3

4.3.2 Integer Shifts

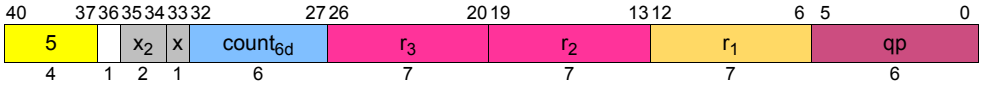
The integer shift, test bit, and test NaT instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) and a 1-bit opcode extension field in bit 33 (x). The extract and test bit instructions also have a 1-bit opcode extension field in bit 13 (y). Table 4-21 shows the test bit, extract, and shift right pair assignments.
Most deposit instructions also have a 1-bit opcode extension field in bit 26 (y). Table 4-22 shows these assignments.

Table 4-22.Deposit Opcode Extensions

Opcode Bits 40:37	x_2 Bits 35:34	x Bit 33	y Bit 26	
			0	1
5	0	1	Test Bit/Test NaT/Test Feature (Table 4-23)	
	1		dep.z I12	dep.z – imm_8 I13
	2			
	3		dep – imm_1 I14	

4.3.2.1 Shift Right Pair

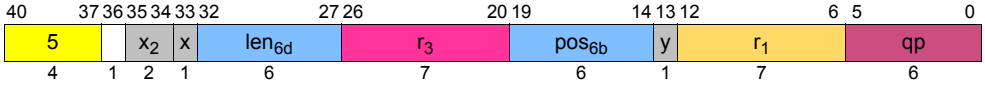
I10



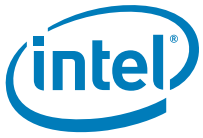
Instruction	Operands	Opcode	Extension	
			x_2	x
shrp	$r_1 = r_2, r_3, count_6$	5	3	0

4.3.2.2 Extract

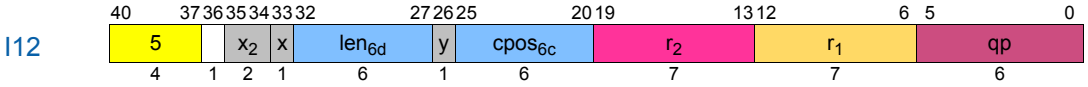
I11



Instruction	Operands	Opcode	Extension		
			x_2	x	y
extr.u	$r_1 = r_3, pos_6, len_6$	5	1	0	0
extr					1

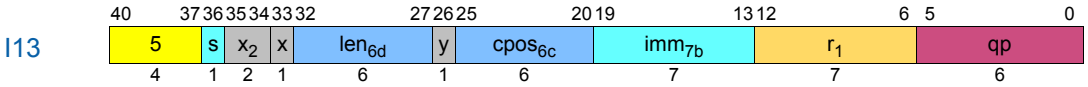


4.3.2.3 Zero and Deposit



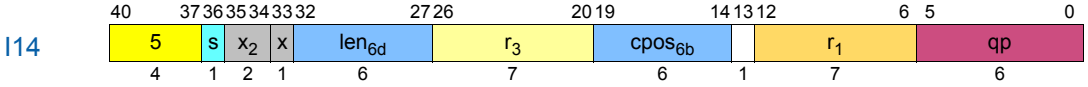
Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = r_2, pos_6, len_6$	5	1	1	0

4.3.2.4 Zero and Deposit Immediate$_8$



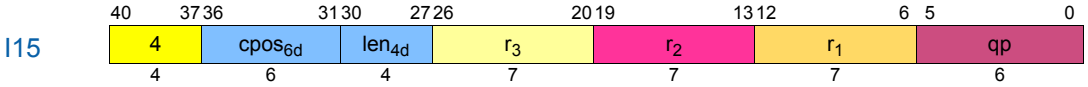
Instruction	Operands	Opcode	Extension		
			x_2	x	y
dep.z	$r_1 = imm_8, pos_6, len_6$	5	1	1	1

4.3.2.5 Deposit Immediate$_1$



Instruction	Operands	Opcode	Extension	
			x_2	x
dep	$r_1 = imm_1, r_3, pos_6, len_6$	5	3	1

4.3.2.6 Deposit



Instruction	Operands	Opcode
dep	$r_1 = r_2, r_3, pos_6, len_4$	4

4.3.3 Test Bit

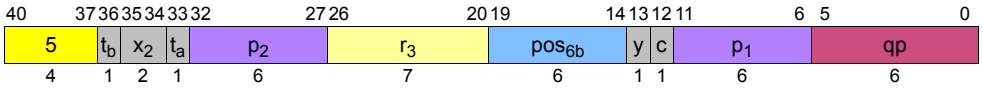
All test bit instructions are encoded within major opcode 5 using a 2-bit opcode extension field in bits 35:34 (x_2) plus five 1-bit opcode extension fields in bits 33 (t_a), 36 (t_b), 12 (c), 13 (y) and 19 (x). Table 4-23 summarizes these assignments.

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 I16	
					1	tnat.z I17	tf.z I30
				1	0	tbit.z.unc I16	
					1	tnat.z.unc I17	tf.z.unc I30
			1	0	0	tbit.z.and I16	
					1	tnat.z.and I17	tf.z.and I30
				1	0	tbit.nz.and I16	
					1	tnat.nz.and I17	tf.nz.and I30
		1	0	0	0	tbit.z.or I16	
					1	tnat.z.or I17	tf.z.or I30
				1	0	tbit.nz.or I16	
					1	tnat.nz.or I17	tf.nz.or I30
			1	0	0	tbit.z.or.andcm I16	
					1	tnat.z.or.andcm I17	tf.z.or.andcm I30
				1	0	tbit.nz.or.andcm I16	
					1	tnat.nz.or.andcm I17	tf.nz.or.andcm I30

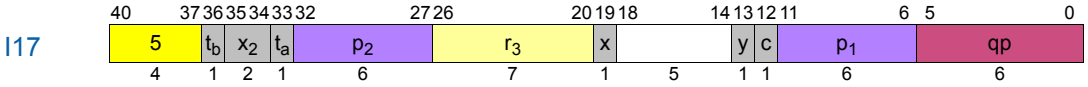
4.3.3.1 Test Bit

I16



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

4.3.4 Miscellaneous I-Unit Instructions

The miscellaneous I-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33. Some also have a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-24 shows the 3-bit assignments and Table 4-25 summarizes the 6-bit assignments.

Table 4-24.Misc I-Unit 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-25)
	1	chk.s.i – int I20
	2	mov to pr.rot – imm_{44} I24
	3	mov to pr I23
	4	
	5	
	6	
	7	mov to b I21

Table 4-25.Misc I-Unit 6-bit Opcode Extensions

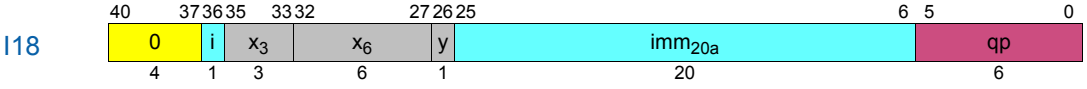
Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	x_6 Bits 32:31 = 1	x_6 Bits 32:31 = 2	x_6 Bits 32:31 = 3
0	0	0	break.i I19	zxt1 I29		mov from ip I25
		1	1-bit Ext (Table 4-26)	zxt2 I29		mov from b I22
		2		zxt4 I29		mov.i from ar I28
		3				mov from pr I25
		4		sxt1 I29		
		5		sxt2 I29		
		6		sxt4 I29		
		7				
		8		czx1.l I29		
		9		czx2.l I29		
		A	mov.i to ar – imm_8 I27		mov.i to ar I26	
		B				
		C		czx1.r I29		
		D		czx2.r I29		
		E				
		F				

4.3.4.1 Nop/Hint (I-Unit)

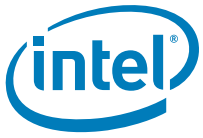
I-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-26.

Table 4-26.Misc I-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.i
			1	hint.i

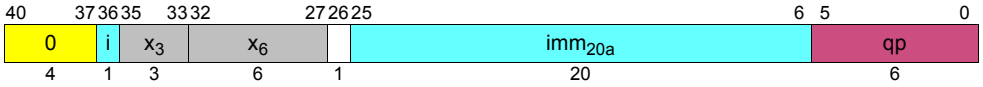


Instruction	Operands	Opcode	Extension x_3	Extension x_6	Extension y
nop.i [i] hint.i	imm_{21}	0	0	01	0 1



4.3.4.2 Break (I-Unit)

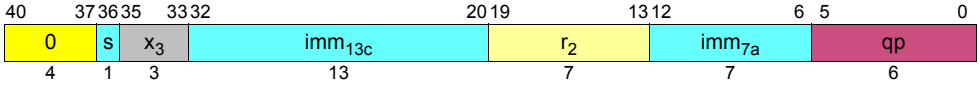
I19



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.i [i]	*imm*$_{21}$	0	0	00

4.3.4.3 Integer Speculation Check (I-Unit)

I20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.i	r_2, *target*$_{25}$	0	1

4.3.5 GR/BR Moves

The GR/BR move instructions are encoded in major opcode 0. See “Miscellaneous I-Unit Instructions” on page 206 for a summary of the opcode extensions. The mov to BR instruction uses a 2-bit “whether” prediction hint field in bits 21:20 (wh) as shown in Table 4-27.

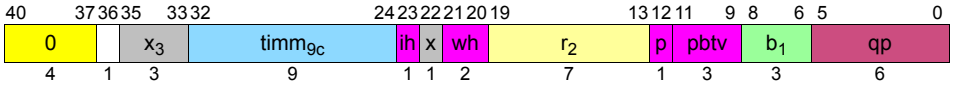
Table 4-27.Move to BR Whether Hint Completer

wh Bits 21:20	*mwh*
0	.sptk
1	*none*
2	.dptk
3	

The mov to BR instruction also uses a 1-bit opcode extension field (x) in bit 22 to distinguish the return form from the normal form, and a 1-bit hint extension in bit 23 (ih) (see Table 4-60 on page 3:244).

4.3.5.1 Move to BR

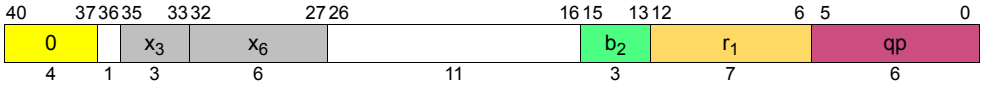
I21



Instruction	Operands	Opcode	Extension					
			x_3	x	ih	p	wh	pbtv
mov.*mwh.ph.pvec.ih*	$b_1 = r_2$, *tag*$_{13}$	0	7	0	See Table 4-60 on page 3:244	See Table 4-63 on page 3:245	See Table 4-27 on page 3:208	See Table 4-64 on page 3:245
mov.ret.*mwh.ph.pvec.ih*				1				

4.3.5.2 Move from BR

I22



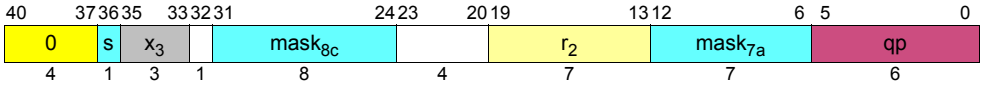
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	$r_1 = b_2$	0	0	31

4.3.6 GR/Predicate/IP Moves

The GR/Predicate/IP move instructions are encoded in major opcode 0. See “Miscellaneous I-Unit Instructions” on page 206 for a summary of the opcode extensions.

4.3.6.1 Move to Predicates – Register

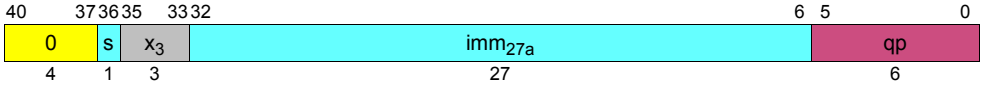
I23



Instruction	Operands	Opcode	Extension
			x_3
mov	pr = r_2, $mask_{17}$	0	3

4.3.6.2 Move to Predicates – Immediate$_{44}$

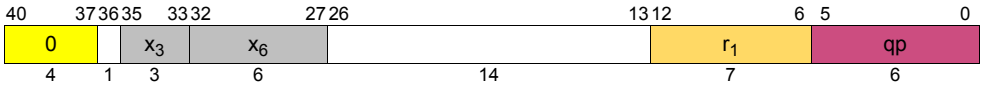
I24



Instruction	Operands	Opcode	Extension
			x_3
mov	pr.rot = imm_{44}	0	2

4.3.6.3 Move from Predicates/IP

I25



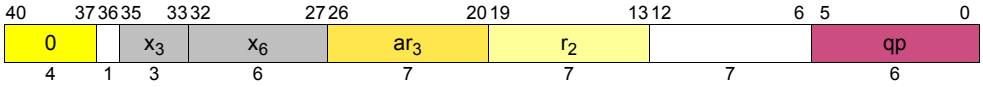
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov	r_1 = ip r_1 = pr	0	0	30 33

4.3.7 GR/AR Moves (I-Unit)

The I-Unit GR/AR move instructions are encoded in major opcode 0. (Some ARs are accessed using system/memory management instructions on the M-unit. See “GR/AR Moves (M-Unit)” on page 232.) See “Miscellaneous I-Unit Instructions” on page 206 for a summary of the I-Unit GR/AR opcode extensions.

4.3.7.1 Move to AR – Register (I-Unit)

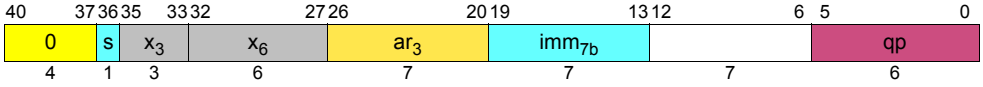
I26



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = r_2$	0	0	2A

4.3.7.2 Move to AR – Immediate$_8$ (I-Unit)

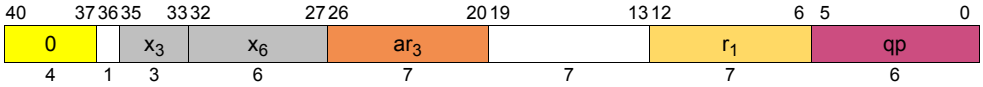
I27



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$ar_3 = imm_8$	0	0	0A

4.3.7.3 Move from AR (I-Unit)

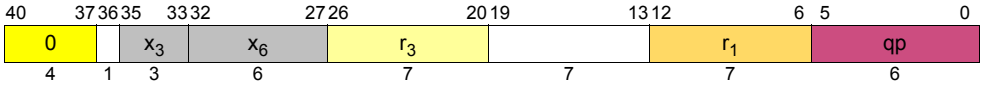
I28



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.i	$r_1 = ar_3$	0	0	32

4.3.8 Sign/Zero Extend/Compute Zero Index

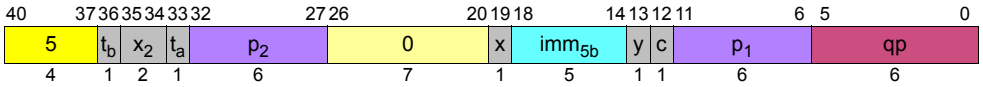
I29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
zxt1	$r_1 = r_3$	0	0	10
zxt2				11
zxt4				12
sxt1				14
sxt2				15
sxt4				16
czx1.l				18
czx2.l				19
czx1.r				1C
czx2.r				1D

4.3.9 Test Feature

I30



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

4.4 M-Unit Instruction Encodings

4.4.1 Loads and Stores

All load and store instructions are encoded within major opcodes 4, 5, 6, and 7 using a 6-bit opcode extension field in bits 35:30 (x_6). Instructions in major opcode 4 (integer load/store, semaphores, and get FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-28. Instructions in major opcode 6 (floating-point load/store, load pair, and set FR) use two 1-bit opcode extension fields in bit 36 (m) and bit 27 (x) as shown in Table 4-29.

Table 4-28.Integer Load/Store/Semaphore/Get FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
4	0	0	Load/Store (Table 4-30)
	0	1	Semaphore/get FR (Table 4-33)
	1	0	Load +Reg (Table 4-31)
	1	1	

Table 4-29.Floating-point Load/Store/Load Pair/Set FR 1-bit Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	
6	0	0	FP Load/Store (Table 4-34)
	0	1	FP Load Pair/set FR (Table 4-38)
	1	0	FP Load +Reg (Table 4-36)
	1	1	FP Load Pair +Imm (Table 4-39)

The integer load/store opcode extensions are summarized in Table 4-30, Table 4-31, and Table 4-32 on page 3:213, and the semaphore and get FR opcode extensions in Table 4-33. The floating-point load/store opcode extensions are summarized in Table 4-34, Table 4-36, and Table 4-37, the floating-point load pair and set FR opcode extensions in Table 4-38 and Table 4-39.

Table 4-30.Integer Load/Store Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	0	0	ld1 M1	ld2 M1	ld4 M1	ld8 M1
			1	ld1.s M1	ld2.s M1	ld4.s M1	ld8.s M1
			2	ld1.a M1	ld2.a M1	ld4.a M1	ld8.a M1
			3	ld1.sa M1	ld2.sa M1	ld4.sa M1	ld8.sa M1
			4	ld1.bias M1	ld2.bias M1	ld4.bias M1	ld8.bias M1
			5	ld1.acq M1	ld2.acq M1	ld4.acq M1	ld8.acq M1
			6				ld8.fill M1
			7				
			8	ld1.c.clr M1	ld2.c.clr M1	ld4.c.clr M1	ld8.c.clr M1
			9	ld1.c.nc M1	ld2.c.nc M1	ld4.c.nc M1	ld8.c.nc M1
			A	ld1.c.clr.acq M1	ld2.c.clr.acq M1	ld4.c.clr.acq M1	ld8.c.clr.acq M1
			B				
			C	st1 M4	st2 M4	st4 M4	st8 M4
			D	st1.rel M4	st2.rel M4	st4.rel M4	st8.rel M4
			E				st8.spill M4
			F				

Table 4-31.Integer Load +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	1	0	0	ld1 M2	ld2 M2	ld4 M2	ld8 M2
			1	ld1.s M2	ld2.s M2	ld4.s M2	ld8.s M2
			2	ld1.a M2	ld2.a M2	ld4.a M2	ld8.a M2
			3	ld1.sa M2	ld2.sa M2	ld4.sa M2	ld8.sa M2
			4	ld1.bias M2	ld2.bias M2	ld4.bias M2	ld8.bias M2
			5	ld1.acq M2	ld2.acq M2	ld4.acq M2	ld8.acq M2
			6				ld8.fill M2
			7				
			8	ld1.c.clr M2	ld2.c.clr M2	ld4.c.clr M2	ld8.c.clr M2
			9	ld1.c.nc M2	ld2.c.nc M2	ld4.c.nc M2	ld8.c.nc M2
			A	ld1.c.clr.acq M2	ld2.c.clr.acq M2	ld4.c.clr.acq M2	ld8.c.clr.acq M2
			B				
			C				
			D				
			E				
			F				

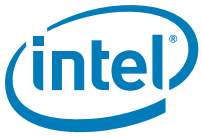


Table 4-32.Integer Load/Store +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
5	0	ld1 M3	ld2 M3	ld4 M3	ld8 M3
	1	ld1.s M3	ld2.s M3	ld4.s M3	ld8.s M3
	2	ld1.a M3	ld2.a M3	ld4.a M3	ld8.a M3
	3	ld1.sa M3	ld2.sa M3	ld4.sa M3	ld8.sa M3
	4	ld1.bias M3	ld2.bias M3	ld4.bias M3	ld8.bias M3
	5	ld1.acq M3	ld2.acq M3	ld4.acq M3	ld8.acq M3
	6				ld8.fill M3
	7				
	8	ld1.c.clr M3	ld2.c.clr M3	ld4.c.clr M3	ld8.c.clr M3
	9	ld1.c.nc M3	ld2.c.nc M3	ld4.c.nc M3	ld8.c.nc M3
	A	ld1.c.clr.acq M3	ld2.c.clr.acq M3	ld4.c.clr.acq M3	ld8.c.clr.acq M3
	B				
	C	st1 M5	st2 M5	st4 M5	st8 M5
	D	st1.rel M5	st2.rel M5	st4.rel M5	st8.rel M5
	E				st8.spill M5
	F				

Table 4-33.Semaphore/Get FR/16-Byte Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
4	0	1	0	cmpxchg1.acq M16	cmpxchg2.acq M16	cmpxchg4.acq M16	cmpxchg8.acq M16
			1	cmpxchg1.rel M16	cmpxchg2.rel M16	cmpxchg4.rel M16	cmpxchg8.rel M16
			2	xchg1 M16	xchg2 M16	xchg4 M16	xchg8 M16
			3				
			4			fetchadd4.acq M17	fetchadd8.acq M17
			5			fetchadd4.rel M17	fetchadd8.rel M17
			6				
			7	getf.sig M19	getf.exp M19	getf.s M19	getf.d M19
			8	cmp8xchg16.acq M16			
			9	cmp8xchg16.rel M16			
			A	ld16 M1			
			B	ld16.acq M1			
			C	st16 M4			
			D	st16.rel M4			
			E				
			F				

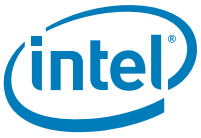


Table 4-34.Floating-point Load/Store/Lfetch Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	0	0	0	ldfe M6	ldf8 M6	ldfs M6	ldfd M6
			1	ldfe.s M6	ldf8.s M6	ldfs.s M6	ldfd.s M6
			2	ldfe.a M6	ldf8.a M6	ldfs.a M6	ldfd.a M6
			3	ldfe.sa M6	ldf8.sa M6	ldfs.sa M6	ldfd.sa M6
			4				
			5				
			6				ldf.fill M6
			7				
			8	ldfe.c.clr M6	ldf8.c.clr M6	ldfs.c.clr M6	ldfd.c.clr M6
			9	ldfe.c.nc M6	ldf8.c.nc M6	ldfs.c.nc M6	ldfd.c.nc M6
			A				
			B	lfetch (Table 4-35)	lfetch.excl M13	lfetch.fault M13	lfetch.fault.excl M13
			C	stfe M9	stf8 M9	stfs M9	stfd M9
			D				
			E				stf.spill M9
			F				

Table 4-35.Lfetch Extensions

y Bit 19	
0	lfetch M51
1	lfetch.count M52

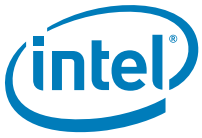


Table 4-36.Floating-point Load/Lfetch +Reg Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	x_6				
			Bits 35:32	Bits 31:30			
				0	1	2	3
6	1	0	0	ldfe M7	ldf8 M7	ldfs M7	ldfd M7
			1	ldfe.s M7	ldf8.s M7	ldfs.s M7	ldfd.s M7
			2	ldfe.a M7	ldf8.a M7	ldfs.a M7	ldfd.a M7
			3	ldfe.sa M7	ldf8.sa M7	ldfs.sa M7	ldfd.sa M7
			4				
			5				
			6				ldf.fill M7
			7				
			8	ldfe.c.clr M7	ldf8.c.clr M7	ldfs.c.clr M7	ldfd.c.clr M7
			9	ldfe.c.nc M7	ldf8.c.nc M7	ldfs.c.nc M7	ldfd.c.nc M7
			A				
			B	lfetch M14	lfetch.excl M14	lfetch.fault M14	lfetch.fault.excl M14
			C				
			D				
			E				
			F				

Table 4-37.Floating-point Load/Store/Lfetch +Imm Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 35:32	Bits 31:30			
		0	1	2	3
7	0	ldfe M8	ldf8 M8	ldfs M8	ldfd M8
	1	ldfe.s M8	ldf8.s M8	ldfs.s M8	ldfd.s M8
	2	ldfe.a M8	ldf8.a M8	ldfs.a M8	ldfd.a M8
	3	ldfe.sa M8	ldf8.sa M8	ldfs.sa M8	ldfd.sa M8
	4				
	5				
	6				ldf.fill M8
	7				
	8	ldfe.c.clr M8	ldf8.c.clr M8	ldfs.c.clr M8	ldfd.c.clr M8
	9	ldfe.c.nc M8	ldf8.c.nc M8	ldfs.c.nc M8	ldfd.c.nc M8
	A				
	B	lfetch M15	lfetch.excl M15	lfetch.fault M15	lfetch.fault.excl M15
	C	stfe M10	stf8 M10	stfs M10	stfd M10
	D				
	E				stf.spill M10
	F				

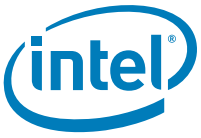


Table 4-38.Floating-point Load Pair/Set FR Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	Bits 35:32	x_6 Bits 31:30 = 0	x_6 = 1	x_6 = 2	x_6 = 3
6	0	1	0		ldfp8 M11	ldfps M11	ldfpd M11
			1		ldfp8.s M11	ldfps.s M11	ldfpd.s M11
			2		ldfp8.a M11	ldfps.a M11	ldfpd.a M11
			3		ldfp8.sa M11	ldfps.sa M11	ldfpd.sa M11
			4				
			5				
			6				
			7	setf.sig M18	setf.exp M18	setf.s M18	setf.d M18
			8		ldfp8.c.clr M11	ldfps.c.clr M11	ldfpd.c.clr M11
			9		ldfp8.c.nc M11	ldfps.c.nc M11	ldfpd.c.nc M11
			A				
			B				
			C				
			D				
			E				
			F				

Table 4-39.Floating-point Load Pair +Imm Opcode Extensions

Opcode Bits 40:37	m Bit 36	x Bit 27	Bits 35:32	x_6 Bits 31:30 = 0	x_6 = 1	x_6 = 2	x_6 = 3
6	1	1	0		ldfp8 M12	ldfps M12	ldfpd M12
			1		ldfp8.s M12	ldfps.s M12	ldfpd.s M12
			2		ldfp8.a M12	ldfps.a M12	ldfpd.a M12
			3		ldfp8.sa M12	ldfps.sa M12	ldfpd.sa M12
			4				
			5				
			6				
			7				
			8		ldfp8.c.clr M12	ldfps.c.clr M12	ldfpd.c.clr M12
			9		ldfp8.c.nc M12	ldfps.c.nc M12	ldfpd.c.nc M12
			A				
			B				
			C				
			D				
			E				
			F				

The load and store instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint). Table 4-39 and Table 4-42 summarize these assignments.

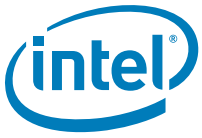


Table 4-40.Load Hint Completer

hint Bits 29: 28	*ldhint*
0	*none*
1	.nt1
2	
3	.nta

Table 4-41.Load Hint Completer for no-base-update forms

h Bit 19	hint Bits 29:28	*ldhintx*
0	0	*none*
	1	.nt1
	2	.d2
	3	.nta

Table 4-42.Load Hint Completer for base-update forms

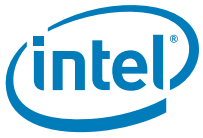
hint Bits 29:28	*ldhint*
0	*none*
1	.nt1
2	.d2
3	.nta

Table 4-43.Store Hint Completer

hint Bits 29:28	*sthint*
0	*none*
1	
2	
3	.nta

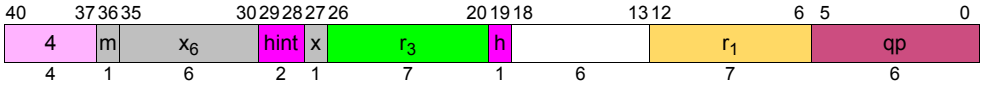
Table 4-44.Store Hint Completer

hint Bits 29:28	*sthint*
0	*none*
1	.d1
2	.d2
3	.nta



4.4.1.1 Integer Load

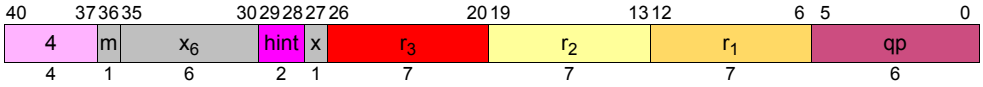
M1



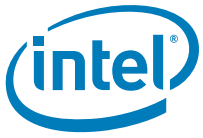
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
ld1.*ldhintx*	r_1 = [r_3]	4	0	0	00	Table 4-39 on page 216
ld2.*ldhintx*					01	
ld4.*ldhintx*					02	
ld8.*ldhintx*					03	
ld1.s.*ldhintx*					04	
ld2.s.*ldhintx*					05	
ld4.s.*ldhintx*					06	
ld8.s.*ldhintx*					07	
ld1.a.*ldhintx*					08	
ld2.a.*ldhintx*					09	
ld4.a.*ldhintx*					0A	
ld8.a.*ldhintx*					0B	
ld1.sa.*ldhintx*					0C	
ld2.sa.*ldhintx*					0D	
ld4.sa.*ldhintx*					0E	
ld8.sa.*ldhintx*					0F	
ld1.bias.*ldhintx*					10	
ld2.bias.*ldhintx*					11	
ld4.bias.*ldhintx*					12	
ld8.bias.*ldhintx*					13	
ld1.acq.*ldhintx*					14	
ld2.acq.*ldhintx*					15	
ld4.acq.*ldhintx*					16	
ld8.acq.*ldhintx*					17	
ld8.fill.*ldhintx*					1B	
ld1.c.clr.*ldhintx*					20	
ld2.c.clr.*ldhintx*					21	
ld4.c.clr.*ldhintx*					22	
ld8.c.clr.*ldhintx*					23	
ld1.c.nc.*ldhintx*					24	
ld2.c.nc.*ldhintx*					25	
ld4.c.nc.*ldhintx*					26	
ld8.c.nc.*ldhintx*					27	
ld1.c.clr.acq.*ldhintx*					28	
ld2.c.clr.acq.*ldhintx*					29	
ld4.c.clr.acq.*ldhintx*					2A	
ld8.c.clr.acq.*ldhintx*					2B	
ld16.*ldhintx*	r_1, ar.csd = [r_3]		0	1	28	
ld16.acq.*ldhintx*					2C	

4.4.1.2 Integer Load – Increment by Register

M2

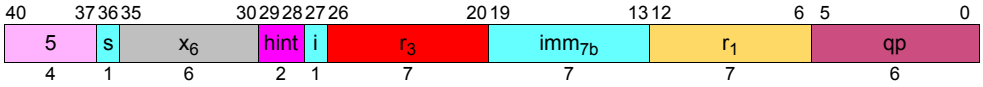


Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ld1.*ldhint*	$r_1 = [r_3], r_2$	4	1	0	00	Table 4-39 on page 216
ld2.*ldhint*					01	
ld4.*ldhint*					02	
ld8.*ldhint*					03	
ld1.s.*ldhint*					04	
ld2.s.*ldhint*					05	
ld4.s.*ldhint*					06	
ld8.s.*ldhint*					07	
ld1.a.*ldhint*					08	
ld2.a.*ldhint*					09	
ld4.a.*ldhint*					0A	
ld8.a.*ldhint*					0B	
ld1.sa.*ldhint*					0C	
ld2.sa.*ldhint*					0D	
ld4.sa.*ldhint*					0E	
ld8.sa.*ldhint*					0F	
ld1.bias.*ldhint*					10	
ld2.bias.*ldhint*					11	
ld4.bias.*ldhint*					12	
ld8.bias.*ldhint*					13	
ld1.acq.*ldhint*					14	
ld2.acq.*ldhint*					15	
ld4.acq.*ldhint*					16	
ld8.acq.*ldhint*					17	
ld8.fill.*ldhint*					1B	
ld1.c.clr.*ldhint*					20	
ld2.c.clr.*ldhint*					21	
ld4.c.clr.*ldhint*					22	
ld8.c.clr.*ldhint*					23	
ld1.c.nc.*ldhint*					24	
ld2.c.nc.*ldhint*					25	
ld4.c.nc.*ldhint*					26	
ld8.c.nc.*ldhint*					27	
ld1.c.clr.acq.*ldhint*					28	
ld2.c.clr.acq.*ldhint*					29	
ld4.c.clr.acq.*ldhint*					2A	
ld8.c.clr.acq.*ldhint*					2B	



4.4.1.3 Integer Load – Increment by Immediate

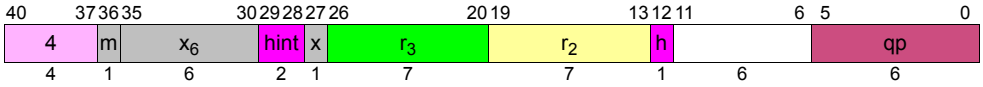
M3



Instruction	Operands	Opcode	Extension	
			x_6	hint
ld1.*ldhint*	$r_1 = [r_3]$, imm_9	5	00	Table 4-39 on page 216
ld2.*ldhint*			01	
ld4.*ldhint*			02	
ld8.*ldhint*			03	
ld1.s.*ldhint*			04	
ld2.s.*ldhint*			05	
ld4.s.*ldhint*			06	
ld8.s.*ldhint*			07	
ld1.a.*ldhint*			08	
ld2.a.*ldhint*			09	
ld4.a.*ldhint*			0A	
ld8.a.*ldhint*			0B	
ld1.sa.*ldhint*			0C	
ld2.sa.*ldhint*			0D	
ld4.sa.*ldhint*			0E	
ld8.sa.*ldhint*			0F	
ld1.bias.*ldhint*			10	
ld2.bias.*ldhint*			11	
ld4.bias.*ldhint*			12	
ld8.bias.*ldhint*			13	
ld1.acq.*ldhint*			14	
ld2.acq.*ldhint*			15	
ld4.acq.*ldhint*			16	
ld8.acq.*ldhint*			17	
ld8.fill.*ldhint*			1B	
ld1.c.clr.*ldhint*			20	
ld2.c.clr.*ldhint*			21	
ld4.c.clr.*ldhint*			22	
ld8.c.clr.*ldhint*			23	
ld1.c.nc.*ldhint*			24	
ld2.c.nc.*ldhint*			25	
ld4.c.nc.*ldhint*			26	
ld8.c.nc.*ldhint*			27	
ld1.c.clr.acq.*ldhint*			28	
ld2.c.clr.acq.*ldhint*			29	
ld4.c.clr.acq.*ldhint*			2A	
ld8.c.clr.acq.*ldhint*			2B	

4.4.1.4 Integer Store

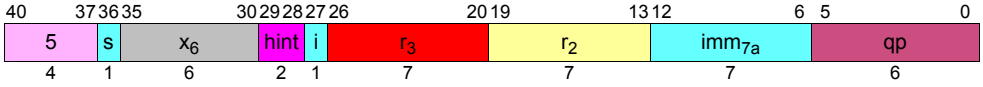
M4



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
st1.*sthintx*	$[r_3] = r_2$	4	0	0	30	Table 4-42 on page 217
st2.*sthintx*					31	
st4.*sthintx*					32	
st8.*sthintx*					33	
st1.rel.*sthintx*					34	
st2.rel.*sthintx*					35	
st4.rel.*sthintx*					36	
st8.rel.*sthintx*					37	
st8.spill.*sthintx*					3B	
st16.*sthintx*	$[r_3] = r_2$, ar.csd		0	1	30	
st16.rel.*sthintx*					34	

4.4.1.5 Integer Store – Increment by Immediate

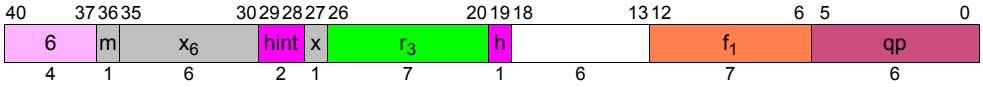
M5



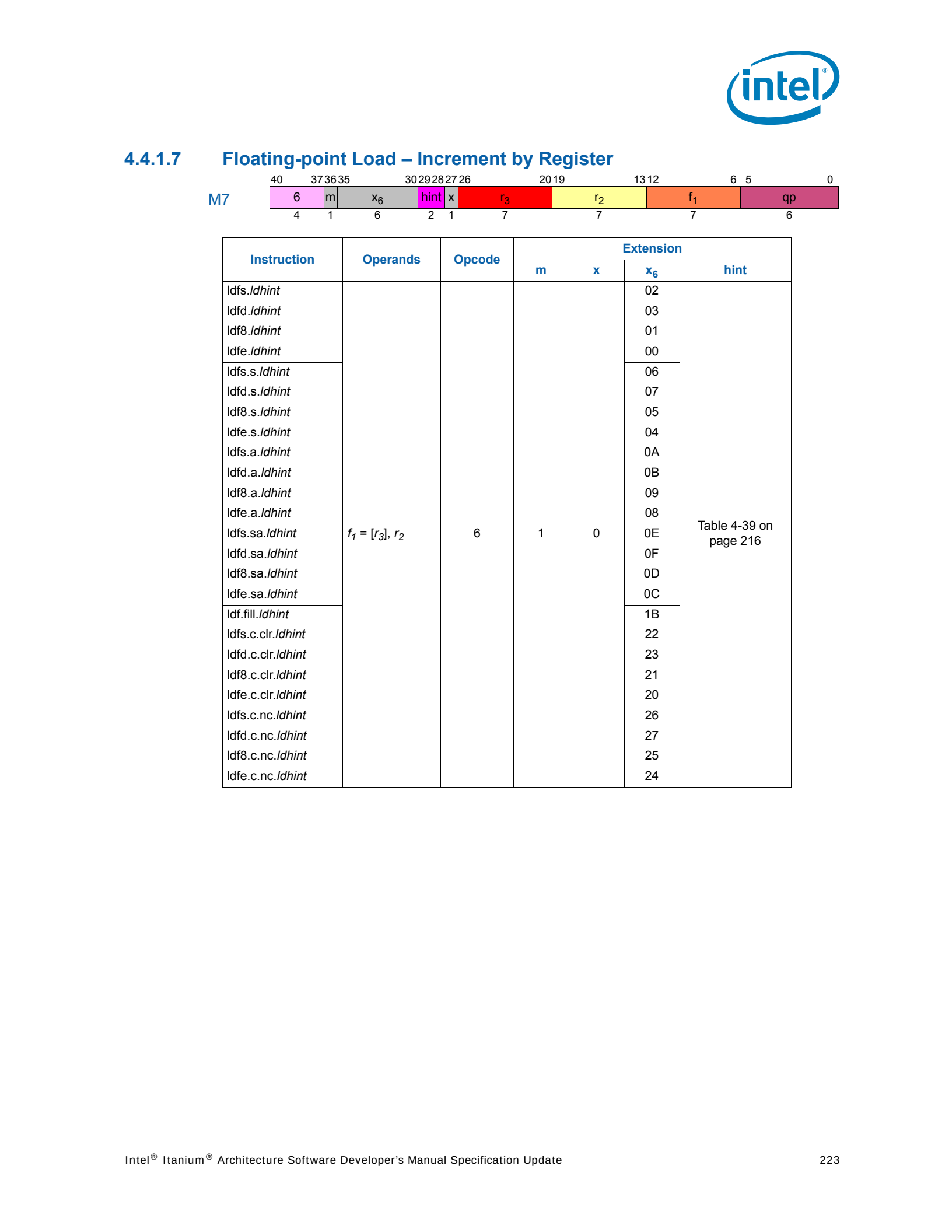
Instruction	Operands	Opcode	Extension	
			x_6	hint
st1.*sthint*	$[r_3] = r_2, imm_9$	5	30	Table 4-42 on page 217
st2.*sthint*			31	
st4.*sthint*			32	
st8.*sthint*			33	
st1.rel.*sthint*			34	
st2.rel.*sthint*			35	
st4.rel.*sthint*			36	
st8.rel.*sthint*			37	
st8.spill.*sthint*			3B	

4.4.1.6 Floating-point Load

M6



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
ldfs.*ldhintx*	$f_1 = [r_3]$	6	0	0	02	Table 4-39 on page 216
ldfd.*ldhintx*					03	
ldf8.*ldhintx*					01	
ldfe.*ldhintx*					00	
ldfs.s.*ldhintx*					06	
ldfd.s.*ldhintx*					07	
ldf8.s.*ldhintx*					05	
ldfe.s.*ldhintx*					04	
ldfs.a.*ldhintx*					0A	
ldfd.a.*ldhintx*					0B	
ldf8.a.*ldhintx*					09	
ldfe.a.*ldhintx*					08	
ldfs.sa.*ldhintx*					0E	
ldfd.sa.*ldhintx*					0F	
ldf8.sa.*ldhintx*					0D	
ldfe.sa.*ldhintx*					0C	
ldf.fill.*ldhintx*					1B	
ldfs.c.clr.*ldhintx*					22	
ldfd.c.clr.*ldhintx*					23	
ldf8.c.clr.*ldhintx*					21	
ldfe.c.clr.*ldhintx*					20	
ldfs.c.nc.*ldhintx*					26	
ldfd.c.nc.*ldhintx*					27	
ldf8.c.nc.*ldhintx*					25	
ldfe.c.nc.*ldhintx*					24	



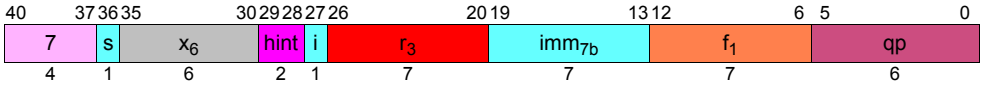
4.4.1.7 Floating-point Load – Increment by Register

M7

Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfs.*ldhint*	$f_1 = [r_3], r_2$	6	1	0	02	Table 4-39 on page 216
ldfd.*ldhint*					03	
ldf8.*ldhint*					01	
ldfe.*ldhint*					00	
ldfs.s.*ldhint*					06	
ldfd.s.*ldhint*					07	
ldf8.s.*ldhint*					05	
ldfe.s.*ldhint*					04	
ldfs.a.*ldhint*					0A	
ldfd.a.*ldhint*					0B	
ldf8.a.*ldhint*					09	
ldfe.a.*ldhint*					08	
ldfs.sa.*ldhint*					0E	
ldfd.sa.*ldhint*					0F	
ldf8.sa.*ldhint*					0D	
ldfe.sa.*ldhint*					0C	
ldf.fill.*ldhint*					1B	
ldfs.c.clr.*ldhint*					22	
ldfd.c.clr.*ldhint*					23	
ldf8.c.clr.*ldhint*					21	
ldfe.c.clr.*ldhint*					20	
ldfs.c.nc.*ldhint*					26	
ldfd.c.nc.*ldhint*					27	
ldf8.c.nc.*ldhint*					25	
ldfe.c.nc.*ldhint*					24	

4.4.1.8 Floating-point Load – Increment by Immediate

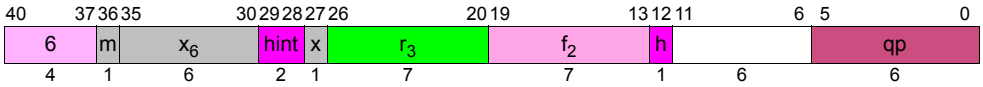
M8



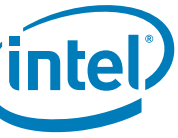
Instruction	Operands	Opcode	Extension	
			x_6	hint
ldfs.*ldhint*	$f_1 = [r_3]$, imm_9	7	02	Table 4-39 on page 216
ldfd.*ldhint*			03	
ldf8.*ldhint*			01	
ldfe.*ldhint*			00	
ldfs.s.*ldhint*			06	
ldfd.s.*ldhint*			07	
ldf8.s.*ldhint*			05	
ldfe.s.*ldhint*			04	
ldfs.a.*ldhint*			0A	
ldfd.a.*ldhint*			0B	
ldf8.a.*ldhint*			09	
ldfe.a.*ldhint*			08	
ldfs.sa.*ldhint*			0E	
ldfd.sa.*ldhint*			0F	
ldf8.sa.*ldhint*			0D	
ldfe.sa.*ldhint*			0C	
ldf.fill.*ldhint*			1B	
ldfs.c.clr.*ldhint*			22	
ldfd.c.clr.*ldhint*			23	
ldf8.c.clr.*ldhint*			21	
ldfe.c.clr.*ldhint*			20	
ldfs.c.nc.*ldhint*			26	
ldfd.c.nc.*ldhint*			27	
ldf8.c.nc.*ldhint*			25	
ldfe.c.nc.*ldhint*			24	

4.4.1.9 Floating-point Store

M9

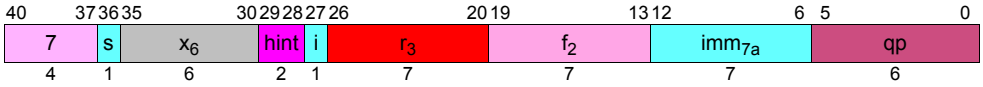


Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
stfs.*sthintx*	$[r_3] = f_2$	6	0	0	32	Table 4-42 on page 217
stfd.*sthintx*					33	
stf8.*sthintx*					31	
stfe.*sthintx*					30	
stf.spill.*sthintx*					3B	



4.4.1.10 Floating-point Store – Increment by Immediate

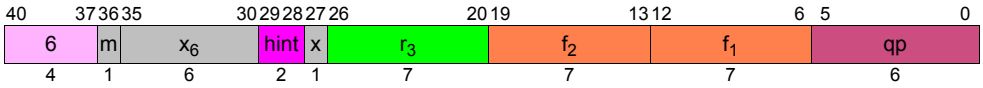
M10



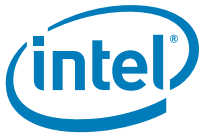
Instruction	Operands	Opcode	Extension	
			x_6	hint
stfs.*sthint*	$[r_3] = f_2, imm_9$	7	32	Table 4-42 on page 217
stfd.*sthint*			33	
stf8.*sthint*			31	
stfe.*sthint*			30	
stf.spill.*sthint*			3B	

4.4.1.11 Floating-point Load Pair

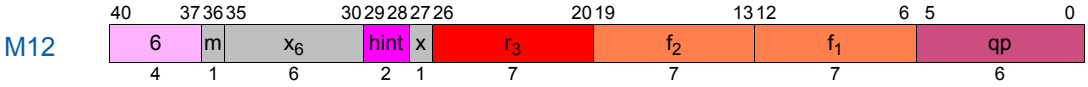
M11



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	$f_1, f_2 = [r_3]$	6	0	1	02	Table 4-39 on page 216
ldfpd.*ldhint*					03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*					06	
ldfpd.s.*ldhint*					07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*					0A	
ldfpd.a.*ldhint*					0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*					0E	
ldfpd.sa.*ldhint*					0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*					22	
ldfpd.c.clr.*ldhint*					23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*					26	
ldfpd.c.nc.*ldhint*					27	
ldfp8.c.nc.*ldhint*					25	



4.4.1.12 Floating-point Load Pair – Increment by Immediate



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
ldfps.*ldhint*	$f_1, f_2 = [r_3]$, 8	6	1	1	02	See Table 4-39 on page 3:169
ldfpd.*ldhint*	$f_1, f_2 = [r_3]$, 16				03	
ldfp8.*ldhint*					01	
ldfps.s.*ldhint*	$f_1, f_2 = [r_3]$, 8				06	
ldfpd.s.*ldhint*	$f_1, f_2 = [r_3]$, 16				07	
ldfp8.s.*ldhint*					05	
ldfps.a.*ldhint*	$f_1, f_2 = [r_3]$, 8				0A	
ldfpd.a.*ldhint*	$f_1, f_2 = [r_3]$, 16				0B	
ldfp8.a.*ldhint*					09	
ldfps.sa.*ldhint*	$f_1, f_2 = [r_3]$, 8				0E	
ldfpd.sa.*ldhint*	$f_1, f_2 = [r_3]$, 16				0F	
ldfp8.sa.*ldhint*					0D	
ldfps.c.clr.*ldhint*	$f_1, f_2 = [r_3]$, 8				22	
ldfpd.c.clr.*ldhint*	$f_1, f_2 = [r_3]$, 16				23	
ldfp8.c.clr.*ldhint*					21	
ldfps.c.nc.*ldhint*	$f_1, f_2 = [r_3]$, 8				26	
ldfpd.c.nc.*ldhint*	$f_1, f_2 = [r_3]$, 16				27	
ldfp8.c.nc.*ldhint*					25	

4.4.2 Line Prefetch

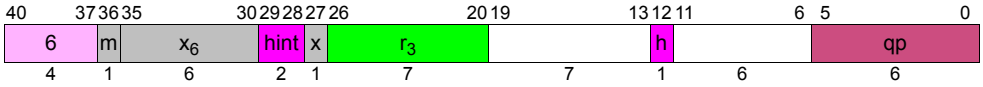
The line prefetch instructions are encoded in major opcodes 6 and 7 along with the floating-point load/store instructions. See “Loads and Stores” on page 211 for a summary of the opcode extensions. The line prefetch instructions all have a 2-bit cache locality opcode hint extension field in bits 29:28 (hint) as shown in Table 4-48.

Table 4-45.Line Prefetch Hint Completer

h Bit 12	hint Bits 29:28	*lfhint*
0	0	*none*
	1	.nt1
	2	.nt2
	3	.nta
1	0	.d4
	1	.d5
	2	.d6
	3	.d7

4.4.2.1 Line Prefetch

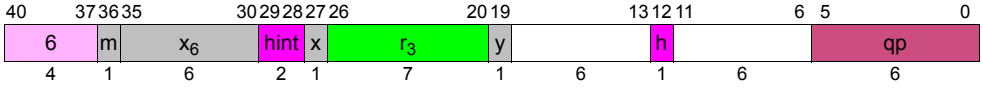
M13



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3]	6	0	0	2D 2E 2F	Table 4-45 on page 226

4.4.2.2 Line Prefetch

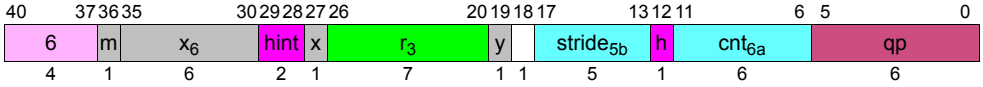
M51



Instruction	Operands	Opcode	Extension				
			m	x	y	x_6	hint, h
lfetch.*lfhint*	[r_3]	6	0	0	0	2C	Table 4-45 on page 226

4.4.2.3 Counted Line Prefetch

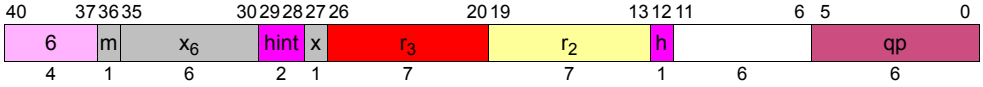
M52



Instruction	Operands	Opcode	Extension				
			m	x	y	x_6	hint, h
lfetch.count.*lfhint*	[r_3], cnt_6, $stride_5$	6	0	0	1	2C	Table 4-45 on page 226

4.4.2.4 Line Prefetch – Increment by Register

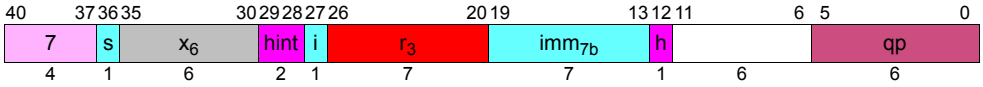
M14



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint, h
lfetch.*lfhint* lfetch.excl.*lfhint* lfetch.fault.*lfhint* lfetch.fault.excl.*lfhint*	[r_3], r_2	6	1	0	2C 2D 2E 2F	Table 4-45 on page 226

4.4.2.5 Line Prefetch – Increment by Immediate

M15



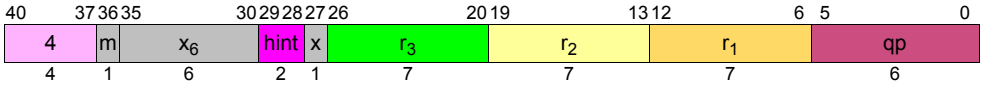
Instruction	Operands	Opcode	Extension	
			x_6	hint, h
lfetch.*lfhint*	[r_3], *imm*$_9$	7	2C	Table 4-45 on page 226
lfetch.excl.*lfhint*			2D	
lfetch.fault.*lfhint*			2E	
lfetch.fault.excl.*lfhint*			2F	

4.4.3 Semaphores

The semaphore instructions are encoded in major opcode 4 along with the integer load/store instructions. See “Loads and Stores” on page 211 for a summary of the opcode extensions. These instructions have the same cache locality opcode hint extension field in bits 29:28 (hint) as load instructions. See Table 4-39 on page 216.

4.4.3.1 Exchange/Compare and Exchange

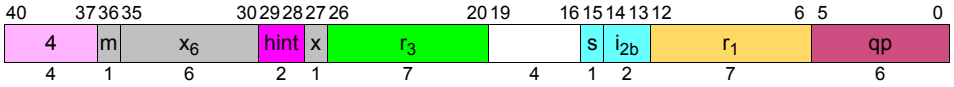
M16



Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
cmpxchg1.acq.*ldhint*	r_1 = [r_3], r_2, ar.ccv	4	0	1	00	Table 4-39 on page 216
cmpxchg2.acq.*ldhint*					01	
cmpxchg4.acq.*ldhint*					02	
cmpxchg8.acq.*ldhint*					03	
cmpxchg1.rel.*ldhint*					04	
cmpxchg2.rel.*ldhint*					05	
cmpxchg4.rel.*ldhint*					06	
cmpxchg8.rel.*ldhint*					07	
cmp8xchg16.acq.*ldhint*	r_1 = [r_3], r_2, ar.csd, ar.ccv				20	
cmp8xchg16.rel.*ldhint*					24	
xchg1.*ldhint*	r_1 = [r_3], r_2				08	
xchg2.*ldhint*					09	
xchg4.*ldhint*					0A	
xchg8.*ldhint*					0B	

4.4.3.2 Fetch and Add – Immediate

M17



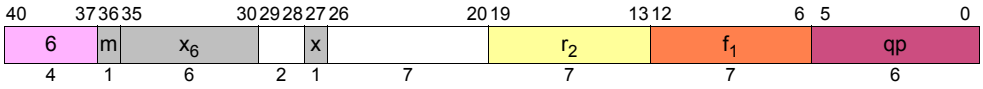
Instruction	Operands	Opcode	Extension			
			m	x	x_6	hint
fetchadd4.acq.*ldhint*	$r_1 = [r_3]$, inc_3	4	0	1	12	Table 4-39 on page 216
fetchadd8.acq.*ldhint*					13	
fetchadd4.rel.*ldhint*					16	
fetchadd8.rel.*ldhint*					17	

13.3.3 Set/Get FR

The set FR instructions are encoded in major opcode 6 along with the floating-point load/store instructions. The get FR instructions are encoded in major opcode 4 along with the integer load/store instructions. See “Loads and Stores” on page 211 for a summary of the opcode extensions.

4.4.3.3 Set FR

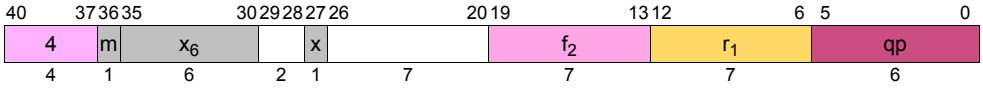
M18



Instruction	Operands	Opcode	Extension		
			m	x	x_6
setf.sig	$f_1 = r_2$	6	0	1	1C
setf.exp					1D
setf.s					1E
setf.d					1F

4.4.3.4 Get FR

M19



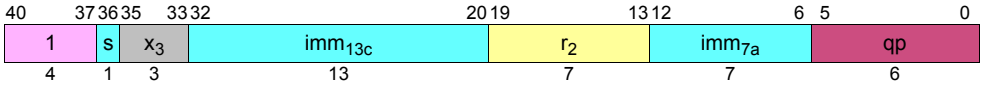
Instruction	Operands	Opcode	Extension		
			m	x	x_6
getf.sig	$r_1 = f_2$	4	0	1	1C
getf.exp					1D
getf.s					1E
getf.d					1F

4.4.4 Speculation and Advanced Load Checks

The speculation and advanced load check instructions are encoded in major opcodes 0 and 1 along with the system/memory management instructions. See “System/Memory Management” on page 234 for a summary of the opcode extensions.

4.4.4.1 Integer Speculation Check (M-Unit)

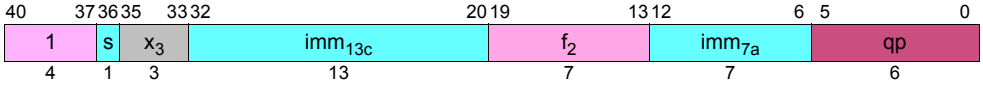
M20



Instruction	Operands	Opcode	Extension
			x_3
chk.s.m	r_2, $target_{25}$	1	1

4.4.4.2 Floating-point Speculation Check

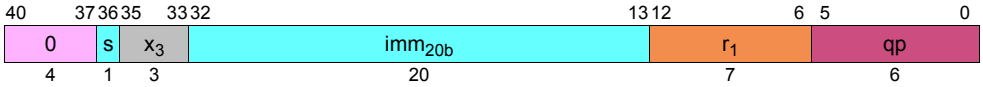
M21



Instruction	Operands	Opcode	Extension
			x_3
chk.s	f_2, $target_{25}$	1	3

4.4.4.3 Integer Advanced Load Check

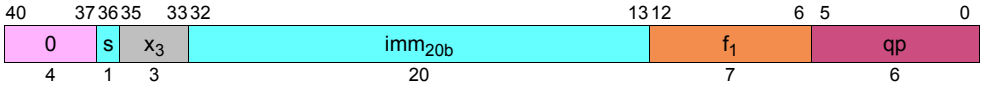
M22



Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	r_1, $target_{25}$	0	4
chk.a.clr			5

4.4.4.4 Floating-point Advanced Load Check

M23



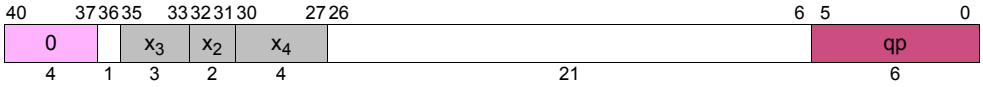
Instruction	Operands	Opcode	Extension
			x_3
chk.a.nc	f_1, $target_{25}$	0	6
chk.a.clr			7

4.4.5 Cache/Synchronization/RSE/ALAT

The cache/synchronization/RSE/ALAT instructions are encoded in major opcode 0 along with the memory management instructions. See “System/Memory Management” on page 234 for a summary of the opcode extensions.

4.4.5.1 Sync/Fence/Serialize/ALAT Control

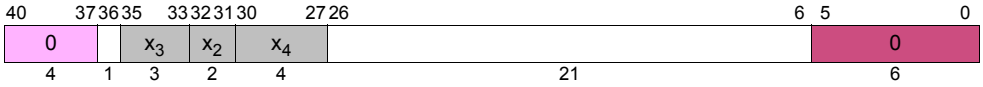
M24



Instruction	Opcode	Extension		
		x_3	x_4	x_2
invala	0	0	0	1
fwb			0	2
mf			2	
mf.a			3	
srlz.d			0	3
srlz.i			1	
sync.i			3	

4.4.5.2 RSE Control

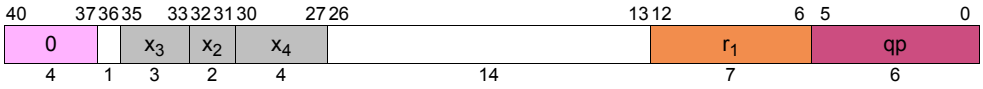
M25



Instruction	Opcode	Extension		
		x_3	x_4	x_2
flushrs [f]	0	0	C	0
loadrs [f]			A	

4.4.5.3 Integer ALAT Entry Invalidate

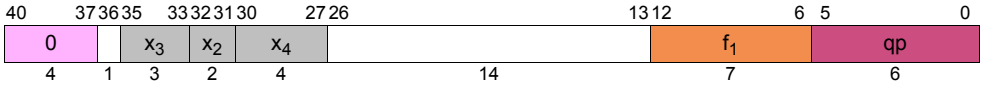
M26



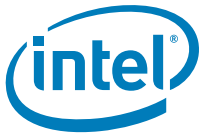
Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	r_1	0	0	2	1

4.4.5.4 Floating-point ALAT Entry Invalidate

M27

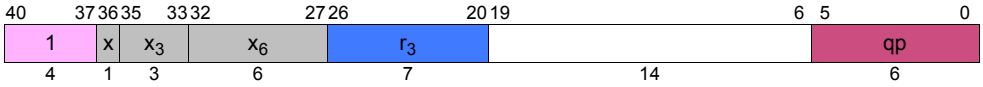


Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
invala.e	f_1	0	0	3	1



4.4.5.5 Flush Cache

M28



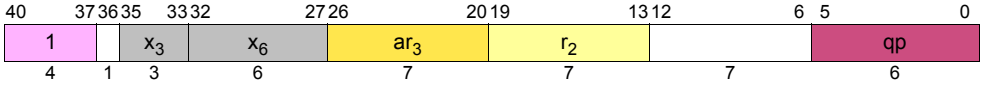
Instruction	Operands	Opcode	Extension		
			x_3	x_6	x
fc fc.i	r_3	1	0	30	0 1

13.3.3 GR/AR Moves (M-Unit)

The M-Unit GR/AR move instructions are encoded in major opcode 0 along with the system/memory management instructions. (Some ARs are accessed using system control instructions on the I-unit. See “GR/AR Moves (I-Unit)” on page 209.) See “System/Memory Management” on page 234 for a summary of the M-Unit GR/AR opcode extensions.

4.4.5.6 Move to AR – Register (M-Unit)

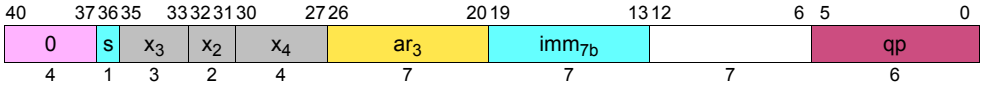
M29



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$ar_3 = r_2$	1	0	2A

4.4.5.7 Move to AR – Immediate$_8$ (M-Unit)

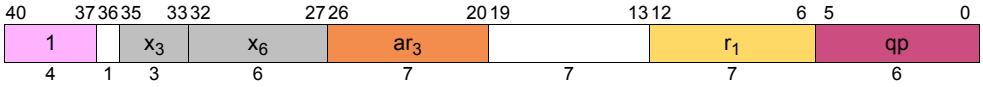
M30



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
mov.m	$ar_3 = imm_8$	0	0	8	2

4.4.5.8 Move from AR (M-Unit)

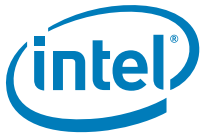
M31



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov.m	$r_1 = ar_3$	1	0	22

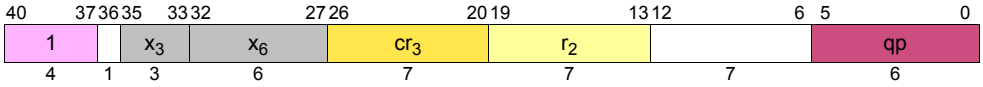
4.4.6 GR/CR Moves

The GR/CR move instructions are encoded in major opcode 0 along with the system/memory management instructions. See “System/Memory Management” on page 234 for a summary of the opcode extensions.



4.4.6.1 Move to CR

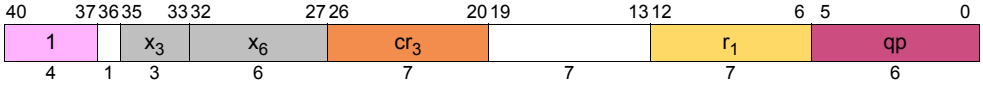
M32



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	$cr_3 = r_2$	1	0	2C

4.4.6.2 Move from CR

M33



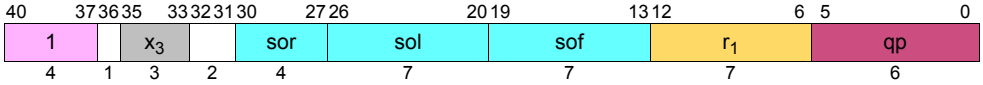
Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	$r_1 = cr_3$	1	0	24

4.4.7 Miscellaneous M-Unit Instructions

The miscellaneous M-unit instructions are encoded in major opcode 0 along with the system/memory management instructions. See “System/Memory Management” on page 234 for a summary of the opcode extensions.

4.4.7.1 Allocate Register Stack Frame

M34

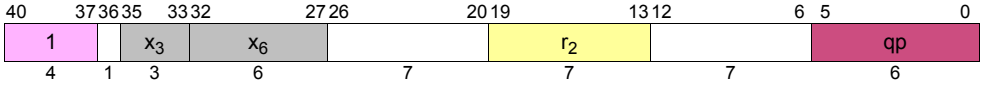


Instruction	Operands	Opcode	Extension
			x_3
alloc [f]	r_1 = ar.pfs, *i*, *l*, *o*, *r*	1	6

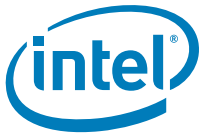
Note: The three immediates in the instruction encoding are formed from the operands as follows:
sof = *i* + *l* + *o*
sol = *i* + *l*
sor = *r* >> 3

4.4.7.2 Move to PSR

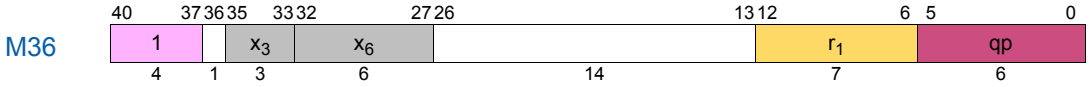
M35



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	psr.l = r_2	1	0	2D
mov	psr.um = r_2			29

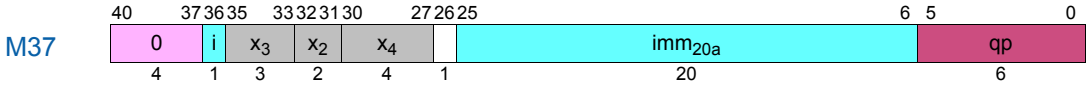


4.4.7.3 Move from PSR



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = psr	1	0	25
mov	r_1 = psr.um			21

4.4.7.4 Break (M-Unit)



Instruction	Operands	Opcode	Extension		
			x_3	x_4	x_2
break.m	imm_{21}	0	0	0	0

13.3.3 System/Memory Management

All system/memory management instructions are encoded within major opcodes 0 and 1 using a 3-bit opcode extension field (x_3) in bits 35:33. Some instructions also have a 4-bit opcode extension field (x_4) in bits 30:27, or a 6-bit opcode extension field (x_6) in bits 32:27. Most of the instructions having a 4-bit opcode extension field also have a 2-bit extension field (x_2) in bits 32:31. Table 4-46 shows the 3-bit assignments for opcode 0, Table 4-47 summarizes the 4-bit+2-bit assignments for opcode 0, Table 4-48 shows the 3-bit assignments for opcode 1, and Table 4-49 summarizes the 6-bit assignments for opcode 1.

Table 4-46.Opcode 0 System/Memory Management 3-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	
0	0	System/Memory Management 4-bit+2-bit Ext (Table 4-47)
	1	
	2	
	3	
	4	chk.a.nc – int M22
	5	chk.a.clr – int M22
	6	chk.a.nc – fp M23
	7	chk.a.clr – fp M23

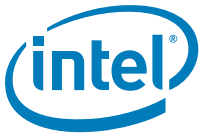


Table 4-47.Opcode 0 System/Memory Management 4-bit+2-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31			
			0	1	2	3
0	0	0	break.m M37	invala M24	fwb M24	srlz.d M24
		1	1-bit Ext (Table 4-51)			srlz.i M24
		2		invala.e – int M26	mf M24	
		3		invala.e – fp M27	mf.a M24	sync.i M24
		4	sum M44			
		5	rum M44			
		6	ssm M44			
		7	rsm M44			
		8			mov.m to ar – imm_8 M30	
		9				
		A	loadrs M25			
		B				
		C	flushrs M25			
		D				
		E				
		F				

Table 4-48.Opcode 1 System/Memory Management 3-bit Opcode Extensions

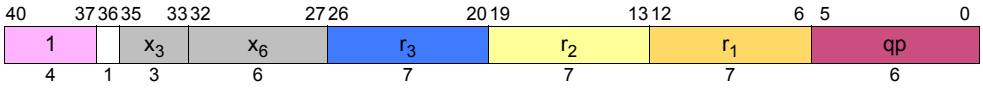
Opcode Bits 40:37	x_3 Bits 35:33	
1	0	System/Memory Management 6-bit Ext (Table 4-49)
	1	chk.s.m – int M20
	2	
	3	chk.s – fp M21
	4	
	5	
	6	alloc M34
	7	

Table 4-49.Opcode 1 System/Memory Management 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6				
		Bits 30:27	Bits 32:31			
			0	1	2	3
1	0	0	mov to rr M42	mov from rr M43	mov from dahr M43	fc M28
		1	mov to dbr M42	mov from dbr M43	mov from psr.um M36	probe.rw.fault – imm_2 M40
		2	mov to ibr M42	mov from ibr M43	mov.m from ar M31	probe.r.fault – imm_2 M40
		3	mov to pkr M42	mov from pkr M43		probe.w.fault – imm_2 M40
		4	mov to pmc M42	mov from pmc M43	mov from cr M33	ptc.e M47
		5	mov to pmd M42	mov from pmd M43	mov from psr M36	
		6				
		7		mov from cpuid M43		
		8		probe.r – imm_2 M39		probe.r M38
		9	ptc.l M45	probe.w – imm_2 M39	mov to psr.um M35	probe.w M38
		A	ptc.g M45	thash M46	mov.m to ar M29	
		B	ptc.ga M45	ttag M46		
		C	ptr.d M45		mov to cr M32	
		D	ptr.i M45		mov to psr.l M35	
		E	itr.d M42	tpa M46	itc.d M41	
		F	itr.i M42	tak M46	itc.i M41	

4.4.7.5 Probe – Register

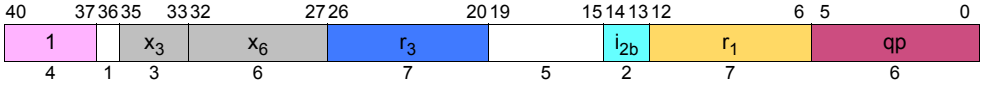
M38



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r	$r_1 = r_3, r_2$	1	0	38
probe.w				39

4.4.7.6 Probe – Immediate$_2$

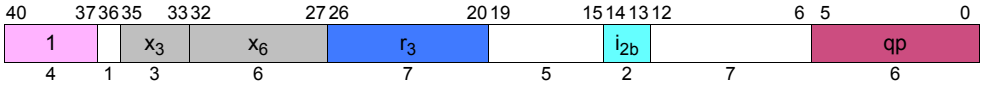
M39



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.r	$r_1 = r_3, imm_2$	1	0	18
probe.w				19

4.4.7.7 Probe Fault – Immediate$_2$

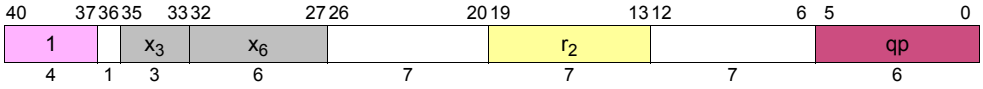
M40



Instruction	Operands	Opcode	Extension	
			x_3	x_6
probe.rw.fault	r_3, imm_2	1	0	31
probe.r.fault				32
probe.w.fault				33

4.4.7.8 Translation Cache Insert

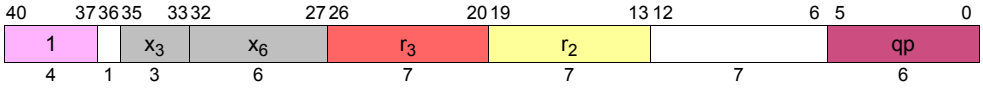
M41



Instruction	Operands	Opcode	Extension	
			x_3	x_6
itc.d [l p]	r_2	1	0	2E
itc.i [l p]				2F

4.4.7.9 Move to Indirect Register/Translation Register Insert

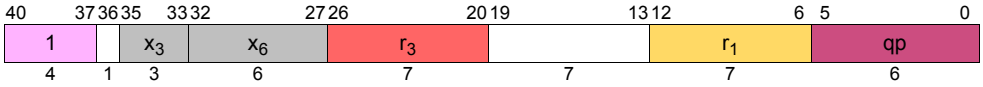
M42



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	rr[r_3] = r_2	1	0	00
	dbr[r_3] = r_2			01
	ibr[r_3] = r_2			02
	pkr[r_3] = r_2			03
	pmc[r_3] = r_2			04
	pmd[r_3] = r_2			05
itr.d [p]	dtr[r_3] = r_2			0E
itr.i [p]	itr[r_3] = r_2			0F

4.4.7.10 Move from Indirect Register

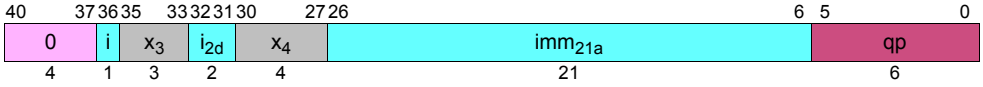
M43



Instruction	Operands	Opcode	Extension	
			x_3	x_6
mov [p]	r_1 = rr[r_3]	1	0	10
	r_1 = dbr[r_3]			11
	r_1 = ibr[r_3]			12
	r_1 = pkr[r_3]			13
	r_1 = pmc[r_3]			14
mov	r_1 = pmd[r_3]			15
	r_1 = cpuid[r_3]			17
	r_1 = dahr[r_3]			20

4.4.7.11 Set/Reset User/System Mask

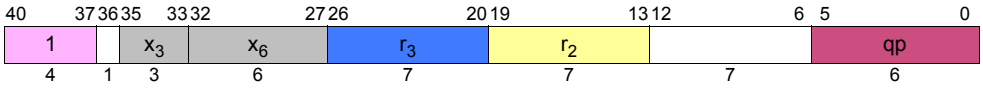
M44



Instruction	Operands	Opcode	Extension	
			x_3	x_4
sum	imm_{24}	0	0	4
rum				5
ssm [p]				6
rsm [p]				7

4.4.7.12 Translation Purge

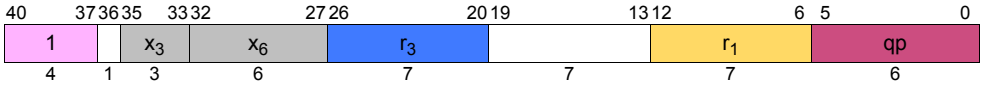
M45



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.l [p]	r_3, r_2	1	0	09
ptc.g [l p]				0A
ptc.ga [l p]				0B
ptr.d [p]				0C
ptr.i [p]				0D

4.4.7.13 Translation Access

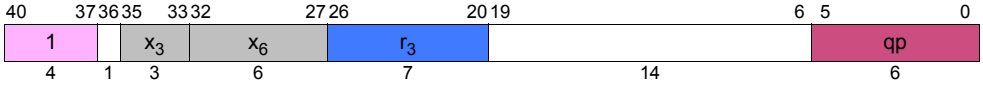
M46



Instruction	Operands	Opcode	Extension	
			x_3	x_6
thash ttag tpa [p] tak [p]	$r_1 = r_3$	1	0	1A 1B 1E 1F

4.4.7.14 Purge Translation Cache Entry

M47



Instruction	Operands	Opcode	Extension	
			x_3	x_6
ptc.e [p]	r_3	1	0	34

4.4.8 Nop/Hint/Move to DAHR (M-Unit)

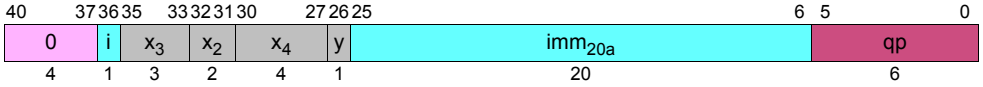
M-unit nop, hint and mov to DAHR instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 2-bit opcode extension field in bits 32:31 (x_2), a 4-bit opcode extension field in bits 30:27 (x_4), a 1-bit opcode extension field in bit 26 (y), and a 2-bit opcode extension field in bits 11:10 (z), as shown in Table 4-50.

Table 4-50.Misc M-Unit 1-bit Opcode Extensions

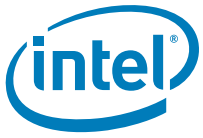
Opcode Bits 40:37	x_3 Bits 35:33	x_4 Bits 30:27	x_2 Bits 32:31	y Bit 26	z Bits 11:10			
					0	1	2	3
0	0	1	0	0	nop.m M48			
				1	hint.m M49	mov to dahr M50		

4.4.8.1 Nop

M48

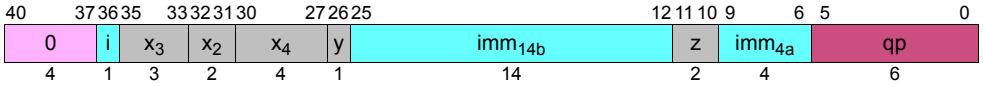


Instruction	Operands	Opcode	Extension			
			x_3	x_4	x_2	y
nop.m	imm_{21}	0	0	1	0	0



4.4.8.2 Hint

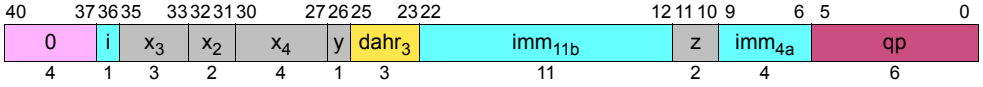
M49



Instruction	Operands	Opcode	Extension				
			x_3	x_4	x_2	y	z
hint.m	imm_{19}	0	0	1	0	1	0

4.4.8.3 Move to DAHR

M50



Instruction	Operands	Opcode	Extension				
			x_3	x_4	x_2	y	z
mov	$dahr_3 = imm_{16}$	0	0	1	0	1	1

4.5 B-Unit Instruction Encodings

The branch-unit includes branch, predict, and miscellaneous instructions.

4.5.1 Branches

Opcode 0 is used for indirect branch, opcode 1 for indirect call, opcode 4 for IP-relative branch, and opcode 5 for IP-relative call.
The IP-relative branch instructions encoded within major opcode 4 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-51.

Table 4-51.IP-Relative Branch Types

Opcode Bits 40:37	btype Bits 8:6	
4	0	br.cond B1
	1	e
	2	br.wexit B1
	3	br.wtop B1
	4	e
	5	br.cloop B2
	6	br.cexit B2
	7	br.ctop B2

The indirect branch, indirect return, and miscellaneous branch-unit instructions are encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-52 summarizes these assignments.

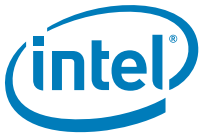


Table 4-52.Indirect/Miscellaneous Branch Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
0	0	break.b B9	epc B8	Indirect Branch (Table 4-53)	e
	1		e	Indirect Return (Table 4-54)	e
	2	cover B8	e	e	e
	3	e	e	e	e
	4	clrrrb B8	e	e	e
	5	clrrrb.pr B8	e	e	e
	6	e	e	e	e
	7	e	e	e	e
	8	rfi B8	vmsw.0 B8	e	e
	9		vmsw.1 B8	e	e
	A	e	e	e	e
	B	e	e	e	e
	C	bsw.0 B8	e	e	e
	D	bsw.1 B8	e	e	e
	E	e	e	e	e
	F	e	e	e	e

The indirect branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-53.

Table 4-53.Indirect Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	20	0	br.cond B4
		1	br.ia B4
		2	e
		3	e
		4	e
		5	e
		6	e
		7	e

The indirect return branch instructions encoded within major opcodes 0 use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-54.

Table 4-54.Indirect Return Branch Types

Opcode Bits 40:37	x_6 Bits 32:27	btype Bits 8:6	
0	21	0	e
		1	e
		2	e
		3	e
		4	br.ret B4
		5	e
		6	e
		7	e

All of the branch instructions have a 1-bit sequential prefetch opcode hint extension field, p, in bit 12. Table 4-55 summarizes these assignments.

Table 4-55.Sequential Prefetch Hint Completer

p Bit 12 or Bit 5	*ph*
0	.few
1	.many

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 4-56. Indirect call instructions have a 3-bit “whether” opcode hint extension field in bits 34:32 (wh) as shown in Table 4-57.

Table 4-56.Branch Whether Hint Completer

wh Bits 34:33	*bwh*
0	.sptk
1	.spnt
2	.dptk
3	.dpnt

Table 4-57.Indirect Call Whether Hint Completer

wh Bits 34:32	*bwh*
0	
1	.sptk
2	
3	.spnt
4	
5	.dptk
6	
7	.dpnt

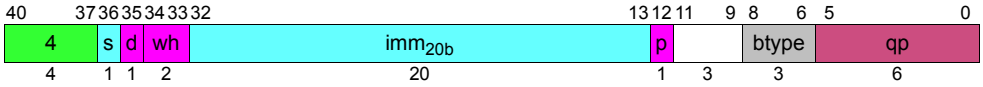
The branch instructions also have a 1-bit branch cache deallocation opcode hint extension field in bit 35 (d) as shown in Table 4-58.

Table 4-58.Branch Cache Deallocation Hint Completer

d Bit 35	*dh*
0	*none*
1	.clr

4.5.1.1 IP-Relative Branch

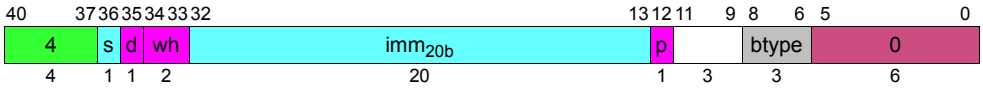
B1



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cond.*bwh.ph.dh* e br.wexit.*bwh.ph.dh* e t br.wtop.*bwh.ph.dh* e t	$target_{25}$	4	0 2 3	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243

4.5.1.2 IP-Relative Counted Branch

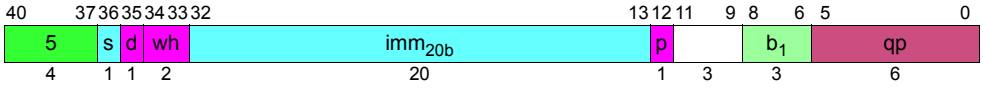
B2



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
br.cloop.*bwh.ph.dh* e t br.cexit.*bwh.ph.dh* e t br.ctop.*bwh.ph.dh* e t	$target_{25}$	4	5 6 7	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243

4.5.1.3 IP-Relative Call

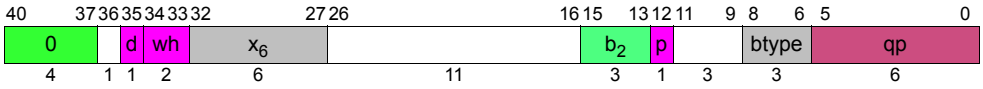
B3



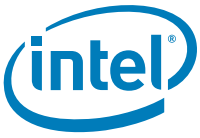
Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* e	$b_1 = target_{25}$	5	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243

4.5.1.4 Indirect Branch

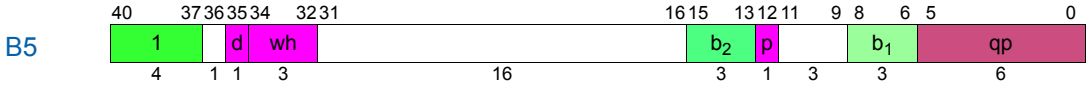
B4



Instruction	Operands	Opcode	Extension				
			x_6	btype	p	wh	d
br.cond.*bwh.ph.dh* e br.ia.*bwh.ph.dh* e	b_2	0	20	0 1	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243
br.ret.*bwh.ph.dh* e			21	4			



4.5.1.5 Indirect Call



Instruction	Operands	Opcode	Extension		
			p	wh	d
br.call.*bwh.ph.dh* [e]	$b_1 = b_2$	1	See Table 4-55 on page 3:242	See Table 4-57 on page 3:242	See Table 4-58 on page 3:243

4.5.2 Branch Predict/Nop/Hint

The branch predict, nop, and hint instructions are encoded in major opcodes 2 (Indirect Predict/Nop/Hint) and 7 (IP-relative Predict). The indirect predict, nop, and hint instructions in major opcode 2 use a 6-bit opcode extension field in bits 32:27 (x_6). Table 4-59 summarizes these assignments.

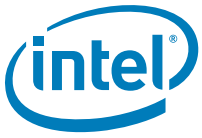
Table 4-59.Indirect Predict/Nop/Hint Opcode Extensions

Opcode Bits 40:37	x_6				
	Bits 30:27	Bits 32:31			
		0	1	2	3
2	0	nop.b B9	brp B7		
	1	hint.b B9	brp.ret B7		
	2				
	3				
	4				
	5				
	6				
	7				
	8				
	9				
	A				
	B				
	C				
	D				
	E				
	F				

The branch predict instructions all have a 1-bit branch importance opcode hint extension field in bit 35 (ih). The mov to BR instruction (page 208) also has this hint in bit 23. Table 4-60 shows these assignments.

Table 4-60.Branch Importance Hint Completer

ih Bit 23 or Bit 35	*ih*
0	*none*
1	.imp



The IP-relative branch predict instructions have a 2-bit branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in Table 4-61. Note that the combination of the .loop or .exit whether hint completer with the *none* importance hint completer is undefined.

Table 4-61.IP-Relative Predict Whether Hint Completer

wh Bits 4:3	*ipwh*
0	.sptk
1	.loop
2	.dptk
3	.exit

The indirect branch predict instructions have a 2-bit branch prediction “whether” opcode hint extension field in bits 4:3 (wh) as shown in Table 4-62.

Table 4-62.Indirect Predict Whether Hint Completer

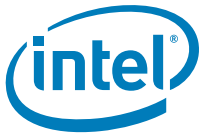
wh Bits 4:3	*indwh*
0	.sptk
1	
2	.dptk
3	

Table 4-63.Sequential Prefetch Hint Completer

p Bit 12	*ph*
0	*none*
1	.many

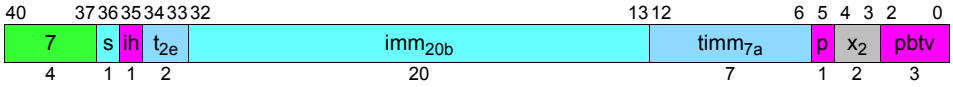
Table 4-64.Prefetch Branch Trace Vector Hint Completer

pbtv Bits 11:9 or Bits 2:0	*pvec*
0	*none*
1	.dc.nt
2	.tk.dc
3	.tk.tk
4	.tk.nt
5	.nt.dc
6	.nt.tk
7	.nt.nt



4.5.2.1 IP-Relative Predict

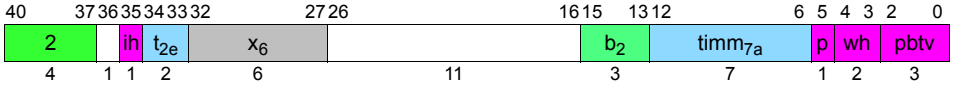
B6



Instruction	Operands	Opcode	Extension			
			x_2	ih	p	pbtv
brp.sptk.*ph.pvec.ih*	*target*$_{25}$, *tag*$_{13}$	7	0	See Table 4-60 on page 3:244	See Table 4-63 on page 3:245	See Table 4-64 on page 3:245
brp.loop.*ph.pvec.ih*			1			
brp.dptk.*ph.pvec.ih*			2			

4.5.2.2 Indirect Predict

B7



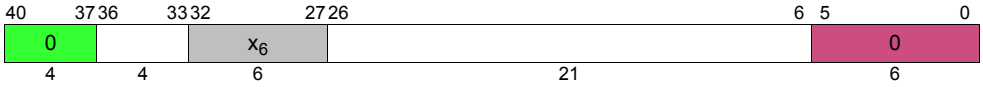
Instruction	Operands	Opcode	Extension				
			x_6	ih	p	wh	pbtv
brp.*indwh.ph.pvec.ih*	*b*$_2$, *tag*$_{13}$	2	10	See Table 4-60 on page 3:244	See Table 4-63 on page 3:245	See Table 4-62 on page 3:245	See Table 4-64 on page 3:245
brp.ret.*indwh.ph.pvec.ih*			11				

4.5.3 Miscellaneous B-Unit Instructions

The miscellaneous branch-unit instructions include a number of instructions encoded within major opcode 0 using a 6-bit opcode extension field in bits 32:27 (x_6) as described in Table 4-52 on page 3:241.

4.5.3.1 Miscellaneous (B-Unit)

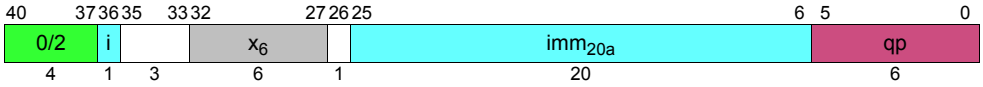
B8



Instruction	Opcode	Extension
		x_6
cover [I]	0	02
clrrrb [I]		04
clrrrb.pr [I]		05
rfi [e I p]		08
bsw.0 [I p]		0C
bsw.1 [I p]		0D
epc		10
vmsw.0 [p]	0	18
vmsw.1 [p]		19

4.5.3.2 Break/Nop/Hint (B-Unit)

B9



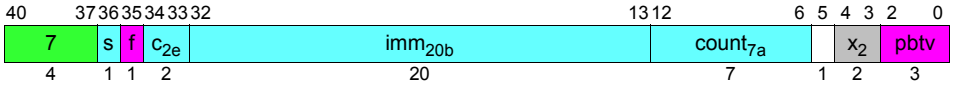
Instruction	Operands	Opcode	Extension
			x_6
break.b [e]	*imm*$_{21}$	0	00
nop.b		2	
hint.b			01

13.3.3 Instruction Prefetch

Table 1: Instruction Prefetch Flush Hint Completer

f Bit 35	*fh*
0	*none*
1	.flush

B10



Instruction	Operands	Opcode	Extension		
			x_2	f	pbtv
ifetch.*pvec.fh*	*target*$_{25}$, *count*$_9$	7	3	See Table 1 on page 3:247	See Table 4-64 on page 3:245

4.6 F-Unit Instruction Encodings

The floating-point instructions are encoded in major opcodes 8 – E for floating-point and fixed-point arithmetic, opcode 4 for floating-point compare, opcode 5 for floating-point class, and opcodes 0 and 1 for miscellaneous floating-point instructions.
The miscellaneous and reciprocal approximation floating-point instructions are encoded within major opcodes 0 and 1 using a 1-bit opcode extension field (x) in bit 33 and either a second 1-bit extension field in bit 36 (q) or a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-65 shows the 1-bit x assignments, Table 4-68 shows the additional 1-bit q assignments for the reciprocal approximation instructions; Table 4-66 and Table 4-67 summarize the 6-bit x_6 assignments.

Table 4-65.Miscellaneous Floating-point 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	
0	0	6-bit Ext (Table 4-66)
	1	Reciprocal Approximation (Table 4-68)
1	0	6-bit Ext (Table 4-67)
	1	Reciprocal Approximation (Table 4-68)

Table 4-66.Opcode 0 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	Bits 32:31 = 0	1	2	3
0	0	0	break.f F15	fmerge.s F9		
		1	1-bit Ext (Table 4-74)	fmerge.ns F9		
		2		fmerge.se F9		
		3				
		4	fsetc F12	fmin F8		fswap F9
		5	fclrf F13	fmax F8		fswap.nl F9
		6		famin F8		fswap.nr F9
		7		famax F8		
		8	fchkf F14	fcvt.fx F10	fpack F9	
		9		fcvt.fxu F10		fmix.lr F9
		A		fcvt.fx.trunc F10		fmix.r F9
		B		fcvt.fxu.trunc F10		fmix.l F9
		C		fcvt.xf F11	fand F9	fsxt.r F9
		D			fandcm F9	fsxt.l F9
		E			for F9	
		F			fxor F9	

Table 4-67.Opcode 1 Miscellaneous Floating-point 6-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	x_6 Bits 30:27	Bits 32:31 = 0	1	2	3
1	0	0		fpmerge.s F9		fpcmp.eq F8
		1		fpmerge.ns F9		fpcmp.lt F8
		2		fpmerge.se F9		fpcmp.le F8
		3				fpcmp.unord F8
		4		fpmin F8		fpcmp.neq F8
		5		fpmax F8		fpcmp.nlt F8
		6		fpamin F8		fpcmp.nle F8
		7		fpamax F8		fpcmp.ord F8
		8		fpcvt.fx F10		
		9		fpcvt.fxu F10		
		A		fpcvt.fx.trunc F10		
		B		fpcvt.fxu.trunc F10		
		C				
		D				
		E				
		F				

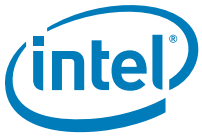


Table 4-68.Reciprocal Approximation 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit 33	q Bit 36	
0	1	0	frcpa F6
		1	frsqrta F7
1		0	fprcpa F6
		1	fprsqrta F7

Most floating-point instructions have a 2-bit opcode extension field in bits 35:34 (sf) which encodes the FPSR status field to be used. Table 4-69 summarizes these assignments.

Table 4-69.Floating-point Status Field Completer

sf Bits 35:34	*sf*
0	.s0
1	.s1
2	.s2
3	.s3

4.6.1 Arithmetic

The floating-point arithmetic instructions are encoded within major opcodes 8 – D using a 1-bit opcode extension field (x) in bit 36 and a 2-bit opcode extension field (sf) in bits 35:34. The opcode and x assignments are shown in Table 4-70.

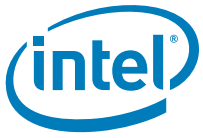
Table 4-70.Floating-point Arithmetic 1-bit Opcode Extensions

x Bit 36	Opcode Bits 40:37					
	8	9	A	B	C	D
0	fma F1	fma.d F1	fms F1	fms.d F1	fnma F1	fnma.d F1
1	fma.s F1	fpma F1	fms.s F1	fpms F1	fnma.s F1	fpnma F1

The fixed-point arithmetic and parallel floating-point select instructions are encoded within major opcode E using a 1-bit opcode extension field (x) in bit 36. The fixed-point arithmetic instructions also have a 2-bit opcode extension field (x_2) in bits 35:34. These assignments are shown in Table 4-71.

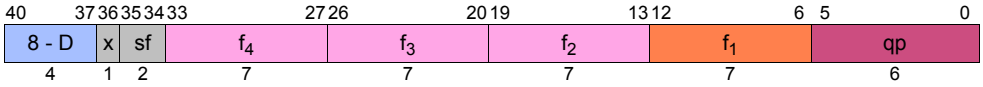
Table 4-71.Fixed-point Multiply Add and Select Opcode Extensions

Opcode Bits 40:37	x Bit 36	x_2 Bits 35:34			
		0	1	2	3
E	0	fselect F3			
	1	xma.l F2		xma.hu F2	xma.h F2



4.6.1.1 Floating-point Multiply Add

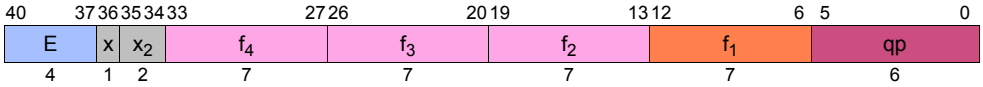
F1



Instruction	Operands	Opcode	Extension	
			x	sf
fma.*sf* fma.s.*sf*	$f_1 = f_3, f_4, f_2$	8	0 1	See Table 4-69 on page 3:249
fma.d.*sf* fpma.*sf*		9	0 1	
fms.*sf* fms.s.*sf*		A	0 1	
fms.d.*sf* fpms.*sf*		B	0 1	
fnma.*sf* fnma.s.*sf*		C	0 1	
fnma.d.*sf* fpnma.*sf*		D	0 1	

4.6.1.2 Fixed-point Multiply Add

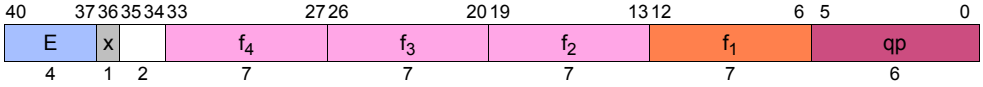
F2



Instruction	Operands	Opcode	Extension	
			x	x_2
xma.l	$f_1 = f_3, f_4, f_2$	E	1	0
xma.h				3
xma.hu				2

4.6.2 Parallel Floating-point Select

F3



Instruction	Operands	Opcode	Extension
			x
fselect	$f_1 = f_3, f_4, f_2$	E	0

4.6.3 Compare and Classify

The predicate setting floating-point compare instructions are encoded within major opcode 4 using three 1-bit opcode extension fields in bits 33 (r_a), 36 (r_b), and 12 (t_a), and a 2-bit opcode extension field (sf) in bits 35:34. The opcode, r_a, r_b, and t_a assignments are shown in Table 4-72. The sf assignments are shown in Table 4-69 on page 3:249.
The parallel floating-point compare instructions are described on page 253.

Table 4-72.Floating-point Compare Opcode Extensions

Opcode Bits 40:37	r_a Bit 33	r_b Bit 36	t_a Bit 12	
			0	1
4	0	0	fcmp.eq F4	fcmp.eq.unc F4
		1	fcmp.lt F4	fcmp.lt.unc F4
	1	0	fcmp.le F4	fcmp.le.unc F4
		1	fcmp.unord F4	fcmp.unord.unc F4

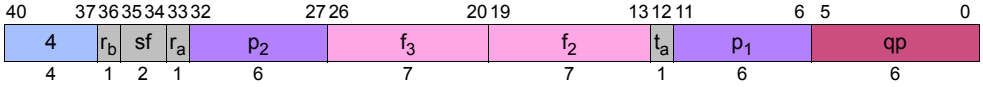
The floating-point class instructions are encoded within major opcode 5 using a 1-bit opcode extension field in bit 12 (t_a) as shown in Table 4-73.

Table 4-73.Floating-point Class 1-bit Opcode Extensions

Opcode Bits 40:37	t_a Bit 12	
5	0	fclass.m F5
	1	fclass.m.unc F5

4.6.3.1 Floating-point Compare

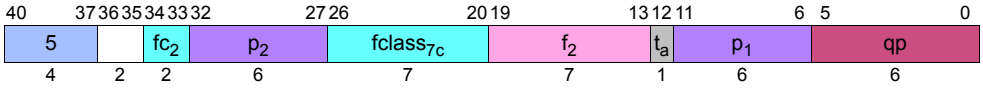
F4



Instruction	Operands	Opcode	Extension			
			r_a	r_b	t_a	sf
fcmp.eq.*sf*	$p_1, p_2 = f_2, f_3$	4	0	0	0	See Table 4-69 on page 3:249
fcmp.lt.*sf*				1		
fcmp.le.*sf*			1	0		
fcmp.unord.*sf*				1		
fcmp.eq.unc.*sf*			0	0	1	
fcmp.lt.unc.*sf*				1		
fcmp.le.unc.*sf*			1	0		
fcmp.unord.unc.*sf*				1		

4.6.3.2 Floating-point Class

F5



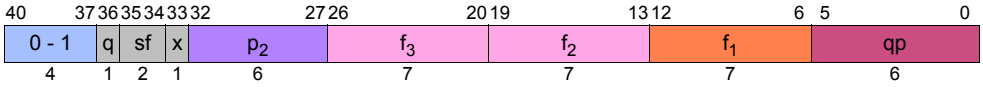
Instruction	Operands	Opcode	Extension
			t_a
fclass.m	$p_1, p_2 = f_2, fclass_9$	5	0
fclass.m.unc			1

4.6.4 Approximation

4.6.4.1 Floating-point Reciprocal Approximation

There are two Reciprocal Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

F6

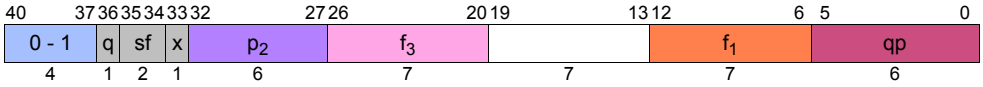


Instruction	Operands	Opcode	Extension		
			x	q	sf
frcpa.*sf* fprcpa.*sf*	$f_1, p_2 = f_2, f_3$	0 1	1	0	See Table 4-69 on page 3:249

4.6.4.2 Floating-point Reciprocal Square Root Approximation

There are two Reciprocal Square Root Approximation instructions. The first, in major op 0, encodes the full register variant. The second, in major op 1, encodes the parallel variant.

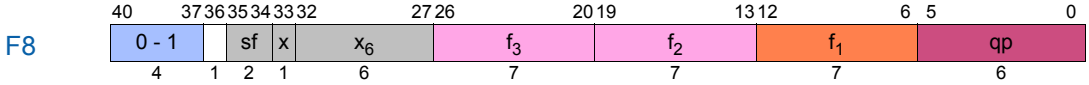
F7



Instruction	Operands	Opcode	Extension		
			x	q	sf
frsqrta.*sf* fprsqrta.*sf*	$f_1, p_2 = f_3$	0 1	1	1	See Table 4-69 on page 3:249

13.3.3 Minimum / Maximum and Parallel Compare

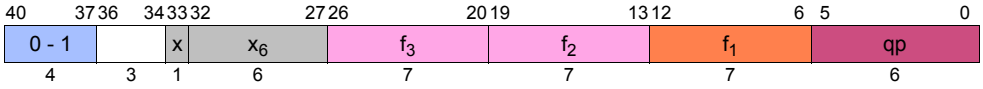
There are two groups of Minimum/Maximum instructions. The first group, in major op 0, encodes the full register variants. The second group, in major op 1, encodes the parallel variants. The parallel compare instructions are all encoded in major op 1.



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fmin.*sf*	$f_1 = f_2, f_3$	0	0	14	See Table 4-69 on page 3:249
fmax.*sf*				15	
famin.*sf*				16	
famax.*sf*				17	
fpmin.*sf*		1		14	
fpmax.*sf*				15	
fpamin.*sf*				16	
fpamax.*sf*				17	
fpcmp.eq.*sf*				30	
fpcmp.lt.*sf*				31	
fpcmp.le.*sf*				32	
fpcmp.unord.*sf*				33	
fpcmp.neq.*sf*				34	
fpcmp.nlt.*sf*				35	
fpcmp.nle.*sf*				36	
fpcmp.ord.*sf*				37	

13.3.3 Merge and Logical

F9

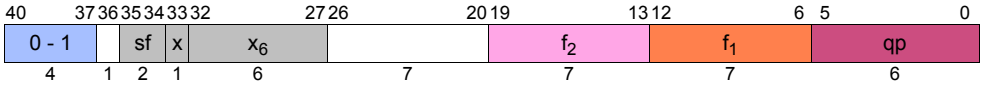


Instruction	Operands	Opcode	Extension	
			x	x_6
fmerge.s	$f_1 = f_2, f_3$	0	0	10
fmerge.ns				11
fmerge.se				12
fmix.lr				39
fmix.r				3A
fmix.l				3B
fsxt.r				3C
fsxt.l				3D
fpack				28
fswap				34
fswap.nl				35
fswap.nr				36
fand				2C
fandcm				2D
for				2E
fxor				2F
fpmerge.s		1		10
fpmerge.ns				11
fpmerge.se				12

4.6.5 Conversion

4.6.5.1 Convert Floating-point to Fixed-point

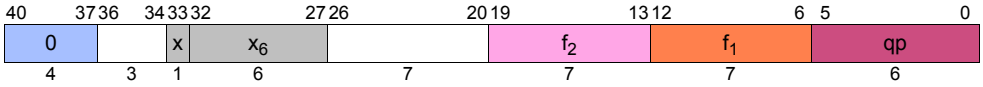
F10



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fcvt.fx.*sf*	$f_1 = f_2$	0	0	18	See Table 4-69 on page 3:249
fcvt.fxu.*sf*				19	
fcvt.fx.trunc.*sf*				1A	
fcvt.fxu.trunc.*sf*				1B	
fpcvt.fx.*sf*		1		18	
fpcvt.fxu.*sf*				19	
fpcvt.fx.trunc.*sf*				1A	
fpcvt.fxu.trunc.*sf*				1B	

4.6.5.2 Convert Fixed-point to Floating-point

F11

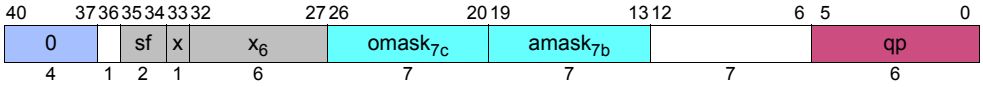


Instruction	Operands	Opcode	Extension	
			x	x_6
fcvt.xf	$f_1 = f_2$	0	0	1C

4.6.6 Status Field Manipulation

4.6.6.1 Floating-point Set Controls

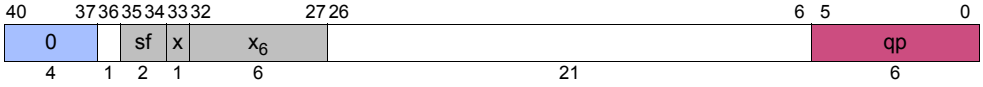
F12



Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fsetc.*sf*	$amask_7$, $omask_7$	0	0	04	See Table 4-69 on page 3:249

4.6.6.2 Floating-point Clear Flags

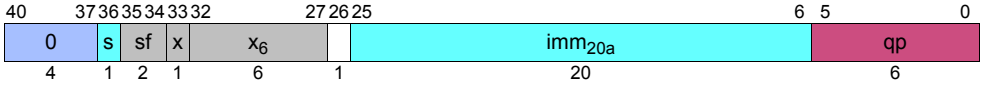
F13



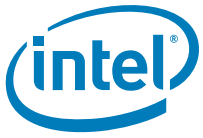
Instruction	Opcode	Extension		
		x	x_6	sf
fclrf.*sf*	0	0	05	See Table 4-69 on page 3:249

4.6.6.3 Floating-point Check Flags

F14

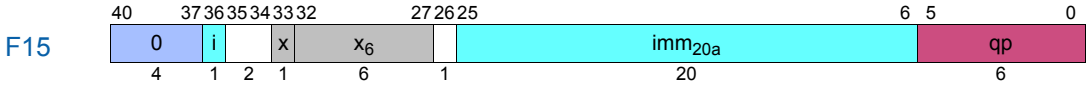


Instruction	Operands	Opcode	Extension		
			x	x_6	sf
fchkf.*sf*	$target_{25}$	0	0	08	See Table 4-69 on page 3:249



4.6.7 Miscellaneous F-Unit Instructions

4.6.7.1 Break (F-Unit)



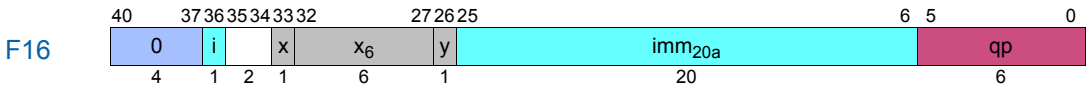
Instruction	Operands	Opcode	Extension	
			x	x_6
break.f	imm_{21}	0	0	00

4.6.7.2 Nop/Hint (F-Unit)

F-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-51.

Table 4-74.Misc F-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x Bit :33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.f
			1	hint.f



Instruction	Operands	Opcode	Extension		
			x	x_6	y
nop.f hint.f	imm_{21}	0	0	01	0 1

4.7 X-Unit Instruction Encodings

The X-unit instructions occupy two instruction slots, L+X. The major opcode, opcode extensions and hints, qp, and small immediate fields occupy the X instruction slot. For movl, break.x, and nop.x, the imm_{41} field occupies the L instruction slot. For brl, the imm_{39} field and a 2-bit Ignored field occupy the L instruction slot.

4.7.1 Miscellaneous X-Unit Instructions

The miscellaneous X-unit instructions are encoded in major opcode 0 using a 3-bit opcode extension field (x_3) in bits 35:33 and a 6-bit opcode extension field (x_6) in bits 32:27. Table 4-75 shows the 3-bit assignments and Table 4-76 summarizes the 6-bit assignments. These instructions are executed by an I-unit.

Table 4-75.Misc X-Unit 3-bit Opcode Extensions

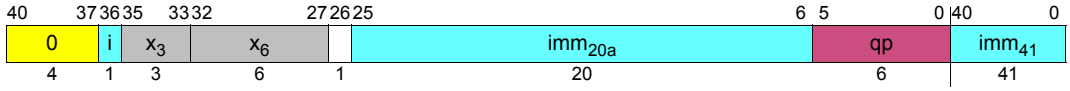
Opcode Bits 40:37	x_3 Bits 35:33	
0	0	6-bit Ext (Table 4-76)
	1	
	2	
	3	
	4	
	5	
	6	
	7	

Table 4-76.Misc X-Unit 6-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 30:27	x_6 Bits 32:31 = 0	1	2	3
0	0	0	break.x X1			
		1	1-bit Ext (Table 4-79)			
		2				
		3				
		4				
		5				
		6				
		7				
		8				
		9				
		A				
		B				
		C				
		D				
		E				
		F				

4.7.1.1 Break (X-Unit)

X1



Instruction	Operands	Opcode	Extension	
			x_3	x_6
break.x	imm_{62}	0	0	00

4.7.2 Move Long Immediate$_{64}$

The move long immediate instruction is encoded within major opcode 6 using a 1-bit reserved opcode extension in bit 20 (v_c) as shown in Table 4-77. This instruction is executed by an I-unit.

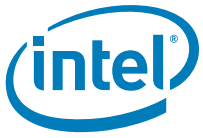
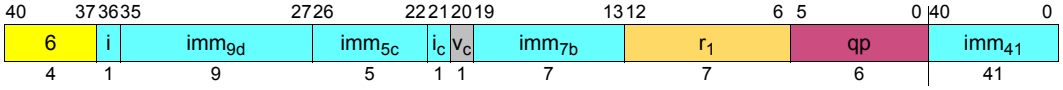


Table 4-77.Move Long 1-bit Opcode Extensions

Opcode Bits 40:37	v_c Bit 20	
6	0	movl X2
	1	

X2



Instruction	Operands	Opcode	Extension
			v_c
movl [i]	$r_1 = imm_{64}$	6	0

4.7.3 Long Branches

Long branches are executed by a B-unit. Opcode C is used for long branch and opcode D for long call. The long branch instructions encoded within major opcode C use a 3-bit opcode extension field in bits 8:6 (btype) to distinguish the branch types as shown in Table 4-78.

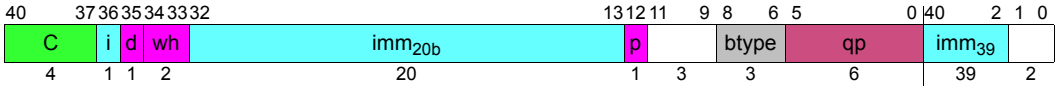
Table 4-78.Long Branch Types

Opcode Bits 40:37	btype Bits 8:6	
C	0	brl.cond X3
	1	
	2	
	3	
	4	
	5	
	6	
	7	

The long branch instructions have the same opcode hint fields in bit 12 (p), bits 34:33 (wh), and bit 35 (d) as normal IP-relative branches. These are shown in Table 4-55 on page 3:242, Table 4-56 on page 3:242, and Table 4-58 on page 3:243.

4.7.3.1 Long Branch

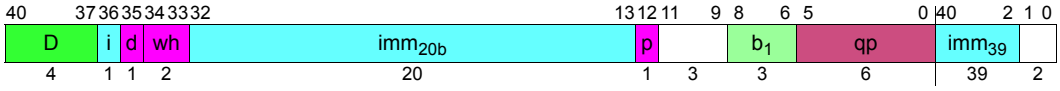
X3



Instruction	Operands	Opcode	Extension			
			btype	p	wh	d
brl.cond.*bwh.ph.dh* [e l]	$target_{64}$	C	0	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243

4.7.3.2 Long Call

X4



Instruction	Operands	Opcode	Extension		
			p	wh	d
brl.call.*bwh.ph.dh* [e l]	b_1 = *target*$_{64}$	D	See Table 4-55 on page 3:242	See Table 4-56 on page 3:242	See Table 4-58 on page 3:243

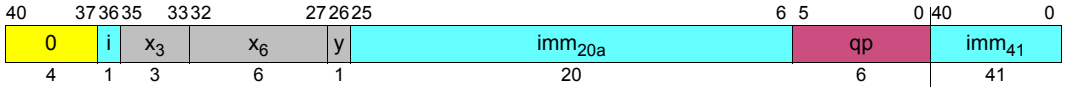
4.7.4 Nop/Hint (X-Unit)

X-unit nop and hint instructions are encoded within major opcode 0 using a 3-bit opcode extension field in bits 35:33 (x_3), a 6-bit opcode extension field in bits 32:27 (x_6), and a 1-bit opcode extension field in bit 26 (y), as shown in Table 4-79. These instructions are executed by an I-unit.

Table 4-79.Misc X-Unit 1-bit Opcode Extensions

Opcode Bits 40:37	x_3 Bits 35:33	x_6 Bits 32:27	y Bit 26	
0	0	01	0	nop.x
			1	hint.x

X5



Instruction	Operands	Opcode	Extension		
			x_3	x_6	y
nop.x	*imm*$_{62}$	0	0	01	0
hint.x					1

4.8 Immediate Formation

Table 4-80 shows, for each instruction format that has one or more immediates, how those immediates are formed. In each equation, the symbol to the left of the equals is the assembly language name for the immediate. The symbols to the right are the field names in the instruction encoding.

Table 4-80.Immediate Formation

Instruction Format	Immediate Formation
A2	$count_2 = ct_{2d} + 1$
A3 A8 I27 M30	$imm_8 = sign\_ext(s << 7 \mid imm_{7b}, 8)$
A4	$imm_{14} = sign\_ext(s << 13 \mid imm_{6d} << 7 \mid imm_{7b}, 14)$
A5	$imm_{22} = sign\_ext(s << 21 \mid imm_{5c} << 16 \mid imm_{9d} << 7 \mid imm_{7b}, 22)$
A10	$count_2 = (ct_{2d} > 2)$? reservedQP[a] : $ct_{2d} + 1$
I1	$count_2 = (ct_{2d} == 0)$? 0 : $(ct_{2d} == 1)$? 7 : $(ct_{2d} == 2)$? 15 : 16
I3	$mbtype_4 = (mbt_{4c} == 0)$? @brcst : $(mbt_{4c} == 8)$? @mix : $(mbt_{4c} == 9)$? @shuf : $(mbt_{4c} ==$ 0xA) ? @alt : $(mbt_{4c} ==$ 0xB) ? @rev : reservedQP[a]
I4	$mhtype_8 = mht_{8c}$

Table 4-80.Immediate Formation (Continued)

Instruction Format	Immediate Formation
I6	$count_5 = count_{5b}$
I8	$count_5 = 31 – ccount_{5c}$
I10	$count_6 = count_{6d}$
I11	$len_6 = len_{6d} + 1$ $pos_6 = pos_{6b}$
I12	$len_6 = len_{6d} + 1$ $pos_6 = 63 – cpos_{6c}$
I13	$len_6 = len_{6d} + 1$ $pos_6 = 63 – cpos_{6c}$ $imm_8 = sign\_ext(s << 7 \mid imm_{7b}, 8)$
I14	$len_6 = len_{6d} + 1$ $pos_6 = 63 – cpos_{6b}$ $imm_1 = sign\_ext(s, 1)$
I15	$len_4 = len_{4d} + 1$ $pos_6 = 63 – cpos_{6d}$
I16	$pos_6 = pos_{6b}$
I18 I19 M37 M55	$imm_{21} = i << 20 \mid imm_{20a}$
M49	$imm_{19} = i << 18 \mid imm_{14b} << 4 \mid imm_{4a}$
M50	$imm_{16} = i << 15 \mid imm_{11b} << 4 \mid imm_{4a}$
I21	$tag_{13} = IP + (sign\_ext(timm_{9c}, 9) << 4)$
I23	$mask_{17} = sign\_ext(s << 16 \mid mask_{8c} << 8 \mid mask_{7a} << 1, 17)$
I24	$imm_{44} = sign\_ext(s << 43 \mid imm_{27a} << 16, 44)$
I30	$imm_5 = imm_{5b} + 32$
M3 M8 M15	$imm_9 = sign\_ext(s << 8 \mid i << 7 \mid imm_{7b}, 9)$
M5 M10	$imm_9 = sign\_ext(s << 8 \mid i << 7 \mid imm_{7a}, 9)$
M17	$inc_3 = sign\_ext(((s) ? –1 : 1) * ((i_{2b} == 3) ? 1 : 1 << (4 – i_{2b})), 6)$
I20 M20 M21	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{13c} << 7 \mid imm_{7a}, 21) << 4)$
M22 M23	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{20b}, 21) << 4)$
M34	il = sol o = sof – sol r = sor << 3
M39 M40	$imm_2 = i_{2b}$
M44	$imm_{24} = i << 23 \mid i_{2d} << 21 \mid imm_{21a}$
M52	$cnt_6 = cnt_{6a} + 1$ $stride_5 = sign\_ext(stride_{5b}, 5) << 6$
B1 B2 B3	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{20b}, 21) << 4)$
B6	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{20b}, 21) << 4)$ $tag_{13} = IP + (sign\_ext(t_{2e} << 7 \mid timm_{7a}, 9) << 4)$
B7	$tag_{13} = IP + (sign\_ext(t_{2e} << 7 \mid timm_{7a}, 9) << 4)$
B9	$imm_{21} = i << 20 \mid imm_{20a}$
B10	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{20b}, 21) << 4)$ $count_9 = c_{2e} << 7 \mid count_{7a}$
F5	$fclass_9 = fclass_{7c} << 2 \mid fc_2$
F12	$amask_7 = amask_{7b}$ $omask_7 = omask_{7c}$
F14	$target_{25} = IP + (sign\_ext(s << 20 \mid imm_{20a}, 21) << 4)$
F15 F16	$imm_{21} = i << 20 \mid imm_{20a}$

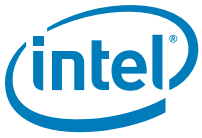
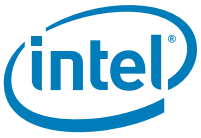


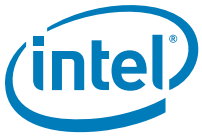
Table 4-80.Immediate Formation (Continued)

Instruction Format	Immediate Formation
X1 X5	$imm_{62} = imm_{41} << 21 \mid i << 20 \mid imm_{20a}$
X2	$imm_{64} = i << 63 \mid imm_{41} << 22 \mid i_c << 21 \mid imm_{5c} << 16 \mid imm_{9d} << 7 \mid imm_{7b}$
X3 X4	$target_{64} = IP + ((i << 59 \mid imm_{39} << 20 \mid imm_{20b}) << 4)$

a. This encoding causes an Illegal Operation fault if the value of the qualifying predicate is 1.

§





V3-M Chapter 5 Resource and Dependency Semantics

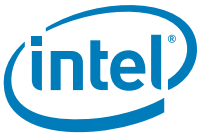
Resource and Dependency Semantics 5

5.1 Reading and Writing Resources

An Itanium instruction is said to be a **reader** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or is one of the instructions that reads a resource even when its qualifying predicate is 0, and the execution of the instruction depends on that resource.
An Itanium instruction is said to be an **writer** of a resource if the instruction's qualifying predicate is 1 or it has no qualifying predicate or writes the resource even when the qualifying predicate is 0, and the execution of the instruction writes that resource.
An Itanium instruction is said to be a reader or writer of a resource even if it only sometimes depends on that resource and it cannot be determined statically whether the resource will be read or written. For example, `cover` only writes CR[IFS] when PSR.ic is 0, but for purposes of dependency, it is treated as if it always writes the resource since this condition cannot be determined statically. On the other hand, `rsm` conditionally writes several bits in the PSR depending on a mask which is encoded as an immediate in the instruction. Since the PSR bits to be written can be determined by examining the encoded instruction, the instruction is treated as only writing those bits which have a corresponding mask bit set. All exceptions to these general rules are described in this appendix.

5.2 Dependencies and Serialization

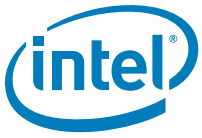
A **RAW** (Read-After-Write) dependency is a sequence of two events where the first is a writer of a resource and the second is a reader of the same resource. Events may be instructions, interruptions, or other 'uses' of the resource such as instruction stream fetches and VHPT walks. Table 5-2 covers only dependencies based on instruction readers and writers.
A **WAW** (Write-After-Write) dependency is a sequence of two events where both events write the resource in question. Events may be instructions, interruptions, or other 'updates' of the resource. Table 5-3 covers only dependencies based on instruction writers.
A **WAR** (Write-After-Read) dependency is a sequence of two instructions, where the first is a reader of a resource and the second is a writer of the same resource. Such dependencies are always allowed except as indicated in Table 5-4 and only those related to instruction readers and writers are included.
A **RAR** (Read-After-Read) dependency is a sequence of two instructions where both are readers of the same resource. Such dependencies are always allowed.
RAW and WAW dependencies are generally not allowed without some type of serialization event (an implied, data, or instruction serialization after the first writing instruction. (See Section 3.2, "Serialization" on page 2:17 for details on serialization.) The tables and associated rules in this appendix provide a comprehensive list of readers and writers of resources and describe the serialization required for the dependency to be observed and possible outcomes if the required serialization is not met. Even when targeting code for machines which do not check for particular disallowed dependencies, such code sequences are considered architecturally undefined and may cause code to behave differently across processors, operating systems, or even separate executions of the code sequence



during the same program run. In some cases, different serializations may yield different, but well-defined results.
The serialization of application level (non-privileged) resources is always implied. This means that if a writer of that resource and a subsequent read of that same resource are in different instruction groups, then the reader will see the value written. In addition, for dependencies on PRs and BRs, where the writer is a non-branch instruction and the reader is a branch instruction, the writer and reader may be in the same instruction group.
System resources generally require explicit serialization, i.e., the use of a `srlz.i` or `srlz.d` instruction, between the writing and the reading of that resource. Note that RAW accesses to CRs are not exceptional – they require explicit data or instruction serialization. However, in some cases (other than CRs) where pairs of instructions explicitly encode the same resource, serialization is implied.
There are cases where it is architecturally allowed to omit a serialization, and that the response from the CPU must be atomic (act as if either the old or the new state were fully in place). The tables in this appendix indicate dependency requirements under the assumption that the desired result is for the dependency to always be observed. In some such cases, the programmer may not care if the old or new state is used; such situations are allowed, but the value seen is not deterministic.
On the other hand, if an *impliedF* dependency is violated, then the program is incorrectly coded and the processor's behavior is undefined.

5.3 Resource and Dependency Table Format Notes

- The “Writers” and “Readers” columns of the dependency tables contain instruction class names and instruction mnemonic prefixes as given in the format section of each instruction page. To avoid ambiguity, instruction classes are shown in bold, while instruction mnemonic prefixes are in regular font. For instruction mnemonic prefixes, all instructions that exactly match the name specified or those that begin with the specified text and are followed by a ‘.’ and then followed by any other text will match.
- The dependency on a listed instruction is in effect no matter what values are encoded in the instruction or what dynamic values occur in operands, unless a superscript is present or one of the special case instruction rules in Section 5.3.1 applies. Instructions listed are still subject to rules regarding qualifying predicates.
- Instruction classes are groups of related instructions. Such names appear in boldface for clarity. The list of all instruction classes is contained in Table 5-5. Note that an instruction may appear in multiple instruction classes, instruction classes may expand to contain other classes, and that when fully expanded, a set of classes (e.g., the readers of some resource) may contain the same instruction multiple times.
- The syntax ‘**x****y**’ where **x** and **y** are both instruction classes, indicates an unnamed instruction class that includes all instructions in instruction class **x** but that are not in instruction class **y**. Similarly, the notation ‘**x****y****z**’ means all instructions in instruction class **x**, but that are not in either instruction class **y** or instruction class **z**.
- Resources on separate rows of a table are independent resources. This means that there are no serialization requirements for an event which references one of them followed by an event which uses a different resource. In cases where resources are broken into subrows, dependencies only apply between instructions within a subrow. Instructions that do not appear in a subrow together have no dependencies (reader/writer or writer/writer dependencies) for the resource in question, although they may still have dependencies on some other resource.



- The dependencies listed for pairs of instructions on each resource are not unique – the same pair of instructions might also have a dependency on some other resource with a different semantics of dependency. In cases where there are multiple resource dependencies for the same pair of instructions, the most stringent semantics are assumed: *instr* overrides *data* which overrides *impliedF* which overrides *implied* which overrides *none*.
- Arrays of numbered resources are represented in a single row of a table using the % notation as a substitute for the number of the resource. In such cases, the semantics of the table are as if each numbered resource had its own row in that table and is thus an independent resource. The range of values that the % can take are given in the "Resource Name" column.
- An asterisk '*' in the "Resource Name" column indicates that this resource may not have a physical resource associated with it, but is added to enforce special dependencies.
- A pound sign '#' in the "Resource Name" column indicates that this resource is an array of resources that are indexed by a value in a GR. The number of individual elements in the array is described in the detailed description of each resource.
- The "Semantics of Dependency" column describes the outcome given various serialization and instruction group boundary conditions. The exact definition for each keyword is given in Table 5-1.

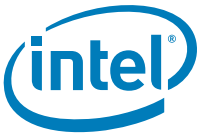
Table 5-1. Semantics of Dependency Codes

Semantics of Dependency Code	Serialization Type Required	Effects of Serialization Violation
instr	Instruction Serialization (See "Instruction Serialization" on page 18).	Atomic: Any attempt to read a resource after one or more insufficiently serialized writes is either the value previously in the register (before any of the unserialized writes) or the value of one of any unserialized writes. Which value is returned is unpredictable and multiple insufficiently serialized reads may see different results. No fault will be caused by the insufficient serialization.
data	Data Serialization (See "Data Serialization" on page 18)	
implied	Instruction Group Break. Writer and reader must be in separate instruction groups. (See "Instruction Sequencing Considerations" on page 39).	
impliedF	Instruction Group Break (same as above).	An undefined value is returned, or an Illegal Operation fault may be taken. If no fault is taken, the value returned is unpredictable, and may be unrelated to past writes, but will not be data which could not be accessed by the current process (e.g., if PSR.cpl != 0, the undefined value to return cannot be read from some control register).
stop	Stop. Writer and reader must be separated by a stop.	
none	None	N/A
specific	Implementation Specific	
SC	Special Case	Described elsewhere in book, see referenced section in the entry.

5.3.1 Special Case Instruction Rules

The following rules apply to the specified instructions when they appear in Table 5-2, Table 5-3, Table 5-4, or Table 5-5:

- An instruction always reads a given resource if its qualifying predicate is 1 and it appears in the "Reader" column of the table (except as noted). An instruction always writes a given resource if its qualifying predicate is 1 and it appears in the "Writer" column of the table (except as noted). An instruction never reads or writes



the specified resource if its qualifying predicate is 0 (except as noted). These rules include branches and their qualifying predicate. Instructions in the **unpredicatable-instructions** class have no qualifying predicate and thus always read or write their resources (except as noted).

- An instruction of type **mov-from-PR** reads all PRs if its PR[qp] is true. If the PR[qp] is false, then only the PR[qp] is read.
- An instruction of type **mov-to-PR** writes only those PRs as indicated by the immediate mask encoded in the instruction.
- A `st8.spill` only writes AR[UNAT]{*X*} where *X* equals the value in bits 8:3 of the store's data address. A `ld8.fill` instruction only reads AR[UNAT]{*Y*} where *Y* equals the value in bits 8:3 of the load's data address.
- Instructions of type **mod-sched-brs** always read AR[EC] and the rotating register base registers in CFM, and always write AR[EC], the rotating register bases in CFM, and PR[63] even if they do not change their values or if their PR[qp] is false.
- Instructions of type **mod-sched-brs-counted** always read and write AR[LC], even if they do not change its value.
- For instructions of type **pr-or-writers** or **pr-and-writers**, if their completer is `or.andcm`, then only the first target predicate is an or-compare and the second target predicate is an and-compare. Similarly, if their completer is `and.orcm`, then only the second target predicate is an or-compare and the first target predicate is an and-compare.
- `rum` and `sum` only read PSR.sp when the bit corresponding to PSR.up (bit 2) is set in the immediate field of the instruction.

5.3.2 RAW Dependency Table

Table 5-2 architecturally defines the following information:

- A list of all architecturally-defined, independently-writable resources in the Itanium architecture. Each row represents an 'atomic' resource. Thus, for each row in the table, hardware will probably require a separate write-enable control signal.
- For each resource, a complete list of readers and writers.
- For each instruction, a complete list of all resources read and written. Such a list can be obtained by taking the union of all the rows in which each instruction appears.

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

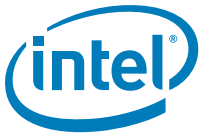


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
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**	
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

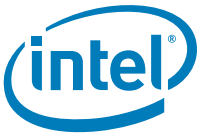


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
AR[RUC]	**mov-to-AR-RUC**	br.ia, **mov-from-AR-RUC**	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, 46-47, 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
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 124
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[IIB%], % in 0 - 1	**mov-to-CR-IIB**	**mov-from-CR-IIB**	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

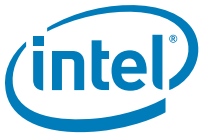


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
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[ITO]	**mov-to-CR-ITO**	**mov-from-AR-ITC, mov-from-CR-ITO**	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 123
CR[LID]	**mov-to-CR-LID**	**mov-from-CR-LID**	SC Section 5.8.3.1, “Local ID (LID – CR64)” on page 122
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[17], ssm[17]	SC Section 5.8.3.3, “Task Priority Register (TPR – CR66)” on page 123
		rfi	implied
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**	**mov-from-CR-rv**[1]	none
DAHR%, % in 0-7	br.call, brl.call, br.ret, mov-to-AR-BSPSTORE, mov-to-DAHR[1], rfi	br.call, brl.call, mem-readers, mem-writers, mov-from-DAHR[1]	implied
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

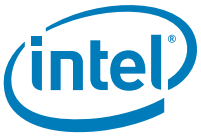


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
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**[1,13]	**gr-readers**[1]	none
	gr-writers[1]**ld-c**[1,13]	**gr-readers**[1]	impliedF
IBR#	**mov-to-IND-IBR**[3]	**mov-from-IND-IBR**[3]	impliedF
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 Section 7.2.1, “Generic Performance Counter Registers” for PMC[0].fr on page 156
PMD#	**mov-to-IND-PMD**[3]	**mov-from-IND-PMD**[3]	impliedF

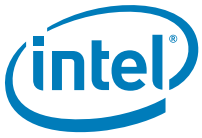


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
PR0	**pr-writers**[1]	**pr-readers-br**[1], **pr-readers-nobr-nomovpr**[1], **mov-from-PR**[12], **mov-to-PR**[12]	none
PR%, % in 1 - 15	**pr-writers**[1], **mov-to-PR-allreg**[7]	**pr-readers-nobr-nomovpr**[1], **mov-from-PR**, **mov-to-PR**[12]	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**[12]	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
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**[12]	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

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

Resource Name	Writers	Readers	Semantics of Dependency
PSR.bn	bsw, rfi	**gr-readers**[10], **gr-writers**[10]	impliedF
PSR.cpl	epc, br.ret	**priv-ops**, br.call, brl.call, epc, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **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-from-AR-RUC**, **mov-to-AR-ITC**, **mov-to-AR-RSC**, **mov-to-AR-RUC**, **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
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**[8], **fr-writers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-readers**[8], **fr-writers**[8], **mov-from-PSR**	impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**	**fr-writers**[8], **fr-readers**[8]	data
		mov-from-PSR	impliedF
	rfi	**fr-writers**[8], **fr-readers**[8], **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

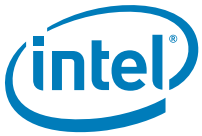


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
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
PSR.mfh	**fr-writers**[9], **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**[9], **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, **mov-from-AR-RUC**	data
	rfi	**mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-PSR**	impliedF

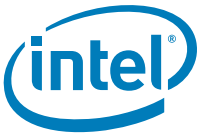


Table 5-2. RAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Readers	Semantics of Dependency
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
PSR.vm	vmsw	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	implied
	rfi	**mem-readers**, **mem-writers**, **mov-from-AR-ITC**, **mov-from-AR-RUC**, **mov-from-IND-CPUID**, **mov-to-AR-ITC**, **mov-to-AR-RUC**, **priv-ops**\vmsw, cover, thash, ttag	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**[14]	**rse-readers**[14]	impliedF

5.3.3 WAW Dependency Table

General rules specific to the WAW table:

- All resources require at most an instruction group break to provide sequential behavior.
- Some resources require no instruction group break to provide sequential behavior.
- There are a few special cases that are described in greater detail elsewhere in the manual and are indicated with an SC (special case) result.
- Each sub-row of writers represents a group of instructions that when taken in pairs in any combination has the dependency result indicated. If the column is split in sub-columns, then the dependency semantics apply to any pair of instructions where one is chosen from left sub-column and one is chosen from the right sub-column.

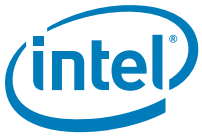


Table 5-3. WAW Dependencies Organized by Resource

Resource Name	Writers		Semantics of Dependency
ALAT	**mem-readers-alat**, **mem-writers**, chk.a.clr, **invala-all**		none
AR[BSP]	br.call, brl.call, br.ret, cover, **mov-to-AR-BSPSTORE**, rfi		impliedF
AR[BSPSTORE]	alloc, loadrs, flushrs, **mov-to-AR-BSPSTORE**		impliedF
AR[CCV]	**mov-to-AR-CCV**		impliedF
AR[CFLG]	**mov-to-AR-CFLG**		impliedF
AR[CSD]	ld16, **mov-to-AR-CSD**		impliedF
AR[EC]	br.ret, **mod-sched-brs**, **mov-to-AR-EC**		impliedF
AR[EFLAG]	**mov-to-AR-EFLAG**		impliedF
AR[FCR]	**mov-to-AR-FCR**		impliedF
AR[FDR]	**mov-to-AR-FDR**		impliedF
AR[FIR]	**mov-to-AR-FIR**		impliedF
AR[FPSR].sf0.controls	**mov-to-AR-FPSR**, fsetc.s0		impliedF
AR[FPSR].sf1.controls	**mov-to-AR-FPSR**, fsetc.s1		impliedF
AR[FPSR].sf2.controls	**mov-to-AR-FPSR**, fsetc.s2		impliedF
AR[FPSR].sf3.controls	**mov-to-AR-FPSR**, fsetc.s3		impliedF
AR[FPSR].sf0.flags	**fp-arith-s0**, **fcmp-s0**, **fpcmp-s0**		none
	fclrf.s0, **fcmp-s0**, **fp-arith-s0**, **fpcmp-s0**, **mov-to-AR-FPSR**	fclrf.s0, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf1.flags	**fp-arith-s1**, **fcmp-s1**, **fpcmp-s1**		none
	fclrf.s1, **fcmp-s1**, **fp-arith-s1**, **fpcmp-s1**, **mov-to-AR-FPSR**	fclrf.s1, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf2.flags	**fp-arith-s2**, **fcmp-s2**, **fpcmp-s2**		none
	fclrf.s2, **fcmp-s2**, **fp-arith-s2**, **fpcmp-s2**, **mov-to-AR-FPSR**	fclrf.s2, **mov-to-AR-FPSR**	impliedF
AR[FPSR].sf3.flags	**fp-arith-s3**, **fcmp-s3**, **fpcmp-s3**		none
	fclrf.s3, **fcmp-s3**, **fp-arith-s3**, **fpcmp-s3**, **mov-to-AR-FPSR**	fclrf.s3, **mov-to-AR-FPSR**	impliedF
AR[FPSR].rv	**mov-to-AR-FPSR**		impliedF
AR[FPSR].traps	**mov-to-AR-FPSR**		impliedF
AR[FSR]	**mov-to-AR-FSR**		impliedF
AR[ITC]	**mov-to-AR-ITC**		impliedF
AR[K%], % in 0 - 7	**mov-to-AR-K**[1]		impliedF
AR[LC]	**mod-sched-brs-counted**, **mov-to-AR-LC**		impliedF
AR[PFS]	br.call, brl.call		none
	br.call, brl.call	**mov-to-AR-PFS**	impliedF
AR[RNAT]	alloc, flushrs, loadrs, **mov-to-AR-RNAT**, **mov-to-AR-BSPSTORE**		impliedF
AR[RSC]	**mov-to-AR-RSC**		impliedF
AR[RUC]	**mov-to-AR-RUC**		impliedF
AR[SSD]	**mov-to-AR-SSD**		impliedF
AR[UNAT]{%}, % in 0 - 63	**mov-to-AR-UNAT**, st8.spill		impliedF

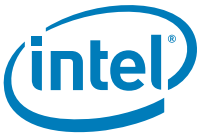


Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
AR%, % in 8-15, 20, 22-23, 31, 33-35, 37-39, 41-43, 46-47, 67-111	**none**		none
AR%, % in 48 - 63, 112-127	**mov-to-AR-ig**[1]		impliedF
BR%, % in 0 - 7	br.call[1], brl.call[1]	**mov-to-BR**[1]	impliedF
	mov-to-BR[1]		impliedF
	br.call[1], brl.call[1]		none
CFM	**mod-sched-brs**, br.call, brl.call, br.ret, alloc, clrrrb, cover, rfi		impliedF
CPUID#	**none**		none
CR[CMCV]	**mov-to-CR-CMCV**		impliedF
CR[DCR]	**mov-to-CR-DCR**		impliedF
CR[EOI]	**mov-to-CR-EOI**		SC Section 5.8.3.4, “End of External Interrupt Register (EOI – CR67)” on page 124
CR[GPTA]	**mov-to-CR-GPTA**		impliedF
CR[IFA]	**mov-to-CR-IFA**		impliedF
CR[IFS]	**mov-to-CR-IFS**, cover		impliedF
CR[IHA]	**mov-to-CR-IHA**		impliedF
CR[IIB%], % in 0 - 1	**mov-to-CR-IIB**		impliedF
CR[IIM]	**mov-to-CR-IIM**		impliedF
CR[IIP]	**mov-to-CR-IIP**		impliedF
CR[IIPA]	**mov-to-CR-IIPA**		impliedF
CR[IPSR]	**mov-to-CR-IPSR**		impliedF
CR[IRR%], % in 0 - 3	**mov-from-CR-IVR**		impliedF
CR[ISR]	**mov-to-CR-ISR**		impliedF
CR[ITIR]	**mov-to-CR-ITIR**		impliedF
CR[ITM]	**mov-to-CR-ITM**		impliedF
CR[ITO]	**mov-to-CR-ITO**		impliedF
CR[ITV]	**mov-to-CR-ITV**		impliedF
CR[IVA]	**mov-to-CR-IVA**		impliedF
CR[IVR]	**none**		SC
CR[LID]	**mov-to-CR-LID**		SC
CR[LRR%], % in 0 - 1	**mov-to-CR-LRR**[1]		impliedF
CR[PMV]	**mov-to-CR-PMV**		impliedF
CR[PTA]	**mov-to-CR-PTA**		impliedF
CR[TPR]	**mov-to-CR-TPR**		impliedF
CR%, % in 3, 5-7, 10-15, 18, 28-63, 75-79, 82-127	**none**		none

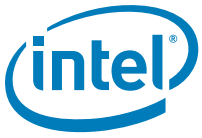


Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
DAHR%, % in 0-7	br.call, brl.call, br.ret, **mov-to-AR-BSPSTORE**, **mov-to-DAHR**, rfi		implied
DBR#	**mov-to-IND-DBR**[3]		impliedF
DTC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
DTC_LIMIT*	ptc.g, ptc.ga		impliedF
DTR	itr.d		impliedF
	itr.d	ptr.d	impliedF
	ptr.d		none
FR%, % in 0 - 1	**none**		none
FR%, % in 2 - 127	**fr-writers**[1], **ldf-c**[1], **ldfp-c**[1]		impliedF
GR0	**none**		none
GR%, % in 1 - 127	**ld-c**[1], **gr-writers**[1]		impliedF
IBR#	**mov-to-IND-IBR**[3]		impliedF
InService*	**mov-to-CR-EOI**, **mov-from-CR-IVR**		SC
IP	**all**		none
ITC	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d		none
	ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, itc.i, itc.d, itr.i, itr.d	itc.i, itc.d, itr.i, itr.d	impliedF
ITR	itr.i	itr.i, ptr.i	impliedF
	ptr.i		none
memory	**mem-writers**		none
PKR#	**mov-to-IND-PKR**[3]	**mov-to-IND-PKR**[4]	none
	mov-to-IND-PKR[3]		impliedF
PMC#	**mov-to-IND-PMC**[3]		impliedF
PMD#	**mov-to-IND-PMD**[3]		impliedF
PR0	**pr-writers**[1]		none
PR%, % in 1 - 15	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7]	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7]	impliedF
PR%, % in 16 - 62	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	pr-unc-writers-fp[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF

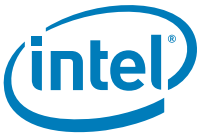


Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers		Semantics of Dependency
PR63	**pr-and-writers**[1]		none
	pr-or-writers[1]		none
	mod-sched-brs, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-and-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	**mod-sched-brs**, **pr-unc-writers-fp**[1], **pr-unc-writers-int**[1], **pr-norm-writers-fp**[1], **pr-norm-writers-int**[1], **pr-or-writers**[1], **mov-to-PR-allreg**[7], **mov-to-PR-rotreg**	impliedF
PSR.ac	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.be	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.bn	bsw, rfi		impliedF
PSR.cpl	epc, br.ret, rfi		impliedF
PSR.da	rfi		impliedF
PSR.db	**mov-to-PSR-l**, rfi		impliedF
PSR.dd	rfi		impliedF
PSR.dfh	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dfl	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.di	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.dt	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ed	rfi		impliedF
PSR.i	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ia	rfi		impliedF
PSR.ic	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.id	rfi		impliedF
PSR.is	br.ia, rfi		impliedF
PSR.it	rfi		impliedF
PSR.lp	**mov-to-PSR-l**, rfi		impliedF
PSR.mc	rfi		impliedF
PSR.mfh	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.mfl	**fr-writers**[9]		none
	user-mask-writers-partial[7], **mov-to-PSR-um**, **fr-writers**[9], **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.pk	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.pp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.ri	rfi		impliedF
PSR.rt	**mov-to-PSR-l**, rfi		impliedF
PSR.si	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF
PSR.sp	**sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi		impliedF

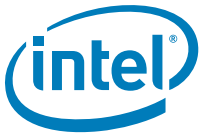


Table 5-3. WAW Dependencies Organized by Resource (Continued)

Resource Name	Writers	Semantics of Dependency
PSR.ss	rfi	impliedF
PSR.tb	**mov-to-PSR-l**, rfi	impliedF
PSR.up	**user-mask-writers-partial**[7], **mov-to-PSR-um**, **sys-mask-writers-partial**[7], **mov-to-PSR-l**, rfi	impliedF
PSR.vm	rfi, vmsw	impliedF
RR#	**mov-to-IND-RR**[6]	impliedF
RSE	**rse-writers**[14]	impliedF

5.3.4 WAR Dependency Table

A general rule specific to the WAR table:

1. WAR dependencies are always allowed within instruction groups except for the entry in Table 5-4 below. The readers and subsequent writers specified must be separated by a stop in order to have defined behavior.

Table 5-4. WAR Dependencies Organized by Resource

Resource Name	Readers	Writers	Semantics of Dependency
PR63	**pr-readers-br**[1]	**mod-sched-brs**	stop

5.3.5 Listing of Rules Referenced in Dependency Tables

The following rules restrict the specific instances in which some of the instructions in the tables cause a dependency and must be applied where referenced to correctly interpret those entries. Rules only apply to the instance of the instruction class, or instruction mnemonic prefix where the rule is referenced as a superscript. If the rule is referenced in Table 5-5 where instruction classes are defined, then it applies to all instances of the instruction class.

Rule 1. These instructions only write a register when that register's number is explicitly encoded as a target of the instruction and is only read when it is encoded as a source of the instruction (or encoded as its PR[qp]).

Rule 2. These instructions only read CFM when they access a rotating GR, FR, or PR. **mov-to-PR** and **mov-from-PR** only access CFM when their qualifying predicate is in the rotating region.

Rule 3. These instructions use a general register value to determine the specific indirect register accessed. These instructions only access the register resource specified by the value in bits {7:0} of the dynamic value of the index register.

Rule 4. These instructions only read the given resource when bits {7:0} of the indirect index register value *does not* match the register number of the resource.

Rule 5. All rules are implementation specific.

Rule 6. There is a dependency only when both the index specified by the reader and the index specified by the writer have the same value in bits {63:61}.

Rule 7. These instructions access the specified resource only when the corresponding mask bit is set.

Rule 8. PSR.dfh is only read when these instructions reference FR32-127. PSR.dfl is only read when these instructions reference FR2-31.

Rule 9. PSR.mfl is only written when these instructions write FR2-31. PSR.mfh is only written when these instructions write FR32-127.

Rule 10. The PSR.bn bit is only accessed when one of GR16-31 is specified in the instruction.

Rule 11. The target predicates are written independently of PR[qp], but source registers are only read if PR[qp] is true.

Rule 12. This instruction only reads the specified predicate register when that register is the PR[qp].

Rule 13. This reference to ld-c only applies to the GR whose value is loaded with data returned from memory, not the post-incremented address register. Thus, a stop is still required between a post-incrementing ld-c and a consumer that reads the post-incremented GR.

Rule 14. The RSE resource includes implementation-specific internal state. At least one (and possibly more) of these resources are read by each instruction listed in the **rse-readers** class. At least one (and possibly more) of these resources are written by each instruction listed in the **rse-writers** class. To determine exactly which instructions read or write each individual resource, see the corresponding instruction pages.

Rule 15. This class represents all instructions marked as Reserved if PR[qp] is 1 B-type instructions as described in “Format Summary” on page 294.

Rule 16. This class represents all instructions marked as Reserved if PR[qp] is 1 instructions as described in “Format Summary” on page 294.

Rule 17. 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.

5.4 Support Tables

Table 5-5. Instruction Classes

Class	Events/Instructions
all	**predicatable-instructions**, **unpredicatable-instructions**
branches	**indirect-brs**, **ip-rel-brs**
cfm-readers	**fr-readers**, **fr-writers**, **gr-readers**, **gr-writers**, **mod-sched-brs**, **predicatable-instructions**, **pr-writers**, alloc, br.call, brl.call, br.ret, cover, loadrs, rfi, **chk-a**, invala.e
chk-a	chk.a.clr, chk.a.nc
cmpxchg	cmpxchg1, cmpxchg2, cmpxchg4, cmpxchg8, cmp8xchg16
czx	czx1, czx2
fcmp-s0	fcmp[Field(sf)==s0]
fcmp-s1	fcmp[Field(sf)==s1]
fcmp-s2	fcmp[Field(sf)==s2]
fcmp-s3	fcmp[Field(sf)==s3]
fetchadd	fetchadd4, fetchadd8

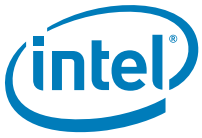


Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
fp-arith	fadd, famax, famin, fcvt.fx, fcvt.fxu, fcvt.xuf, fma, fmax, fmin, fmpy, fms, fnma, fnmpy, fnorm, fpamax, fpamin, fpcvt.fx, fpcvt.fxu, fpma, fpmax, fpmin, fpmpy, fpms, fpnma, fpnmpy, fprcpa, fprsqrta, frcpa, frsqrta, fsub
fp-arith-s0	**fp-arith**[Field(sf)==s0]
fp-arith-s1	**fp-arith**[Field(sf)==s1]
fp-arith-s2	**fp-arith**[Field(sf)==s2]
fp-arith-s3	**fp-arith**[Field(sf)==s3]
fp-non-arith	fabs, fand, fandcm, fclass, fcvt.xf, fmerge, fmix, fneg, fnegabs, for, fpabs, fpmerge, fpack, fpneg, fpnegabs, fselect, fswap, fsxt, fxor, xma, xmpy
fpcmp-s0	fpcmp[Field(sf)==s0]
fpcmp-s1	fpcmp[Field(sf)==s1]
fpcmp-s2	fpcmp[Field(sf)==s2]
fpcmp-s3	fpcmp[Field(sf)==s3]
fr-readers	**fp-arith**, **fp-non-arith**, **mem-writers-fp**, **pr-writers-fp**, chk.s[Format in {M21}], getf
fr-writers	**fp-arith**, **fp-non-arith**\fclass, **mem-readers-fp**, setf
gr-readers	**gr-readers-writers**, **mem-readers**, **mem-writers**, chk.s, cmp, cmp4, fc, itc.i, itc.d, itr.i, itr.d, **mov-to-AR-gr**, **mov-to-BR**, **mov-to-CR**, **mov-to-IND**, **mov-from-IND**, **mov-to-PR-allreg**, **mov-to-PSR-l**, **mov-to-PSR-um**, **probe-all**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, setf, tbit, tnat
gr-readers-writers	**mov-from-IND**, add, addl, addp4, adds, and, andcm, clz, **czx**, dep\dep[Format in {I13}], extr, **mem-readers-int**, **ld-all-postinc**, **lfetch-postinc**, **mix**, **mux**, or, **pack**, **padd**, **pavg**, **pavgsub**, **pcmp**, **pmax**, **pmin**, **pmpy**, **pmpyshr**, popcnt, **probe-regular**, **psad**, **pshl**, **pshladd**, **pshr**, **pshradd**, **psub**, shl, shladd, shladdp4, shr, shrp, **st-postinc**, sub, **sxt**, tak, thash, tpa, ttag, **unpack**, xor, **zxt**
gr-writers	alloc, dep, getf, **gr-readers-writers**, **mem-readers-int**, **mov-from-AR**, **mov-from-BR**, **mov-from-CR**, **mov-from-PR**, **mov-from-PSR**, **mov-from-PSR-um**, **mov-ip**, movl
indirect-brp	brp[Format in {B7}]
indirect-brs	br.call[Format in {B5}], br.cond[Format in {B4}], br.ia, br.ret
invala-all	invala[Format in {M24}], invala.e
ip-rel-brs	**mod-sched-brs**, br.call[Format in {B3}], brl.call, brl.cond, br.cond[Format in {B1}], br.cloop
ld	ld1, ld2, ld4, ld8, ld8.fill, ld16
ld-a	ld1.a, ld2.a, ld4.a, ld8.a
ld-all-postinc	**ld**[Format in {M2 M3}], **ldfp**[Format in {M12}], **ldf**[Format in {M7 M8}]
ld-c	**ld-c-nc**, **ld-c-clr**
ld-c-clr	ld1.c.clr, ld2.c.clr, ld4.c.clr, ld8.c.clr, **ld-c-clr-acq**
ld-c-clr-acq	ld1.c.clr.acq, ld2.c.clr.acq, ld4.c.clr.acq, ld8.c.clr.acq
ld-c-nc	ld1.c.nc, ld2.c.nc, ld4.c.nc, ld8.c.nc
ld-s	ld1.s, ld2.s, ld4.s, ld8.s
ld-sa	ld1.sa, ld2.sa, ld4.sa, ld8.sa
ldf	ldfs, ldfd, ldfe, ldf8, ldf.fill
ldf-a	ldfs.a, ldfd.a, ldfe.a, ldf8.a
ldf-c	**ldf-c-nc**, **ldf-c-clr**
ldf-c-clr	ldfs.c.clr, ldfd.c.clr, ldfe.c.clr, ldf8.c.clr
ldf-c-nc	ldfs.c.nc, ldfd.c.nc, ldfe.c.nc, ldf8.c.nc
ldf-s	ldfs.s, ldfd.s, ldfe.s, ldf8.s
ldf-sa	ldfs.sa, ldfd.sa, ldfe.sa, ldf8.sa
ldfp	ldfps, ldfpd, ldfp8
ldfp-a	ldfps.a, ldfpd.a, ldfp8.a

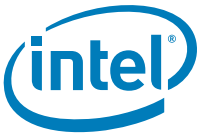


Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
ldfp-c	**ldfp-c-nc**, **ldfp-c-clr**
ldfp-c-clr	ldfps.c.clr, ldfpd.c.clr, ldfp8.c.clr
ldfp-c-nc	ldfps.c.nc, ldfpd.c.nc, ldfp8.c.nc
ldfp-s	ldfps.s, ldfpd.s, ldfp8.s
ldfp-sa	ldfps.sa, ldfpd.sa, ldfp8.sa
lfetch-all	lfetch
lfetch-fault	lfetch[Field(lftype)==fault]
lfetch-nofault	lfetch[Field(lftype)==]
lfetch-postinc	lfetch[Format in {M20 M22}]
mem-readers	**mem-readers-fp**, **mem-readers-int**
mem-readers-alat	**ld-a**, **ldf-a**, **ldfp-a**, **ld-sa**, **ldf-sa**, **ldfp-sa**, **ld-c**, **ldf-c**, **ldfp-c**
mem-readers-fp	**ldf**, **ldfp**
mem-readers-int	**cmpxchg**, **fetchadd**, **xchg**, **ld**
mem-readers-spec	**ld-s**, **ld-sa**, **ldf-s**, **ldf-sa**, **ldfp-s**, **ldfp-sa**
mem-writers	**mem-writers-fp**, **mem-writers-int**
mem-writers-fp	**stf**
mem-writers-int	**cmpxchg**, **fetchadd**, **xchg**, **st**
mix	mix1, mix2, mix4
mod-sched-brs	br.cexit, br.ctop, br.wexit, br.wtop
mod-sched-brs-counted	br.cexit, br.cloop, br.ctop
mov-from-AR	**mov-from-AR-M**, **mov-from-AR-I**, **mov-from-AR-IM**
mov-from-AR-BSP	**mov-from-AR-M**[Field(ar3) == BSP]
mov-from-AR-BSPSTORE	**mov-from-AR-M**[Field(ar3) == BSPSTORE]
mov-from-AR-CCV	**mov-from-AR-M**[Field(ar3) == CCV]
mov-from-AR-CFLG	**mov-from-AR-M**[Field(ar3) == CFLG]
mov-from-AR-CSD	**mov-from-AR-M**[Field(ar3) == CSD]
mov-from-AR-EC	**mov-from-AR-I**[Field(ar3) == EC]
mov-from-AR-EFLAG	**mov-from-AR-M**[Field(ar3) == EFLAG]
mov-from-AR-FCR	**mov-from-AR-M**[Field(ar3) == FCR]
mov-from-AR-FDR	**mov-from-AR-M**[Field(ar3) == FDR]
mov-from-AR-FIR	**mov-from-AR-M**[Field(ar3) == FIR]
mov-from-AR-FPSR	**mov-from-AR-M**[Field(ar3) == FPSR]
mov-from-AR-FSR	**mov-from-AR-M**[Field(ar3) == FSR]
mov-from-AR-I	mov_ar[Format in {I28}]
mov-from-AR-ig	**mov-from-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-from-AR-IM	mov_ar[Format in {I28 M31}]
mov-from-AR-ITC	**mov-from-AR-M**[Field(ar3) == ITC]
mov-from-AR-K	**mov-from-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-from-AR-LC	**mov-from-AR-I**[Field(ar3) == LC]
mov-from-AR-M	mov_ar[Format in {M31}]
mov-from-AR-PFS	**mov-from-AR-I**[Field(ar3) == PFS]
mov-from-AR-RNAT	**mov-from-AR-M**[Field(ar3) == RNAT]
mov-from-AR-RSC	**mov-from-AR-M**[Field(ar3) == RSC]
mov-from-AR-RUC	**mov-from-AR-M**[Field(ar3) == RUC]
mov-from-AR-rv	**none**

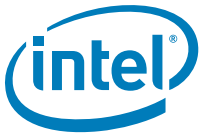


Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-from-AR-SSD	**mov-from-AR-M**[Field(ar3) == SSD]
mov-from-AR-UNAT	**mov-from-AR-M**[Field(ar3) == UNAT]
mov-from-BR	mov_br[Format in {I22}]
mov-from-CR	mov_cr[Format in {M33}]
mov-from-CR-CMCV	**mov-from-CR**[Field(cr3) == CMCV]
mov-from-CR-DCR	**mov-from-CR**[Field(cr3) == DCR]
mov-from-CR-EOI	**mov-from-CR**[Field(cr3) == EOI]
mov-from-CR-GPTA	**mov-from-CR**[Field(cr3) == GPTA]
mov-from-CR-IFA	**mov-from-CR**[Field(cr3) == IFA]
mov-from-CR-IFS	**mov-from-CR**[Field(cr3) == IFS]
mov-from-CR-IHA	**mov-from-CR**[Field(cr3) == IHA]
mov-from-CR-IIB	**mov-from-CR**[Field(cr3) in {IIB0 IIB1}]
mov-from-CR-IIM	**mov-from-CR**[Field(cr3) == IIM]
mov-from-CR-IIP	**mov-from-CR**[Field(cr3) == IIP]
mov-from-CR-IIPA	**mov-from-CR**[Field(cr3) == IIPA]
mov-from-CR-IPSR	**mov-from-CR**[Field(cr3) == IPSR]
mov-from-CR-IRR	**mov-from-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-from-CR-ISR	**mov-from-CR**[Field(cr3) == ISR]
mov-from-CR-ITIR	**mov-from-CR**[Field(cr3) == ITIR]
mov-from-CR-ITM	**mov-from-CR**[Field(cr3) == ITM]
mov-from-CR-ITO	**mov-from-CR**[Field(cr3) == ITO]
mov-from-CR-ITV	**mov-from-CR**[Field(cr3) == ITV]
mov-from-CR-IVA	**mov-from-CR**[Field(cr3) == IVA]
mov-from-CR-IVR	**mov-from-CR**[Field(cr3) == IVR]
mov-from-CR-LID	**mov-from-CR**[Field(cr3) == LID]
mov-from-CR-LRR	**mov-from-CR**[Field(cr3) in {LRR0 LRR1}]
mov-from-CR-PMV	**mov-from-CR**[Field(cr3) == PMV]
mov-from-CR-PTA	**mov-from-CR**[Field(cr3) == PTA]
mov-from-CR-rv	**none**
mov-from-CR-TPR	**mov-from-CR**[Field(cr3) == TPR]
mov-from-DAHR	mov_indirect[Format in {M43}]
mov-from-IND	mov_indirect[Format in {M43}]
mov-from-IND-CPUID	**mov-from-IND**[Field(ireg) == cpuid]
mov-from-IND-DBR	**mov-from-IND**[Field(ireg) == dbr]
mov-from-IND-IBR	**mov-from-IND**[Field(ireg) == ibr]
mov-from-IND-PKR	**mov-from-IND**[Field(ireg) == pkr]
mov-from-IND-PMC	**mov-from-IND**[Field(ireg) == pmc]
mov-from-IND-PMD	**mov-from-IND**[Field(ireg) == pmd]
mov-from-IND-priv	**mov-from-IND**[Field(ireg) in {dbr ibr pkr pmc rr}]
mov-from-IND-RR	**mov-from-IND**[Field(ireg) == rr]
mov-from-interruption-CR	**mov-from-CR-ITIR**, **mov-from-CR-IFS**, **mov-from-CR-IIB**, **mov-from-CR-IIM**, **mov-from-CR-IIP**, **mov-from-CR-IPSR**, **mov-from-CR-ISR**, **mov-from-CR-IFA**, **mov-from-CR-IHA**, **mov-from-CR-IIPA**
mov-from-PR	mov_pr[Format in {I25}]
mov-from-PSR	mov_psr[Format in {M36}]
mov-from-PSR-um	mov_um[Format in {M36}]

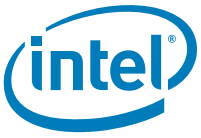


Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-ip	mov_ip[Format in {I25}]
mov-to-AR	**mov-to-AR-M**, **mov-to-AR-I**
mov-to-AR-BSP	**mov-to-AR-M**[Field(ar3) == BSP]
mov-to-AR-BSPSTORE	**mov-to-AR-M**[Field(ar3) == BSPSTORE]
mov-to-AR-CCV	**mov-to-AR-M**[Field(ar3) == CCV]
mov-to-AR-CFLG	**mov-to-AR-M**[Field(ar3) == CFLG]
mov-to-AR-CSD	**mov-to-AR-M**[Field(ar3) == CSD]
mov-to-AR-EC	**mov-to-AR-I**[Field(ar3) == EC]
mov-to-AR-EFLAG	**mov-to-AR-M**[Field(ar3) == EFLAG]
mov-to-AR-FCR	**mov-to-AR-M**[Field(ar3) == FCR]
mov-to-AR-FDR	**mov-to-AR-M**[Field(ar3) == FDR]
mov-to-AR-FIR	**mov-to-AR-M**[Field(ar3) == FIR]
mov-to-AR-FPSR	**mov-to-AR-M**[Field(ar3) == FPSR]
mov-to-AR-FSR	**mov-to-AR-M**[Field(ar3) == FSR]
mov-to-AR-gr	**mov-to-AR-M**[Format in {M29}], **mov-to-AR-I**[Format in {I26}]
mov-to-AR-I	mov_ar[Format in {I26 I27}]
mov-to-AR-ig	**mov-to-AR-IM**[Field(ar3) in {48-63 112-127}]
mov-to-AR-IM	mov_ar[Format in {I26 I27 M29 M30}]
mov-to-AR-ITC	**mov-to-AR-M**[Field(ar3) == ITC]
mov-to-AR-K	**mov-to-AR-M**[Field(ar3) in {K0 K1 K2 K3 K4 K5 K6 K7}]
mov-to-AR-LC	**mov-to-AR-I**[Field(ar3) == LC]
mov-to-AR-M	mov_ar[Format in {M29 M30}]
mov-to-AR-PFS	**mov-to-AR-I**[Field(ar3) == PFS]
mov-to-AR-RNAT	**mov-to-AR-M**[Field(ar3) == RNAT]
mov-to-AR-RSC	**mov-to-AR-M**[Field(ar3) == RSC]
mov-to-AR-RUC	**mov-to-AR-M**[Field(ar3) == RUC]
mov-to-AR-SSD	**mov-to-AR-M**[Field(ar3) == SSD]
mov-to-AR-UNAT	**mov-to-AR-M**[Field(ar3) == UNAT]
mov-to-BR	mov_br[Format in {I21}]
mov-to-CR	mov_cr[Format in {M32}]
mov-to-CR-CMCV	**mov-to-CR**[Field(cr3) == CMCV]
mov-to-CR-DCR	**mov-to-CR**[Field(cr3) == DCR]
mov-to-CR-EOI	**mov-to-CR**[Field(cr3) == EOI]
mov-to-CR-GPTA	**mov-to-CR**[Field(cr3) == GPTA]
mov-to-CR-IFA	**mov-to-CR**[Field(cr3) == IFA]
mov-to-CR-IFS	**mov-to-CR**[Field(cr3) == IFS]
mov-to-CR-IHA	**mov-to-CR**[Field(cr3) == IHA]
mov-to-CR-IIB	**mov-to-CR**[Field(cr3) in {IIB0 IIB1}]
mov-to-CR-IIM	**mov-to-CR**[Field(cr3) == IIM]
mov-to-CR-IIP	**mov-to-CR**[Field(cr3) == IIP]
mov-to-CR-IIPA	**mov-to-CR**[Field(cr3) == IIPA]
mov-to-CR-IPSR	**mov-to-CR**[Field(cr3) == IPSR]
mov-to-CR-IRR	**mov-to-CR**[Field(cr3) in {IRR0 IRR1 IRR2 IRR3}]
mov-to-CR-ISR	**mov-to-CR**[Field(cr3) == ISR]
mov-to-CR-ITIR	**mov-to-CR**[Field(cr3) == ITIR]

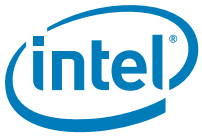


Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
mov-to-CR-ITM	**mov-to-CR**[Field(cr3) == ITM]
mov-to-CR-ITO	**mov-to-CR**[Field(cr3) == ITO]
mov-to-CR-ITV	**mov-to-CR**[Field(cr3) == ITV]
mov-to-CR-IVA	**mov-to-CR**[Field(cr3) == IVA]
mov-to-CR-IVR	**mov-to-CR**[Field(cr3) == IVR]
mov-to-CR-LID	**mov-to-CR**[Field(cr3) == LID]
mov-to-CR-LRR	**mov-to-CR**[Field(cr3) in {LRR0 LRR1}]
mov-to-CR-PMV	**mov-to-CR**[Field(cr3) == PMV]
mov-to-CR-PTA	**mov-to-CR**[Field(cr3) == PTA]
mov-to-CR-TPR	**mov-to-CR**[Field(cr3) == TPR]
mov-to-DAHR	mov_dahr[Format in {M58}]
mov-to-IND	mov_indirect[Format in {M42}]
mov-to-IND-CPUID	**mov-to-IND**[Field(ireg) == cpuid]
mov-to-IND-DBR	**mov-to-IND**[Field(ireg) == dbr]
mov-to-IND-IBR	**mov-to-IND**[Field(ireg) == ibr]
mov-to-IND-PKR	**mov-to-IND**[Field(ireg) == pkr]
mov-to-IND-PMC	**mov-to-IND**[Field(ireg) == pmc]
mov-to-IND-PMD	**mov-to-IND**[Field(ireg) == pmd]
mov-to-IND-priv	**mov-to-IND**
mov-to-IND-RR	**mov-to-IND**[Field(ireg) == rr]
mov-to-interruption-CR	**mov-to-CR-ITIR**, **mov-to-CR-IFS**, **mov-to-CR-IIB**, **mov-to-CR-IIM**, **mov-to-CR-IIP**, **mov-to-CR-IPSR**, **mov-to-CR-ISR**, **mov-to-CR-IFA**, **mov-to-CR-IHA**, **mov-to-CR-IIPA**
mov-to-PR	**mov-to-PR-allreg**, **mov-to-PR-rotreg**
mov-to-PR-allreg	mov_pr[Format in {I23}]
mov-to-PR-rotreg	mov_pr[Format in {I24}]
mov-to-PSR-l	mov_psr[Format in {M35}]
mov-to-PSR-um	mov_um[Format in {M35}]
mux	mux1, mux2
non-access	fc, lfetch, **probe-all**, tpa, tak
none	-
pack	pack2, pack4
padd	padd1, padd2, padd4
pavg	pavg1, pavg2
pavgsub	pavgsub1, pavgsub2
pcmp	pcmp1, pcmp2, pcmp4
pmax	pmax1, pmax2
pmin	pmin1, pmin2
pmpy	pmpy2
pmpyshr	pmpyshr2
pr-and-writers	**pr-gen-writers-int**[Field(ctype) in {and andcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-gen-writers-fp	fclass, fcmp
pr-gen-writers-int	cmp, cmp4, tbit, tf, tnat
pr-norm-writers-fp	**pr-gen-writers-fp**[Field(ctype)==]
pr-norm-writers-int	**pr-gen-writers-int**[Field(ctype)==]

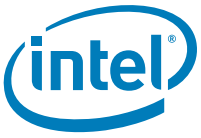
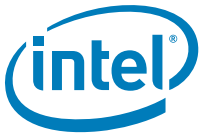
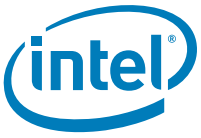


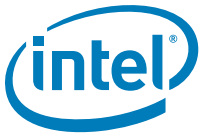
Table 5-5. Instruction Classes (Continued)

Class	Events/Instructions
pr-or-writers	**pr-gen-writers-int**[Field(ctype) in {or orcm}], **pr-gen-writers-int**[Field(ctype) in {or.andcm and.orcm}]
pr-readers-br	br.call, br.cond, brl.call, brl.cond, br.ret, br.wexit, br.wtop, break.b, hint.b, nop.b, **ReservedBQP**
pr-readers-nobr-nomovpr	add, addl, addp4, adds, and, andcm, break.f, break.i, break.m, break.x, chk.s, **chk-a**, cmp, cmp4, **cmpxchg**, clz, **czx**, dep, extr, **fp-arith**, **fp-non-arith**, fc, fchkf, fclrf, fcmp, **fetchadd**, fpcmp, fsetc, fwb, getf, hint.f, hint.i, hint.m, hint.x, **invala-all**, itc.i, itc.d, itr.i, itr.d, **ld**, **ldf**, **ldfp**, **lfetch-all**, mf, **mix**, **mov-from-AR-M**, **mov-from-AR-IM**, **mov-from-AR-I**, **mov-to-AR-M**, **mov-to-AR-I**, **mov-to-AR-IM**, **mov-to-BR**, **mov-from-BR**, **mov-to-CR**, **mov-from-CR**, **mov-to-IND**, **mov-from-IND**, **mov-ip**, **mov-to-PSR-l**, **mov-to-PSR-um**, **mov-from-PSR**, **mov-from-PSR-um**, movl, **mux**, nop.f, nop.i, nop.m, nop.x, or, **pack**, **padd**, **pavg**, **pavgsub**, **pcmp**, **pmax**, **pmin**, **pmpy**, **pmpyshr**, popcnt, **probe-all**, **psad**, **pshl**, **pshladd**, **pshr**, **pshradd**, **psub**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.d, ptr.i, **ReservedQP**, rsm, setf, shl, shladd, shladdp4, shr, shrp, srlz.i, srlz.d, ssm, **st**, **stf**, sub, sum, **sxt**, sync, tak, tbit, tf, thash, tnat, tpa, ttag, **unpack**, **xchg**, xma, xmpy, xor, **zxt**
pr-unc-writers-fp	**pr-gen-writers-fp**[Field(ctype)==unc][11], fprcpa[11], fprsqrta[11], frcpa[11], frsqrta[11]
pr-unc-writers-int	**pr-gen-writers-int**[Field(ctype)==unc][11]
pr-writers	**pr-writers-int**, **pr-writers-fp**
pr-writers-fp	**pr-norm-writers-fp**, **pr-unc-writers-fp**
pr-writers-int	**pr-norm-writers-int**, **pr-unc-writers-int**, **pr-and-writers**, **pr-or-writers**
predicatable-instructions	**mov-from-PR**, **mov-to-PR**, **pr-readers-br**, **pr-readers-nobr-nomovpr**
priv-ops	**mov-to-IND-priv**, bsw, itc.i, itc.d, itr.i, itr.d, **mov-to-CR**, **mov-from-CR**, **mov-to-PSR-l**, **mov-from-PSR**, **mov-from-IND-priv**, ptc.e, ptc.g, ptc.ga, ptc.l, ptr.i, ptr.d, rfi, rsm, ssm, tak, tpa, vmsw
probe-all	**probe-fault**, **probe-regular**
probe-fault	probe[Format in {M40}]
probe-regular	probe[Format in {M38 M39}]
psad	psad1
pshl	pshl2, pshl4
pshladd	pshladd2
pshr	pshr2, pshr4
pshradd	pshradd2
psub	psub1, psub2, psub4
ReservedBQP	-[15]
ReservedQP	-[16]
rse-readers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-from-AR-BSP**, **mov-from-AR-BSPSTORE**, **mov-to-AR-BSPSTORE**, **mov-from-AR-RNAT**, **mov-to-AR-RNAT**, rfi
rse-writers	alloc, br.call, br.ia, br.ret, brl.call, cover, flushrs, loadrs, **mov-to-AR-BSPSTORE**, rfi
st	st1, st2, st4, st8, st8.spill, st16
st-postinc	**stf**[Format in {M10}], **st**[Format in {M5}]
stf	stfs, stfd, stfe, stf8, stf.spill
sxt	sxt1, sxt2, sxt4
sys-mask-writers-partial	rsm, ssm
unpack	unpack1, unpack2, unpack4
unpredicatable-instructions	alloc, br.cloop, br.ctop, br.cexit, br.ia, brp, bsw, clrrrb, cover, epc, flushrs, loadrs, rfi, vmsw
user-mask-writers-partial	rum, sum
xchg	xchg1, xchg2, xchg4, xchg8
zxt	zxt1, zxt2, zxt4



§





V4-A CPUID CPU Identification

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 “ineI” (i in DL)
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)

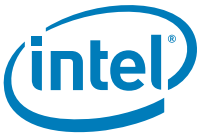


Table 2-4. Information Returned by CPUID Instruction

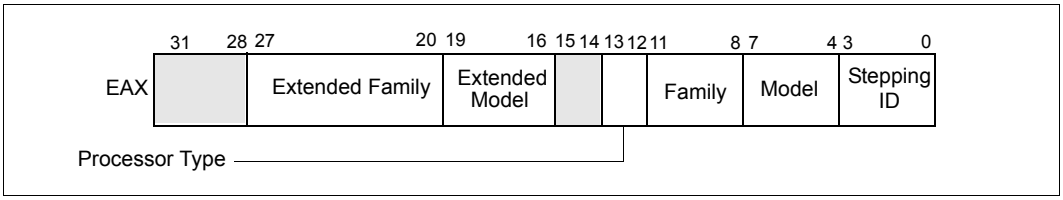
Initial EAX Value	Information Provided about the Processor	
2H	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information
		Extended Function CPUID Information
8000000H	EAX	Maximum Input Value for Extended Function CPUID Information
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
8000001H	EAX	Extended Processor Signature and Extended Feature Bits. (Currently reserved.) Reserved
	EBX	Reserved
	ECX	Reserved
	EDX	
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

Notes:

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.

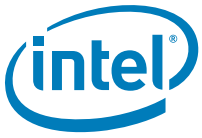
When the input value is 1, the processor returns version information in the EAX register (see Figure 2-4). The version information consists of an Intel architecture family identifier, a model identifier, a stepping ID, and a processor type.

Figure 2-4. 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.



CPUID—CPU Identification (Continued)

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.

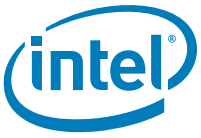


Table 2-5. Feature Flags Returned in EDX Register (Continued)

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.
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

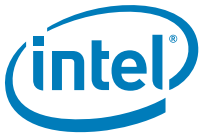


Table 2-5. Feature Flags Returned in EDX Register (Continued)

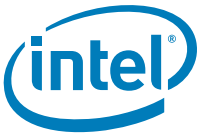
Bit	Mnemonic	Description
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued 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.

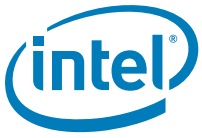
CPUID performs instruction serialization and a memory fence operation.



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.

§