PowerPC AS Operating Environment Architecture

Book III

Version 2.00

Feb. 24, 1999

Manager:
Paul Ledak/Burlington/IBM
Phone: 802-769-6960
Tie: 446-6960

Technical Content:

Ed Silha/Austin/IBM	Andy Wottreng/Rochester/IBM	Cathy May/Watson/IBM
Phone: 512-838-1848	Phone: 507-253-3597	Phone: 914-945-1054
Tie: 678-1848	Tie: 553-3597	Tie: 862-1054

IBM Confidential - Feb. 24, 1999

Softcopy Distribution:
VM: KISS64 disk Rochester: VM DOC disk BOOK4
DFS: /.../austin.ibm.com/fs/projects/utds/server_arch/Books
/.../rchland.ibm.com/fs/eng/docs/workbooks/cec_architecture/
Web: (Austin users) file:/.../austin.ibm.com/fs/projects/utds/server_arch/index.html
(Rochester users) file:/.../austin.ibm.com/fs/projects/system_arch/public_html/amazon.html
DFS Access Information: file:/.../austin.ibm.com/fs/projects/utds/index.html
Hardcopy distribution for Rochester: video conference center 025-1/A206

NOTES

- This is a controlled document.
- Verify version and completeness prior to use.
- See Preface for additional important information.

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC AS User Instruction Set Architecture and PowerPC AS Virtual Environment Architecture, that are provided by the PowerPC AS Operating Environment Architecture. It covers instructions and facilities not available to the application programmer, affecting storage control, interrupts, and timing facilities.

Other related documents define the PowerPC AS User Instruction Set Architecture, the PowerPC AS Virtual Environment Architecture, and PowerPC AS Implementation Features. Book I, *PowerPC AS User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book II, *PowerPC AS Virtual Environment Architecture* defines the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer. Book IV, *PowerPC AS Implementation Features* defines the implementation-dependent aspects of a particular implementation.

As used in this document, the term "PowerPC AS Architecture" refers to the instructions and facilities described in Books I, II, and III. The description of the instantiation of the PowerPC AS Architecture in a given implementation includes also the material in Book IV for that implementation.

Note: Two kinds of change bar are used. Both mark changes from Version 1.07.

| This marks a substantive change.

† This marks a non-substantive change.

User Responsibilities

- Do not make any unauthorized alterations to the document (user notes are permitted).
- Destroy the entire document when it is superseded, obsolete, or no longer needed.
- Distribute copies of the document or portions of the document only to IBM employees with a need to know.
- Verify the version prior to use. The version verification procedure is described later in this preface.
- Verify completeness prior to use. The last page is labeled "Last Page - End of Document". The end of the Table of Contents shows the last page number.
- Report any deviations from these procedures to the document owner.

Next Scheduled Review

There is no scheduled review.

Approval Process

The process used by the Processor Architecture Review Board (PARB) to approve or reject changes proposed for this architecture is documented at the following DFS directory: /.../austin.ibm.com/fs/projects/utds/server_arch/process

Approvals

This version has been approved by the PARB.

Version Verification for those with access to KISS64

- Link to the KISS64 disk in Yorktown or a shadow of this disk in Austin or Endicott. In Yorktown, linking to KISS64 can be done by executing the command "GIME KISS64". In Rochester, the shadow disk is VMCTOOLS 801.
- Browse the file "AMAZON VERSION" by typing "br" next to the file name.
- Verify that your version matches this file.

Version Verification for those without access to KISS64

- Verify that the version date matches the date on the Books on the Web site at:

  http://w3.austin.ibm.com/.../austin.ibm.com/fs/projects/utds/server_arch/

Table of Contents

Figures

Changes as of 1999/02/24 Version 2.00

change	reason	page
Add Data Stream Touch variant of ***dcbt*** instruction.	RFC02000 and Correspondence of 3 Nov. '98.	18, 41-42, 56-57, 81, 94
Make RA 62-bit, and eliminate E=R, E=DS, and T=1.	RFC02001. In addition, in Section 4.5.2 the first sentence of the paragraph that describes how the HTABSIZE field is used was corrected by inserting a parenthetical phrase.	2-4, 11-12, 15, 21, 24-27, 29-30, 32, 34-36, 40, 45-47, 60-68, 70, 72-74, 80, 82, 84, 84+1, 89
Add lightweight ***sync***; drop ***vsync***; make other changes regarding shared storage.	RFC02002. In addition the following changes were made. ■ Because it is mentioned in item 2 of the section, ***isync*** was added to the list of examples at the end of the first paragraph of Section 1.6.1. ■ For clarity, “in the PowerPC AS Operating Environment Architecture” was inserted in the new paragraph for Appendix G.	3, 24, 26, 39, 41, 46, 56-58, 61, 64-66, 69, 93-94, 102+1, 117
Minimize ACCR, make Data Address Compare and Data Address Breakpoint more similar.	RFC02003. In addition, in the Engineering Note in Section 10.2, for consistency with the preceding paragraphs “target storage location” was changed to “storage operand”.	37-38, 47, 64-66, 84
Relax rules regarding setting C and TS bits; drop Floating-Point Assist interrupt.	RFC02004.	1, 3, 26, 35, 40-42, 47, 62, 69, 71-74

change	reason	page
Add software-managed SLB and make various other MMU changes.	RFC02005. In addition the following changes were made. ■ For consistency with the new Programming Note for ***mtmsrd***, in other such Notes "please refer to" was changed to "see". These are regarded as minor editorial changes, and are neither marked with change bars nor reflected in the page list in this entry. ■ "interrupt" was changed to "exception" in a few places, and eliminated in a few others. ■ The RFC's wording for Sections 4.2.1 and 5.2.1 was modified in several respects, including the following. — To avoid requiring the list of storage exceptions to be complete, the list was stated to give examples. — Because RFC02011 permits certain instructions to cause either a Data Storage interrupt or an Alignment interrupt if they attempt to access Write Through Required or Caching Inhibited storage, Alignment interrupt was added to the list of interrupts that a storage exception can cause. ■ The second bullet of the Programming Note in Section 4.4.1.1 was clarified. ■ The RTL for ***slbie*** and ***slbia*** was revised to avoid double subscripting. ■ Because the phrase seemed more confusing than helpful, "in real address space" was not added at the end of the paragraph preceding Section 6.2.1. ■ For consistency with the description of DAR setting for Data Storage interrupts, the same parenthesized explanation of "first" was added to the description of DAR setting for Data Segment interrupts. ■ Chapter 9 Note 6: — The RFC's wording for the last sentence of the second paragraph was modifed for consistency with changes made to the preceding sentence by the RFC02000 Correspondence of 3 Nov. '98. — Item 1 of the Programming Note was clarified slightly. ■ In Section 10.3, for consistency with wording in RFC02007 "real mode" was changed to "real addressing mode" except in the section title. Also the last Engineering Note was deleted because it applied only to the real mode I bit portion of the section, which RFC02007 moves to the architecture proper. ■ ***mtmsr*** definition: — The definition was placed in Chapter 10, instead of in Chapter 11, because AIX software does not intend to phase out the use of ***mtmsr***. (This also affects Appendix C.) — The paragraph referring to Chapter 9 was made a separate Note, for consistency with usage elsewhere, and in the existing Programming Note "additional" was changed to "analogous".	1-4, 7-9, 19-21, 24-27, 30-37, 41-43, 46-47, 49-58, 61-62, 64-66, 71, 73-74, 79-82, 84+1, 86-92, 94, 99, 106, 117

change	reason	page
Define a common Logical Partitioning architecture for AS/400 and RS/6000.	RFC02007 and Correspondence of 13 Jan. '99. In addition the following changes were made. ■ Because for AS/400 "supervisor" is not equivalent to "privileged", in the relevant places "supervisor" was changed to "supervisor (privileged)". ■ The name of the "A4R6" bit was changed to the less cryptic "LPES" ("Logical Partitioning Environment Selector"). ■ The last sentence of the Programming Note in Section 4.2.5.1 was reworded slightly, for consistency with Section 4.2.6. ■ The end of the last sentence of the first paragraph of Section 4.5.1 was revised to cite Section 4.2.5.2 instead of Section 4.7. ■ The MSR_{PR} and MSR_{US} bullets in Section 4.9.1 were reworded slightly for consistency with Section 4.9.2. ■ In Sections 4.9.1 and 4.9.2, material was added near the beginning of the second "rule" to avoid conflict between that "rule" and the first "rule". ■ In Figure 30, "(implementation-dependent)" was added to the definitions of m, s, and v, for consistency with the subsequent Programming Note and with the ***rfcsv*** definition. ■ In Appendix G the definition of "TA" was reworded slightly for consistency with changes made in the "Key" definitions by this RFC and RFC02005.	2-6, 8-13, 16-21, 25, 27-29, 31, 34-35, 38, 42-43, 46-47, 55-57, 62-66, 69-70, 75-76, 79-80, 83, 84, 84+1, 86-87, 91-92, 117
Adopt PowerPC model of DSI/ISI exception ordering.	RFC02008.	65-66, 72-73, 84
Eliminate Firm Consistency.	RFC02009. In addition, the Architecture Note describing the old use of MSR_{FC} was reworded slightly for consistency with other such Notes.	10, 26, 62, 80
Describe ***dcbz*** as architecture, drop ***dcba*** and ***dcbi***.	RFC02010.	41, 49, 63-64, 89, 99, 115, 117

change	reason	page
Make changes regarding Alignment interrupts and instruction restart.	RFC02011 and Correspondence of 18 Dec. '98. In addition the following changes were made. ■ Section 7.5.3: — The current fifth bullet, as modified by RFC02001, was merged into the current second bullet, as modified by RFC02002. (The parenthesized phrase in the current two bullets is no longer true: RFC02011 permits these cases to cause an Alignment interrupt. RFC02002 removes the phrase for the current second bullet.) — For clarity, in the Engineering Note at the end of the section "may not be supported" was changed to "need not be supported". ■ In Sections 7.5.8 and 7.7.1 a few very minor changes (for clarity or consistency) were made in the current wording. ■ Section 10.2: — Because ***dcbz*** does not appear in the bulleted list that precedes the Programming Note, ***dcbz*** was omitted from the second sentence of the paragraph before the list. (Both the paragraph and the list are supplied by the Correspondence.) — The last sentence of the new paragraph proposed in the Correspondence for the Programming Note added by RFC02003 was omitted, because it is an obvious consequence of the paragraph before the Note. (Also the sentence was incorrect; the cases covered by the cited list are those in which the operand is *not* altered. And the citation of "the preceding paragraph" mistakenly referred to the first paragraph of the Note.)	25, 27, 38, 65, 67-68, 72-74, 84, 101

change	reason	page
Reduce complexity associated with interruption vectors.	RFC02013 and Correspondence of 21 Dec. '98. In addition the following changes were made. ■ In Section 4.2.6, explanation of the use of the first 256 bytes was added to the first bullet. Also, minor wording changes were made for internal consistency and for consistency with Section 4.2.5.1. ■ The section citation in the description of ***scv*** was changed from Section 7.4 to Section 7.5, for correctness and to match ***sc***. ■ Section 7.4: — The material excepting Machine Check and System Call Vectored interrupts was reworded somewhat and moved. — For consistency with the RFC's approach for this section, mention of Machine Check was deleted from items 3 and 5, and item 4 was reworded somewhat. — The sentence about interrupts and reservations was not moved. (The sentence is architecture, and Book II expects Book III to treat it as such.) ■ Section 7.5 Figure 31: — An ending delimiter was added to the 00Exx range. — The current ending delimiter (03FFF) was retained, modified appropriately. — The notation in Note 1 was changed slightly. ■ In the new Programming Note in Section 7.5.15, for clarity the order of the two paragraphs was reversed and the first sentence of the (now) second paragraph was reworded somewhat. Also, a new second sentence was added to that paragraph.	9-12, 29, 47, 61-71, 80, 113

change	reason	page
Make the following changes. ■ Make Little-Endian optional. ■ Make support of Write Through Required optional. ■ Delete tagged pointer support. ■ Delete Programming Notes and Architecture Notes regarding deviations by earlier PowerPC AS processors. ■ Delete old Performance Monitor. ■ Clarify tag bit description.	RFC02014 and Correspondence of 10 Feb. '99. In addition the following changes were made. ■ Because “tags_active” RTL notation is used only in the ***lmd*** section, which this RFC deletes, its definition in Section 1.3.1 was deleted. ■ The definitions of MSR_{LE} and MSR_{ILE} were reworded slightly, for consistency with the RFC's changes in Book I. ■ The RFC's change for Chapter 9 was not made. RFC02007 changes the definition of ***mtmsr***[***d***] such that using ***mtmsr***[***d***] to set PR to 1 has the side effect of setting IR to 1. Thus ***mtmsr***[***d***] (PR) can still cause an implicit branch in real address space. ■ In Section A.1, MMCRA and MMCR1 were also added to the table (not SIAR or SDAR, because they are less likely to become part of the base PM architecture). Also the PM SPRs were inserted in order of SPR number, and the Programming Note was reworded slightly. ■ In Section E.2, IMRU and IMRL were deleted from the figure because they are not mentioned in the rest of the section. ■ “Amazon” was changed to “PowerPC AS” throughout the Book, without change bars.	2, 7-10, 12-13, 20, 23-24, 38-39, 45, 58, 62, 68, 84+1, 86, 89, 94, 107, 117

Note: Change list entries for versions of the Books earlier than the current version may have been simplified in order to avoid references to deleted material.

For Version 1.07 and earlier versions, PowerPC AS Requests for Change (RFCs) are explicitly identified as such; other RFCs that are not explicitly identified are PowerPC changes that are adopted for PowerPC AS.

Changes as of 1998/04/30 Version 1.07

change	reason	page
Identify $MMCR0_{1:4}$ as a basic feature. Remove $MMCR0_6$ as a basic feature.	Amazon RFC 371	108
Remove Architecture Note that said the base address would be 0 when E=R was dropped.	Amazon RFC 370	9, old LPAR sect.
Remove $VSID_{0:24}$ from the RS register for ***mtsr*** and ***mtsrin***.	Amazon RFC 369	90
Modify *tags inactive* mode page protection to allow PP encodes of 4 and 5 to behave the same as encodes 0 and 1, respectively.	Amazon RFC 367	43
Miscellaneous changes: ■ In Section 2.2.3 drop the explanation for "full function" since it duplicates the explanation in the Architecture Note. ■ Add the missing note 2 cited by the entries in the ***mfspr*** table (Figure 12) for perf_mon and for SPRG3. ■ Adopt PowerPC note numbering convention to use the same note number for each register in Figure 12 as in Figure 11. ■ Remove IPR from extended mnemonics for ***mfspr*** and ***mtspr*** ■ Add the word "Lower" to the SPR "Time Base" for the ***mttbl*** extended mnemonic.	Obvious errors or minor changes for consistency with PowerPC	7, 20, 94
Adopt changes in Chapter 4 corresponding to PowerPC RFC00248 (large pages) and RFC00249 (BAT).	Amazon RFC 372	24-25, 26-27, old addr. xlation mech. sect., old SegTab sects., 34, 36-40, 42

change	reason	page
Start phasing Block Address Translation (BAT) out of the architecture.	RFC00249 and Correspondence of 17 Dec. '97, as amended at 16 Dec. '97 PAWG conference call. The changes are modified by Amazon RFC 372. In addition the following changes were made. ■ For consistency with wording elsewhere, “data storage access” was changed to “data access” in the first paragraph of Section 7.5.3. ■ An item in the new Section 10.3 was reworded somewhat. ■ The paragraph after the table showing the allowed BL values was deleted, because it is redundant with the last paragraph of its section. ■ A few minor changes were made to wording proposed in the RFC.	19-20, 46, old Ch. 5 sects., 47-47, 43, 64-67, 72, 80-80, 84, 86, 89, old dir. store sect., old BAT sect., 93-94

change	reason	page
Provide support for 4 KB pages, one larger page size, and No-execute pages.	RFC00248 as modified by Amazon RFC 372 and rewritten by Correspondence of 9 Dec. '97, and Correspondence of 17 Dec. '97 and 10 March '98. In addition the following changes were made. ■ To avoid confusion with large pages, two instances of the word "page" were clarified to refer to 4 KB explicitly. For consistency and improved accuracy, "power of 2" was changed to "multiple of 4 KB". ■ For accuracy, in the last sentence of the first paragraph of Section 7.7 "that spanned a page boundary" was changed to "for which the storage operand crosses a virtual page boundary", and the other two instances of "page" were changed to "virtual page". ■ In the old "Large Page" section: — p was used instead of n for $\log_2$ of the page size; p seems more suggestive, and permits retaining "PTEGn" in the old "Large Page" section's virtual-to-real figure. — The introduction to the description of the PTE was reworded for consistency with the description of the STE. The second sentence was omitted because it is redundant with the PS bit definition. — The first sentence after the RTL in the definition of ***tlbie*** was reworded and moved to the beginning of the sub-section, for consistency with the intro-duction to the description of the STE. — The name of the PS field used by the new form of ***tlbie*** was changed to S, because a 1-letter name fits better in the instruction format, and use of PS here might cause confusion with the other use of PS (field in PTE). (S was chosen instead of P to avoid confusion with uses of p to represent the $\log_2$ of the page size.) ■ "VPS" was changed to "pg_ind" in the RTL for ***tlbie*** (old form and new). ■ A few minor changes were made to wording proposed in the RFC.	old Ch. 5 sects., 55, 64-66, 72, old Large Page sect.
Make several corrections related to changing PTEs.	RFC00247. The second proposed UP simplifi-cation, for the new "General Case" sequence of modifying a PTE, was omitted because it is incor-rect. Because a TLB entry can be loaded at any time (e.g., to prefetch the instructions following the final ***sync***), absence of the ***eieio*** that sepa-rates the second PTE update from the third would permit an inconsistent TLB entry to be loaded.	56-58
Permit *Load and Reserve* and *Store Conditional* to Caching Inhibited storage to cause a Data Storage interrupt.	RFC00245.	64ff

Changes as of 1998/03/27 Version 1.06

change	reason	page
Permit SPRG3 to be read in problem state.	RFC00246. In addition the following change was made. ■ The new paragraph proposed by the RFC00243 Correspondence of 2 Nov. '97 to be added near the end of the ***mtspr*** and ***mfspr*** instruction descriptions was added as part of RFC00246, because RFC00246's new note 3 for Figure 12 assumes that the paragraph exists (and absence of the paragraph is an oversight in Version 1.09).	16, 19-20
Permit *Load and Reserve* and *Store Conditional* to Caching Inhibited storage to cause a Data Storage interrupt.	RFC00245.	64ff
Remove the Architecture Note that says bits 11:15 of ***slbia*** must be zero.	Amazon RFC 363	52
Show $SDR1_{0:11}$ must be zero.	Amazon RFC 362	35
Clarify high order EA bits are ignored in real addressing mode.	Amazon RFC 361	27
Make it implementation-dependent whether a 65-bit or 80-bit VA is supported.	Amazon RFC 358	24, 45
Adopt PowerPC change to identify real storage locations having defined uses.	Amazon RFC 357	29, 47
Remove references to *tags inactive* mode direct-store from PowerPC AS.	Amazon RFC 356	47, old dir. store sect.
Add quadword atomicity requirement to ***dcbi*** instruction description	Amazon RFC 351	old ***dcbi*** def.
Remove IPR from the architecture.	Amazon RFC 350	old MSR_{IP} definition, 10, 19, 20, 80
Allow instruction fetch storage protection for tags active mode to be implementation-dependent.	Amazon RFC 349	43
Correct the Page Table Hash description to refer to the correct VPN bits.	Amazon RFC 348	36
Added Process Local Storage addressing and additional storage protection states	Amazon RFC 347	24-25, 30-33, 42, 64, old DSI/ISI excep. ord. section
Add requirement for synchronizing TBs in all processors.	Amazon RFC 343	76
Removed C2 Security Mode	Amazon RFC 342	8, 10, 24, 26
Ease requirement to block covert channel; add the requirement to synchronize all processors TBs to almost the same value.	Amazon RFC 341	76

change	reason	page
Delete Appendix F. Implementation-Specific SPRs	PowerPC RFC00167 and Amazon RFC 340	old implem'n.-specific SPRs appendix
Allow an implementation-dependent DSI for ***lq*** and ***stq*** to Write Through Required or Caching Inhibited storage or direct-store segments	Amazon RFCs 304 and 339	old dir. store sect., 64
Highlight that for implementations like Northstar that provide a mechanism to define a portion of real address space as non-Guarded, software should not map a virtual page with an I bit of 1 to such a real address.	Amazon RFC 336	27, old Ch. 5 sect.
Require an ***isync*** after a *Load* from a direct-store segment	Amazon RFC 333	old dir. store sect.
Define DAR to be undefined after a direct-store access.	Amazon RFC 332	15, old direct-store sections
Add Programming Note in 7.5.3 explaining how consistent behavior can be obtained for Problem State E=R accesses.	Amazon RFC 331	64
Define the exception priorities for T=1 instruction/data accesses and for Guarded instruction fetches	Amazon RFC 329	old DSI/ISI excep. ord. section
Add LPAR support: ■ Add mode for translating E=R addresses ■ Add mode for preventing instructions from changing MSR_{IR} or MSR_{DR} from 1 to 0. ■ Remove the option for hardware to modify SRR0 & SRR1 when MSR_{IR} = 1 or MSR_{DR} = 1.	Amazon RFC 328	7, 9, 24, old Stg. Seg. sect., 47, 61, old LPAR section
Delete references to an implied N bit for Direct-Store segments.	ISI occurs for instruction fetch of Direct-Store segment regardless of implied N bit.	old Stg. Seg. sect.
If an EAO exception occurs in *tags active* mode simultaneously with an Alignment interrupt, the DAR can be loaded based on a 24 or 64-bit add if the instruction is ***lq***, ***stq*** or in Little-Endian mode, ***lmd***, ***stmd lmw***, ***stmw*** or Move Assist instructions.	Amazon RFC 326	68
Restrict E=DS accesses to elementary loads/stores with doubleword operands that are doubleword-aligned.	Amazon RFC 325	old dir. store sects., 72
Re-define TAGR as 32-bit register and change ***mtspr*** and ***mfspr*** to show only 32 or 64-bit register moves.	Amazon RFC 324	19, 20, old TAGR sect.
Adopt PowerPC Performance Monitor facility as optional architecture and phase out the previous Amazon Performance Monitor facility.	Amazon RFC 322	79, old Perf. Mon. sect., 105ff
Clarify that the Tag Set bit is set by out-of-order accesses only if $XER_{:43}$= 1	Amazon RFC 321	40, 41

change	reason	page
Clarify that E=DS instruction fetches with PR=0 result in an Instruction Storage interrupt. Clarify how cache management instructions with an E=DS operand are handled.	Amazon RFC 318	25, old dir. store sect., old ***dcbi*** def.
Clarify that it is implementation-dependent which SRR1 or DSISR bits are set for Problem State E=DS accesses.	Amazon RFC 317	64, 66
State the exception priority for Data Address Breakpoint exceptions	Amazon RFC 314	old DSI/ISI excep. ord. section, 84
Remove statements that STE_T bit must be 0 in *tags active* mode	Amazon RFC 311	old SegTab sect.
Specify context synchronizing requirements for moves to ACCR.	Amazon RFC 310	80
Make TAGR optional.	Amazon RFC 309	17, 20, 72, old ***lmd*** def.
State that ***sync*** does not synchronize Direct-Store Errors.	Amazon RFC 308	4, old direct-store sections, old TA/ direct-store synch. Note
Clarify that for Big-Endian, *tags active* mode, an EAO type of Data Storage interrupt has priority over Alignment interrupt	Amazon RFC 307	67
Remove note relating to synchronization requirements for changing Endian mode by ***scv*** since this ***scv*** does not change MSR_{LE}.	Amazon RFC 306	80
State that if the length is zero, ***stsdx*** is allowed to set the Change bit and ***stsdx*** and ***lsdx*** can set the Reference bit.	Amazon RFC 306	40, 41
Clarify that E=R and E=DS addresses with MSR_{PR}= 0 are not “translated”	Amazon RFC 306	24
Remove references to MXU	Amazon RFC 302	101, 102
List the key bit as an item that is passed to the storage controller for direct-store addresses.	Amazon RFC 301	old dir. store sect.
State that direct-store errors are another exception to the sequential execution model.	Amazon RFC 300	2
Add Programming Notes for Cobra 4 deviations ■ ***sync*** required in instruction stream after ***rfi*** or ***rfscv.*** ■ no modification of interrupt vector locations after IPL	Amazon RFC 298	12, 13, old dir. store sect., 62
Remove ***slbiex*** and ***tlbiex***	Amazon RFC 294	37, 49, 99
Remove references to MUSKIE Pass 1 and COBRA 0 since these processor versions are no longer used.	Amazon RFC 294	12-13
Clarify the definition of “protection boundary”.	Amazon RFC 291	42

change	reason	page
Correct the description of coherence when the W bit differs among processors.	Amazon RFC 290	old "Mis-matched WIMG Bits" section
Clarify or restrict several aspects of out-of-order operations	Amazon RFC 289	25, 27
Tighten the rules for setting the Change and Tag Set bits out-of-order.	Amazon RFC 288	40-41
Modify the description of Change bit setting to avoid mentioning the TLB.	Amazon RFC 287	40
Clarify that Reference and Change bits are set for virtual pages	Amazon RFC 286	40
Adopt wording changes similar to those in PowerPC version 1.07 and Morgan Kaufmann	Amazon RFC 285	24ff
Add minor changes for 32-bit Bridge facilities to Chapter 4.	Amazon RFC 284	25, 25, 30, old STE fig.
Adapt PowerPC Bridge Facilities wording to PowerPC AS	Amazon RFC 236	89ff
Delete sections on instruction formats and fields.	RFC00231.	2
Redefine ***sync*** to make it a memory barrier, redefine ***tlbsync*** to make it order ***tlbie*** effects only with respect to the memory barrier created by a subsequent ***sync***, and make ***eieio*** order ***tlbie*** and ***tlbsync*** as a third set.	Amazon RFC 360, RFC00233 and Correspondence of 7 Nov. '96. In addition the following changes were made, for consistency with changes made by the RFC. ■ Minor font and wording changes were made in the names of the instructions in the first sentence of Section 5.1. Also, "and" was changed to "or" for consistency with Book I. ■ In the paragraph before the old Chapter 5 R/C bit figure, "treated as loads with respect to address translation" was changed to "treated as *Loads*", and similarly for stores. ■ In item 2 of Section 7.3.1, "generated by" was changed to "associated with" and "all other" was changed to "other".	3, 24, 26, 27, 40ff, 45, old Ch. 5 sect., 47, 47ff, 55-56, 57ff, 60-61, 79ff, old ***dcbi*** def.
Require SRR0 and SRR1 to be preserved when addresses are translated by BAT.	RFC00235 and Correspondence of 5 Nov. '96, as amended at Oct. PAWG meeting. In addition the following changes were made, for consistency with wording elsewhere in the Books. ■ In the first paragraph of Section 5.2.5, "translation modes" was changed to "translation mechanisms". ■ In the last Programming Note in the old Chapter 5 "Segment Table Search" section, "covered by BAT translation" was changed to "translated by BAT".	7-7, 46, 47, old Ch. 5 sect., 61
Clarify "last to be assigned a meaning" for resources used by Amazon (MSR bits 1 and 33, bit 54 of doubleword 1 of PTE).	Consistency with RFC00225 as amended at Oct. PAWG meeting.	8
clarify various aspects of Trace.	RFC00223 as rewritten by Correspondence of 19 Nov. '96. In addition, for brevity and readability, in the new appendix "$MSR_{SE\ BE}$" was used instead of "MSR_{SE} || MSR_{BE}".	9, 71, 115ff

change	reason	page
Add optional Performance Monitor facility.	RFC00222 and Correspondence of 6 Nov. '96 and 14 Nov. '96, as amended at 21 Nov. PAWG conference call. In addition the following changes were made. ■ In Figure 12, the text of the 14 Nov. Correspondence's Note 6 was appended to its Note 4, with no citation of Note 6 at the top of the “SPR” column, because Note 6 applies only to the Performance Monitor registers. ■ The second sentence of Section 7.5.14 was reworded for consistency with the description of the PMM bit. ■ The word “Facility” was omitted from the title of the new appendix, for consistency with similar sections. ■ In the second bullet describing the Performance Monitor hierarchy (Appendix E), “the MMCRs” was changed to “an MMCR” for consistency with the next bullet. ■ In Section E.1, for reasons of page layout and consistency the Notes were placed at the end of the section. ■ In the last paragraph of the Programming Note for $MMCR0_{TBSEL}$, “system service routine” was changed to “system service program” for consistency with the Books' terminology. ■ Clarify that the Trace facility need not include setting SIAR and SDAR (see changes made by RFC00223).	10-10, 19-20, 62, 71, 80, 105ff
Reserve SPRs for implementation-specific uses.	RFC00167 as rewritten by Correspondence of 30 May '96, as amended at Oct. PAWG meeting. In addition the following changes were made. ■ “(SPRs)” was inserted in the first new sentence in Section 3.2, because the abbreviation is used subsequently. ■ Because RFC00222 causes the Performance Monitor SPR numbers to be specified in the architecture proper, those SPR numbers were removed from the new Engineering Note in Section 3.4.1 and “implementation specific uses” was moved from the list of SPR numbers to the first sentence of that Note. SPR numbers 944-945 and 952-955 were removed from the new Architecture Note because they are in the Performance Monitor range. ■ For reasons of page layout, the new Notes in Section 3.4.1 were placed after the paragraph covering extended mnemonics instead of before it.	15, 18ff, old implem'n.-specific SPRs appendix
Add PIR, add a requirement for processors to provide a way to clean the entire data cache, and make a number of miscellaneous changes.	RFC00238 as amended at Oct. PAWG meeting. In addition, the first sentence of the new paragraph at the end of Section 6.1.1 was reworded slightly, for reasons of consistency with RFC00242 and of grammar, and the citation of Book IV in the next sentence was changed to match the Books' usual style.	17, 20, old BAT section, 49, 51, 55, 65, 66, 68, old dir. store sect., 94

change	reason	page
Reorganize WIMG description, and remove redundant descriptions of *Cache Management* instructions.	Amazon RFC 359, RFC00242 and Correspondence of 14 Nov. '96. In addition the following changes were made. ■ For consistency with the changes for item 4 of Section 5.2, "devices" was changed to "I/O devices" in item 1. ■ For consistency with the changes regarding "data storage" (Correspondence of 14 Nov. '96) and with the title of Section 5.2.2, "Storage" was deleted from the title of Section 5.2.3. ■ For consistency with other changes made by the RFC and for readability, "Caching Inhibited Guarded storage" was changed to "storage that is both Caching Inhibited and Guarded" near the beginning of the third paragraph of the old Chapter 5 "Guarded Storage" section. ■ Because the AIM Books should not attempt to dictate section titles for alternative Books, the reference to Book II was changed from generic to specific in the first paragraph of Section 5.5 and in the second paragraph of the last Engineering Note of that section.	24, 25-27, 27, old SegTab sects., 34, 37, 46, 46, old Ch. 5 sects., old dir. store sect., old ***dcbi*** def.
Revise the rules for fetching instructions when $MSR_{IR}=0$.	RFC00239. In addition, for reasons of grammar "is met" was changed to "are met" in the first sentence under "Instruction Fetch" in the old Chapter 5 "Out-of-Order Accesses to Guarded Storage" section.	old Ch. 5 sect.
Permit aliasing of ***dcbi*** as ***dcbf***, and start phasing ***dcbi*** out of the architecture.	RFC00234 and Correspondence of 23 Sept. '96, as amended at Oct. PAWG meeting. In addition the following changes were made in the description of ***dcbi***. ■ In the second and third paragraphs, the first sentence uses wording from RFC00242 rather than that in RFC00234, and the second sentence was reworded for consistency with RFC00242 (including that RFC's effects on ***dcbf***) and to clarify the temporal ordering. ■ The fifth paragraph combines wording from RFC00233 with that in RFC00234, with some modifications for consistency with RFC00233 and RFC00242. ■ The third bullet was reworded for consistency with RFC00233. ■ Adding a clause at the end of the Architecture Note as agreed at the meeting (similar to wording in the Architecture Note that RFC00167 adds to Section 3.4.1) required changing "architecture" to "PowerPC AS Architecture" in the preceding clause.	old Ch. 5 R/C fig., 49, 64ff, 89, old ***dcbi*** def., 99
Clarify what causes $FPSCR_{FEX}$ to be set to 1.	RFC00230.	68
Make various clarifications regarding instruction restart.	RFC00237 and Correspondence of 19 Sept. '96, as amended at Oct. PAWG meeting.	72
Clarify that reserved MSR bits need not be saved or restored. Delete obsolete Engineering Note about reserved bits.	RFC00213 and Correspondence of 23 March '96, as amended at March PAWG meeting.	2, 7

change	reason	page
Start to phase direct-store out of the architecture.	RFC00220. In addition the following changes were made. ■ The old Chapter 5 and direct-store STE figures were made single-column. ■ In the description of DSISR bit 5 for Section 7.5.3 the word "instruction" was added in the Write Through Required clause, for consistency with wording elsewhere in the section.	3-3, 45-46, 47, 47, old Ch. 5 sects., old BAT section, 47-43, 49, old STE update sects., 63-68, 72, 83ff, 90ff, old dir. store sect., 101, old UP Appendix
Clarify definition of execution synchronization.	RFC00221 as amended at March PAWG meeting.	4, 87
Clarify use of storage now shown as allocated to interrupt vectors.	RFC00214. In addition, boldface was used for the bullets on p. 47 for consistency with usage elsewhere in the Books.	9, 47, old Ch. 5 sect., 62
Make MSR_{IP}= 1 always vector interrupts to base real address 0xFFF0_0000.	RFC00216. In addition, "0xC00" was changed to "0x00C00" in the ***sc*** description on p. 11 for consistency with the System Call interrupt description.	9, 11, 63-71
Revise description of PVR.	RFC00211 and Correspondence of 11 March '96, as amended at March PAWG meeting. The changes proposed for the "Processor Version Numbers" appendix were not made because RFC00212 deletes that appendix.	17
Permit alternative virtual address size of 64 bits.	RFC00229.	45, old Ch. 5 sects., old "Bridge" fig., old ***mtsrd***[***in***] sect.
Add new *Cache Management* instruction *Data Cache Block Allocate* (***dcba***).	RFC00228 and Correspondence of 10 May '96. In addition, in the first sentence of the second Programming Note in Section 7.5.2 the phrase "execution of ***dcbz*** or ***dcba***" was changed to "executing a ***dcbz*** or ***dcba*** instruction", for consistency with wording elsewhere in the Note.	46, old Ch. 5 R/C fig., 49, 63, 65, 84, old dir. store sect.
Revise the description of table update synchronization requirements.	RFC00226 as amended at March PAWG meeting. In addition the following changes were made. ■ The wording of the new clause about WIMG for storage tables was changed slightly (consistency with related sections). ■ The content and layout of several comments in the operation sequences in Section 6.2 were changed slightly (page layout). ■ In the last paragraph under "Modifying the Virtual Address" on p. 58, "is replaced" was changed to "would be replaced" (grammar).	46, old Ch. 5 sects., old R/C synch. section, 55-56, 57ff

change	reason	page
Describe why various facilities and instructions are optional.	RFC00218 as amended at March PAWG meeting, and Correspondence of 10 April '96. The "Optional" appendices were made chapters, as agreed at the March PAWG meeting; this necessitated changing "appendix" to "chapter" in several places. In addition the following changes were made. ■ The wording for the introduction to the new chapter (optionality category 2) was made singular and "instructions" was omitted, because the chapter contains just one facility (direct-store). ■ The new chapter was placed after the "Bridge" chapter, instead of before it as proposed in the RFC, because (a) the new chapter refers to the "Bridge" facility, and (b) the "Bridge" facility is likely to be elevated to optionality category 3.	old Ch. 5 sects., 49, 71, 83, 89, 91ff, 89
Amplify Engineering Note about ignoring M bit for instruction fetch.	RFC00200 as amended at March PAWG meeting.	old Ch. 5 sect.
Delete "Processor Version Numbers" appendix.	RFC00212 as amended at March PAWG meeting.	old PVN appendix
Incorporate minor changes from the Morgan Kaufmann book. All such changes that seem desirable have now been made. Very minor changes (e.g., fixing grammatical errors) are not marked with change bars.	Agreed in discussion of RFC00173 at Nov. '94 PAWG meeting.	various
Correct the "Approval Process" description.	Correspondence of 27 Oct. '94.	iii
Clarify or restrict several aspects of out-of-order operations.	RFC00187 as amended at Nov. PAWG meeting. The change proposed for the old Chapter 5 R/C figure was not made because it is overridden by RFC00184.	2, 46-46
Make 64-bit MMU functions an extension of 32-bit MMU functions.	RFC00178 as rewritten by Correspondence of 24 Oct. '94. The change proposed for the old Chapter 5 R/C figure was not made because it is superseded by RFC00184. A few minor formatting changes were made in 11.1, to make various schematic descriptions fit in a single column. In Chapter 1 the "[***d***]" suffix was added to a few instances of ***rfi*** and ***mtmsr*** that were missed by RFC00178.	2-7, old ***rfi*** def., 87, old Ch. 5 sects., old STE update sect., 61, 73, 79ff, 83, 71, 89ff, 99,
Correct several minor errors.	Error Notice of 27 Oct. '94, Book III items 1-5, 7-11, and 13-14. (Items 6 and 12 are done as part of RFC 173.) Also, "Floating-Point Enabled Exception interrupt" has been changed to "Floating-Point Enabled Exception type Program interrupt" in a few places in Sections 7.3.2 and 7.5.9 that items 7 and 11 missed.	7, 9, old Ch. 5 sect., 43, old ***dcbi*** def., 60, 68-70, 73, 97, 101
Clarify meaning of "reserved full/partial function".	RFC00188. ***rfid***, added by RFC00178, has been included in the new Architecture Note.	8-10
Change the definition of MSR_{46} to use an Architecture Note, and state that MSR_{61} is "implementation-specific".	RFC00189.	8, 9
Relax rules for hardware's handling of reserved bits in registers.	RFC00195.	9

change	reason	page
Clarify instruction fetching and instruction cache paradoxes.	RFC00202.	46
Delete item 3 (a cautionary remark about cache synonyms) from three Programming Notes.	RFC00173.	old Ch. 5 sects.
Specify that IBATs contain W and G bits and that software must not write 1s to them.	RFC00191 as amended at Nov. PAWG meeting.	old BAT section, 47
Change “only have meaning” to “have meaning only”.	RFC00208.	47
Correct the description of coherence when the W bit differs among processors.	RFC00190.	old “Mismatched WIMG Bits” section
Clarify that Reference and Change bits are set for virtual pages.	RFC00179.	47
Revise description of Change bit setting to avoid depending on the TLB.	RFC00183.	47, old R/C synch. section
Tighten the rules for setting the Change bit out-of-order.	RFC00184 as amended at Nov. PAWG meeting.	47, old R/C synch. section
Change “load or store” to “load, store, or instruction fetch”.	RFC00173.	old R/C synch. section
Clarify the definition of “protection boundary”.	RFC00193.	43
Change “WIM” to “WIMG” (two places).	RFC00173.	57, 58
Clarify Programming Note about Machine Check corrupting registers.	RFC00207.	63
Describe which multiple DSISR bits may be set due to simultaneous Data Storage exceptions.	RFC00163 as amended at Nov. PAWG meeting.	65
Remove software synchronization requirements for TB and DEC.	RFC00182.	75-77
Clarify “monotonically increasing” in Programming Notes for Time Base.	RFC00206.	76
Say that ***rfi*** and interrupts change Endian mode reliably for I-fetch.	RFC00181. ***mtmsrd***, added by RFC00178, has been included in the revised Note.	80, 80
Simplify DAR setting for a DABR interrupt.	RFC00192.	84
Say that assemblers *should* provide the listed extended mnemonics, not that they *must*.	RFC00173.	93
Define “AIM” and use “-AIM” suffix on citations as needed.	RFC00203. The change proposed for the “New Instructions” appendix was not made because it is overridden by RFC00178.	various
Incorporate minor changes from the Morgan Kaufmann book. Not all such changes have been made; the rest will be made in future versions of this Book. Very minor changes (e.g., fixing grammatical errors) are not marked with change bars.	Agreed in discussion of RFC00173 at Nov. PAWG meeting.	various

Chapter 1. Introduction

1.1 Overview

Chapter 1 of Book I, *PowerPC AS User Instruction Set Architecture* describes computation modes, compatibility with the POWER Architecture, document conventions, a general systems overview, instruction formats, and storage addressing. This chapter augments that description as necessary for the PowerPC AS Operating Environment Architecture.

1.2 Compatibility with the POWER Architecture

The PowerPC AS Architecture provides binary compatibility for POWER application programs, except as described in the appendix entitled “Incompatibilities with the POWER Architecture” in Book I, *PowerPC AS User Instruction Set Architecture*. Binary compatibility is not necessarily provided for privileged POWER instructions.

1.3 Document Conventions

The notation and terminology used in Book I apply to this Book also, with the following substitutions.

- For “system alignment error handler” substitute “Alignment interrupt”.
- For “system data storage error handler” substitute “Data Storage interrupt”, “Data Segment interrupt”, or “Data Storage or Data Segment interrupt”, as appropriate.
- For “system error handler” substitute “interrupt”.
- For “system floating-point enabled exception error handler” substitute “Floating-Point Enabled Exception type Program interrupt”.
- For “system illegal instruction error handler” substitute “Illegal Instruction type Program Interrupt”.
- For “system instruction storage error handler” substitute “Instruction Storage interrupt”, “Instruction Segment interrupt”, or “Instruction Storage or Instruction Segment interrupt”, as appropriate.
- For “system privileged instruction error handler” substitute “Privileged Instruction type Program interrupt”.
- For “system service program” substitute “System Call interrupt”.
- For “system trap handler” substitute “Trap type Program interrupt”.

1.3.1 Definitions and Notation

The definitions and notation given in Book I, *PowerPC AS User Instruction Set Architecture* are augmented by the following.

- A real page is a 4 KB unit of real storage that is aligned at a 4 KB boundary.
- The context of a program is the environment (e.g., privilege and relocation) in which the program executes. That context is controlled by

the contents of certain System Registers, such as the MSR and SDR1, and of the address translation tables.

- An exception is an error, unusual condition, or external signal, that may set a status bit and may or may not cause an interrupt, depending upon whether the corresponding interrupt is enabled.
- An interrupt is the act of changing the machine state in response to an exception, as described in Chapter 7, "Interrupts" on page 59.
- A trap interrupt is an interrupt that results from execution of a *Trap* instruction.
- Additional exceptions to the rule that the processor obeys the sequential execution model, beyond those described in the section entitled "Instruction Fetching" in Book I, are the following.

  — A System Reset or Machine Check interrupt may occur. The determination of whether an instruction is required by the sequential execution model is not affected by the potential occurrence of a System Reset or Machine Check interrupt. (The determination *is* affected by the potential occurrence of any other kind of interrupt.)

  Engineering Note

  Although External, Decrementer, and imprecise interrupts must be considered in determining whether an instruction is required by the sequential execution model, the fact that these interrupts are not required to be recognized at any specific point in the instruction stream allows an implementation to halt instruction dispatching and delay recognition of the interrupt until the processor comes into a state consistent with the sequential execution model. Such an implementation need not consider these interrupts in determining whether an instruction is required by the sequential execution model.

  Instruction-caused precise interrupts must also be considered in determining whether an instruction is required by the sequential execution model. However, for these it is always possible to predict whether they might be caused by any given instruction and thus to determine whether subsequent instructions are sure to be required by the sequential execution model.

  — A context-altering instruction is executed (see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79). The context alteration need not take effect until the required subsequent synchronizing operation has occurred.

- Hardware means any combination of hard-wired implementation, emulation assist, or interrupt for software assistance. In the last case, the interrupt may be to an architected location or to an implementation-dependent location. Any use of emulation assists or interrupts to implement the architecture is described in Book IV, *PowerPC AS Implementation Features*.
- /, //, ///, ... denotes a field that is reserved in an instruction, in a register, or in an architected storage table.

1.3.2 Reserved Fields

Some fields of certain storage tables may be written to automatically by hardware, e.g., Reference and Change bits in the Page Table. When the hardware writes to such a table, the following rules are obeyed.

- Unless otherwise stated, no defined field other than the one(s) the hardware is specifically updating are modified.
- Contents of reserved fields are either preserved by hardware or written as 0s. No other changes to reserved fields are made.

The handling of reserved bits in System Registers described in Book I applies here as well. The reader should be aware that reading and writing of some of these registers (e.g., the MSR) can occur as a side effect of processing an interrupt and of returning from an interrupt, as well as when requested explicitly by the appropriate instruction (e.g., ***mtmsrd***).

Programming Note

System software should initialize reserved fields in architected storage tables (e.g., the Page Table) to 0s and not keep data in them, as the fields may be assigned a meaning in some future version of the architecture.

1.4 General Systems Overview

The processor or processor unit contains the sequencing and processing controls for instruction fetch, instruction execution, and interrupt action. Instructions that the processing unit can execute fall into three classes:

- instructions executed in the Branch Processor
- instructions executed in the Fixed-Point Processor
- instructions executed in the Floating-Point Processor

Almost all instructions executed in the Branch Processor, Fixed-Point Processor, and Floating-Point Processor are nonprivileged and are described in Book I, *PowerPC AS User Instruction Set Architecture*. Book II, *PowerPC AS Virtual Environment Architecture* may describe additional nonprivileged instructions (e.g., Book II describes some nonprivileged instructions for cache management). Instructions related to the privileged state of the processor, control of processor resources, control of the storage hierarchy, and all other privileged instructions are described here or in Book IV, *PowerPC AS Implementation Features*.

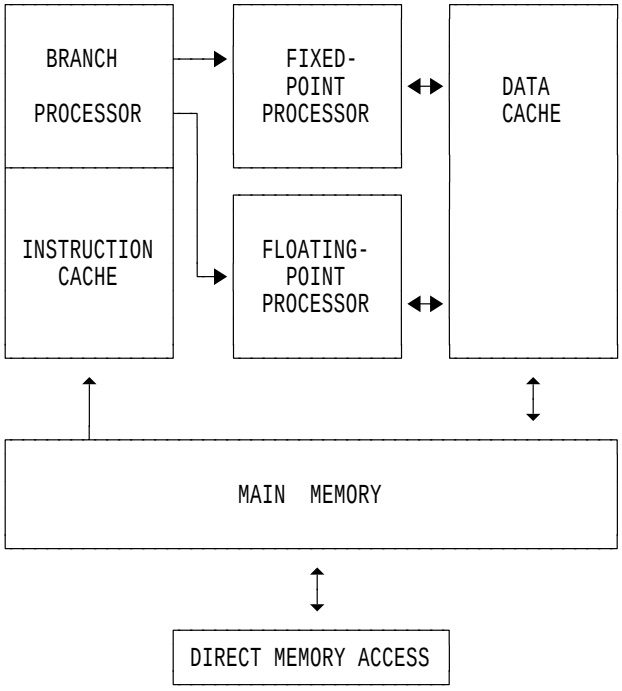


Figure 1. Logical view of the PowerPC AS processor architecture

1.5 Exceptions

The following augments the list, given in Book I, of exceptions that can be caused directly by the execution of an instruction:

- the execution of a floating-point instruction when $MSR_{FP}= 0$ (Floating-Point Unavailable interrupt)
- an attempt to modify a hypervisor resource when the processor is in privileged but non-hypervisor state (see Section 1.7), or an attempt to execute a hypervisor-only instruction (e.g., ***tlbie***) when the processor is in privileged but non-hypervisor state.
- the execution of a traced instruction (Trace interrupt)

1.6 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

1.6.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations.* Examples of context synchronizing operations include the ***sc*** instruction, the ***isync*** instruction, the ***rfid*** instruction, and most interrupts.

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions already in execution have completed to a point at which they have reported all exceptions they will cause.
3. The instructions that precede the operation complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.
4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 7.8, “Interrupt Priorities” on page 73).
5. The instructions that follow the operation will be fetched and executed in the context established

by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 4.2.4, "Performing Operations Out-of-Order" on page 25.)

A context synchronizing operation is necessarily execution synchronizing; see Section 1.6.2, "Execution Synchronization". Unlike the ***sync*** instruction, a context synchronizing operation does not affect the order in which storage accesses are performed with respect to other processors and mechanisms, or the order in which Reference, Change, and Tags Set bit updates are performed.

1.6.2 Execution Synchronization

An instruction is *execution synchronizing* if it satisfies items 2 on page 3 and 3 on page 3 of the definition of context synchronization (see Section 1.6.1). ***sync*** is treated like ***isync*** with respect to item 2 on page 3 (i.e., the conditions described in item 2 on page 3 apply to the completion of ***sync***). Examples of execution synchronizing instructions are ***sync*** and ***mtmsrd***. Also, all context synchronizing instructions are execution synchronizing.

Unlike a context synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

1.7 Logical Partitioning (LPAR)

The Logical Partitioning (LPAR) facility permits processors and portions of real storage to be assigned to logical collections called *partitions*, such that a program executing on a processor in one partition cannot interfere with any program executing on a processor in a different partition. This isolation can be provided for both problem state and privileged state programs, by using a layer of trusted software, called a *hypervisor* program (or simply a "hypervisor"), and the resources provided by this facility to manage system resources. (A hypervisor is a program that runs in hypervisor state; see below.)

The number of partitions supported is implementation-dependent.

A processor is in only one partition at any given time. Partitions can be defined without consideration of the physical configuration of the system (e.g., shared caches, organization of the storage hierarchy).

A processor may be removed from one partition and assigned to a different partition while other processors continue to execute programs in their respective partitions. The operations necessary to assign a processor to a different partition are implementation-dependent.

The following resources are provided to support logical partitioning.

1. HV bit of the MSR

   This bit, along with MSR_{PR}, controls whether the processor is in hypervisor state (see Section 2.2.3 on page 7).

2. Logical Partitioning Environment Selector (LPES) bit

   This bit affects how storage is accessed in real addressing mode (see Section 4.2.5 on page 27 and Section 4.9.3 on page 43) and how the MSR is set when an interrupt occurs (see Section 7.5 on page 62).

   Programming Note

   LPES=0 provides an environment in which only the hypervisor can run with address translation disabled and in which all interrupts except the System Call Vectored interrupt invoke the hypervisor. This value (along with MSR_{HV}=1) can also be used in a system that is not partitioned, to permit the operating system (except the System Call Vectored interrupt handler) to access all system resources.

3. Real mode storage access control

   The Real Mode Offset Register (RMOR), Real Mode Limit Register (RMLR), and Real Mode Caching Inhibited bit control access to storage in real addressing mode, as described in Section 4.2.5 on page 27.

4. Logical Partition Identity Register (LPIDR)

   This register contains a value that identifies the partition to which the processor is assigned.

With the exception of MSR_{HV}, the format and contents of these resources, the conditions that must be established before they are altered, the means provided for altering them, and the software synchronization required in order to make the alterations effective are implementation-dependent.

With the exception of MSR_{HV}, the resources defined above and those in the following list are hypervisor resources.

- All implementation-specific resources, including implementation-specific registers (e.g., "HID" registers), that control hardware functions or affect the results of instruction execution. Examples

include resources that disable caches, disable hardware error detection, set breakpoints, control power management, or significantly affect performance.

- ME bit of the MSR
- SDR1, EAR, SPRG0, Time Base, PIR, DABR (if implemented)
- the large virtual page size, if a means is provided by which software can alter it

The contents of a hypervisor resource can be modified by the execution of an instruction (e.g., ***mtspr***) only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). Whether an attempt to modify the contents of a given hypervisor resource, other than MSR_{ME}, in privileged but non-hypervisor state ($MSR_{HV\ PR}$ = 0b00) is ignored (i.e., treated as a no-op) or causes a Privileged Instruction type Program interrupt is implementation-dependent. An attempt to modify MSR_{ME} in privileged but non-hypervisor state is ignored (i.e., the bit is not changed).

Engineering Note

Causing a Privileged Instruction type Program interrupt if attempt is made to modify the contents of a hypervisor resource in privileged but non-hypervisor state facilitates the debugging of software.

The ***tlbie*** and ***tlbsync*** instructions can be executed only in hypervisor state; see the descriptions of these instructions on pages 55 and 56.

In general, if software violates a rule that is stated in the Books using the word “must” (e.g., “this field must be set to 0”) the results are boundedly undefined. The only exception is that if hypervisor software violates such a rule that pertains to the contents of a hypervisor resource, to accessing storage in real addressing mode, or to using the ***tlbie*** and ***tlbsync*** instructions, the results are undefined, and may include altering resources belonging to other partitions, causing the system to “hang”, etc.

Programming Note

Because the SPRs listed above are privileged for writing, an attempt to modify the contents of any of these SPRs in problem state (MSR_{PR}=1) using ***mtspr*** causes a Privileged Instruction type Program exception, and similarly for MSR_{ME}.

If the hypervisor sets a breakpoint for an operating system program without verifying the requested breakpoint conditions, the breakpoint could cause an unexpected Data Storage interrupt when the hypervisor is executing.

Architecture Note

MSR_{ME}, the SPRs listed above, and the large virtual page size are hypervisor resources because they must be altered only by hypervisor software. Consequences of permitting alteration by non-hypervisor software include the following.

MSR_{ME}: Non-hypervisor software could cause a subsequent Machine Check to cause a system-wide Checkstop.

SDR1, large virtual page size: Non-hypervisor software could access storage not allocated to the partition in which it is running.

EAR: Non-hypervisor software could access memory controller resources (corresponding to values of EAR_{RID}) not allocated to the partition in which it is running.

SPRG0: Non-hypervisor software could cause the hypervisor to use invalid data (see the intended use of this register described in Section 3.3.3).

Time Base: Non-hypervisor software could cause the Time Base on one processor to be out of synchronization with that on other processors, with the result that the first processor's Time Base would have to be resynchronized as part of assigning the processor to a different partition.

PIR: Because the PIR is used in communication with other processors and with I/O devices, non-hypervisor software could cause the system to “hang”.

DABR: Non-hypervisor software could set a breakpoint in hypervisor data.

Engineering Note

On an implementation that provides a Performance Monitor facility (e.g., see Appendix E), any Performance Monitor resource having the property that alteration of the resource by a processor in one partition could affect the integrity of other partitions must be a hypervisor resource. (It is expected that most Performance Monitor resources will not have this property.)

Control bits that are hypervisor resources should not be defined in registers or resources that contain bits that can be altered by non-hypervisor programs.

Engineering Note

The requirements for altering hypervisor resources must be such that a processor assigned to one partition can be reassigned to a different partition without affecting the execution of programs on other processors. In addition, deterministic means must be provided to perform other functions associated with reassigning a processor to a different partition, such as invalidating SLB, TLB, and ERAT entries.

The speed with which this reassignment can be performed may affect how the LPAR facility is used. Decisions regarding how reassignment is accomplished in a given implementation must include consideration of the intended uses of the facility and of the consequent performance requirements.

Chapter 2. Branch Processor

2.1 Branch Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Branch Processor that are not covered in Book I, *PowerPC AS User Instruction Set Architecture*.

2.2 Branch Processor Registers

2.2.1 Machine Status Save/Restore Register 0

The Machine Status Save/Restore Register 0 (SRR0) is a 64-bit register. This register is used to save machine status on interrupts, except for System Call Vectored interrupts, and to restore machine status when an ***rfid*** instruction is executed.

On interrupt, SRR0 is set to the current or next instruction address. Thus if the interrupt occurs in 32-bit mode, the high-order 32 bits of SRR0 are set to 0. When ***rfid*** is executed, the contents of SRR0 are copied to the next instruction address (NIA), except that the high-order 32 bits of the NIA are set to 0 when returning to 32-bit mode.

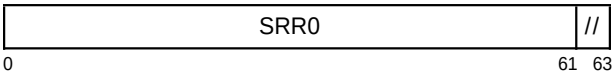


Figure 2. Save/Restore Register 0

In general, SRR0 contains either the address of the instruction that caused the interrupt, or the address of the instruction to return to after an interrupt is serviced.

2.2.2 Machine Status Save/Restore Register 1

The Machine Status Save/Restore Register 1 (SRR1) is a 64-bit register. This register is used to save machine status on interrupts, except for System Call Vectored interrupts, and to restore machine status when an ***rfid*** instruction is executed.

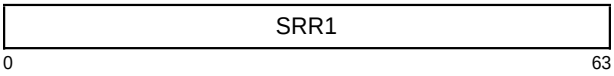


Figure 3. Save/Restore Register 1

In general, when an interrupt occurs, bits 33:36 and 42:47 of SRR1 are loaded with information specific to the interrupt type, and bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bit positions of SRR1.

SRR1 bits in the range 0:32, 37:41, and 48:63 may be treated as reserved in a given implementation if they correspond to MSR bits that are reserved or are treated as reserved in that implementation.

2.2.3 Machine State Register

The Machine State Register (MSR) is a 64-bit register. This register defines the state of the processor. On interrupt, the MSR bits are altered in accordance with Figure 30 on page 62. The MSR can also be modified by the ***mtmsr***[***d***], ***sc***, ***scv***, ***rfscv***, and ***rfid*** instructions. It can be read by the ***mfmsr*** instruction.

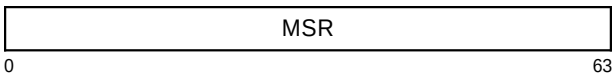


Figure 4. Machine State Register

Below are shown the bit definitions for the Machine State Register.

Architecture Note

Defined MSR bits are classified as either full function or partial function. Full function MSR bits are saved in SRR1 when an interrupt other than System Call Vectored interrupt occurs and restored by ***rfid*** or ***rfscv***, while partial function MSR bits are not saved or restored. Full function MSR bits lie in the range 0:32, 37:41, and 48:63, and partial function MSR bits lie in the range 33:36 and 42:47.

Reserved MSR bits in the "full function range" need not be saved or restored. If they are not restored then they must be written as 0; if they are not saved then for System Call Vectored interrupt the corresponding CTR bits must be written as 0, and for interrupts other than System Call Vectored interrupt the corresponding SRR1 bits must be written as 0. The properties of reserved bits in System Registers are such that this alternative behavior does not conflict with the descriptions of ***sc, scv, rfid***, and interrupt processing elsewhere in this Book.

Bit **Description**

0 ***Sixty-Four-Bit Mode*** (SF)

0 The processor is in 32-bit mode.
1 The processor is in 64-bit mode.

If $MSR_{SF\ TA}$ = 0b01, all results are boundedly undefined.

1 ***Tags Active Mode*** (TA)

0 The processor is in *tags inactive* mode.
1 The processor is in *tags active* mode.

If $MSR_{SF\ TA}$ = 0b01, all results are boundedly undefined.

Engineering Note

One way to ensure that results are limited to being boundedly undefined if software attempts to set $MSR_{SF\ TA}$ = 0b01 is to set MSR_{SF} to the OR of the supplied SF value and the supplied TA value whenever an ***rfscv, rfid,*** or ***mtmsrd*** instruction is executed. E.g., ***rfid*** would set MSR_0 to $SRR1_0$ | $SRR1_1$. This technique may simplify verification. (Interrupts are not a problem in this regard, because all interrupts set MSR_{SF} to 1.)

2 Reserved

Architecture Note

Bit 2 will be among the last to be assigned a meaning. It was the ISF (Interrupt Sixty-Four Bit Mode) bit in earlier versions of the architecture.

3 ***Hypervisor State*** (HV)

0 The processor is not in hypervisor state.
1 If MSR_{PR}= 0 the processor is in hypervisor state; otherwise the processor is not in hypervisor state.

Programming Note

The privilege state of the processor is determined by MSR_{HV} and MSR_{PR}, as follows.

HV	PR	
0	0	privileged
0	1	problem
1	0	privileged and hypervisor
1	1	problem

MSR_{HV} can be set to 1 only by the *System Call* instruction and some interrupts. It can be set to 0 only by the ***rfid*** instruction, and possibly by the ***rfscv*** instruction and some interrupts.

4:46 Reserved

Architecture Note

Bits 33 and 45 will be among the last to be assigned a meaning. In earlier versions of the architecture bit 33 was the C2 Security bit and bit 45 was the POW (Power Management) bit.

In the POWER Architecture, $SRR1_5$ is used by the Instruction Storage interrupt to indicate a loop in the translation mechanism. Because of this, MSR_{37} will not be assigned a new meaning in the near future.

Bit 46 is used on some implementations for an implementation-specific function. See the Book IV, *PowerPC AS Implementation Features* document for the implementation. (Bit 46 is the Temporary GPR Remapping (TGPR) bit on the 603.)

47 ***Interrupt Little-Endian Mode*** (ILE)

This bit is part of the optional Little-Endian facility; see the section entitled "Little-Endian" in Book I.

If the Little-Endian facility is implemented, when an interrupt occurs this bit is copied to MSR_{LE} to select the Endian mode for the context established by the interrupt.

If the Little-Endian facility is not implemented, this bit is treated as reserved.

48 ***External Interrupt Enable*** (EE)

0 External and Decrementer interrupts are disabled.
1 External and Decrementer interrupts are enabled.

49 ***Problem State*** (PR)

0 The processor is in privileged state.
1 The processor is in problem state.

Programming Note

Any instruction or event that sets MSR_{PR} to 1 also sets MSR_{IR} and MSR_{DR} to 1.

50 ***Floating-Point Available*** (FP)

0 The processor cannot execute any floating-point instructions, including floating-point loads, stores, and moves.
1 The processor can execute floating-point instructions.

51 ***Machine Check Enable*** (ME)

0 Machine Check interrupts are disabled.
1 Machine Check interrupts are enabled.

This bit is a hypervisor resource; see Section 1.7, "Logical Partitioning (LPAR)" on page 4.

Programming Note

The only instruction that can alter MSR_{ME} is the ***rfid*** instruction.

52 ***Floating-Point Exception Mode 0*** (FE0)

See below.

53 ***Single-Step Trace Enable*** (SE)

0 The processor executes instructions normally.
1 The processor generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction (unless that instruction is ***rfid*** or ***rfscv***, which are never traced). Successful completion means that the instruction caused no other interrupt.

54 ***Branch Trace Enable*** (BE)

0 The processor executes branch instructions normally.
1 The processor generates a Branch type Trace interrupt after completing the execution of a branch instruction, whether or not the branch is taken. See Book IV, *PowerPC AS Implementation Features*.

Branch tracing may not be present on all implementations. If the function is not implemented, this bit is treated as reserved.

55 ***Floating-Point Exception Mode 1*** (FE1)

See below.

56 ***User State*** (US)

In *tags inactive* mode this bit is treated as reserved.

In *tags active* mode this bit distinguishes between operating system code and user code for purposes of storage protection.

0 Operating system code is executing.
1 User code is executing.

Architecture Note

This bit corresponds to the AL bit of the POWER Architecture (see the appendix entitled "Incompatibilities with the POWER Architecture" in Book I). Therefore the *tags active* function of this bit should not be made available in *tags inactive* mode until POWER-compatible operating system code no longer needs to be supported on PowerPC AS processors.

57 Reserved

Architecture Note

Bit 57 will be among the last to be assigned a meaning. It was the IP (Interrupt Prefix) bit in earlier versions of the architecture.

58 ***Instruction Relocate*** (IR)

0 Instruction address translation is off.
1 Instruction address translation is on.

Programming Note

Any instruction or event that sets MSR_{IR} to 0 also sets MSR_{PR} to 0.

59 ***Data Relocate*** (DR)

0 Data address translation is off.
1 Data address translation is on.

Programming Note

Any instruction or event that sets MSR_{DR} to 0 also sets MSR_{PR} to 0.

60 Reserved

Architecture Note

Bit 60 will be among the last to be assigned a meaning. It was the FC (Firmly Consistent) bit in earlier versions of the architecture.

61 ***Performance Monitor Mark*** (PMM)

This bit is part of the optional Performance Monitor facility; see Appendix E. If the Performance Monitor facility is not implemented or does not use this bit, this bit is treated as reserved.

62 ***Recoverable Interrupt*** (RI)

0 Interrupt is not recoverable.
1 Interrupt is recoverable.

Additional information about the use of this bit is given in Sections 7.4, “Interrupt Processing” on page 61, 7.5.1, “System Reset Interrupt” on page 63, and 7.5.2, “Machine Check Interrupt” on page 63.

63 ***Little-Endian Mode*** (LE)

This bit is part of the optional Little-Endian facility; see the section entitled “Little-Endian” in Book I.

If the Little-Endian facility is implemented, this bit has the following meaning.

0 The processor is in Big-Endian mode.
1 The processor is in Little-Endian mode.

If the Little-Endian facility is not implemented, this bit is treated as reserved.

The Floating-Point Exception Mode bits FE0 and FE1 are interpreted as shown below. For further details see Book I, *PowerPC AS User Instruction Set Architecture*.

FE0	FE1	Mode
0	0	Ignore Exceptions
0	1	Imprecise Nonrecoverable
1	0	Imprecise Recoverable
1	1	Precise

Architecture Note

The initial state of the MSR should be as follows:

Bit	Name	tags inactive mode*	tags active mode*
0	SF	1	1
1	TA	*	*
2		unspecified	unspecified
3	HV	1	1
4:46		unspecified	unspecified
47	ILE	0	0
48	EE	0	0
49	PR	0	0
50	FP	0	0
51	ME	0	0
52	FE0	0	0
53	SE	0	0
54	BE	0	0
55	FE1	0	0
56	US	unspecified	0
57		unspecified	unspecified
58	IR	0	0
59	DR	0	0
60		unspecified	unspecified
61	PMM	unspecified	unspecified
62	RI	0	0
63	LE	0	0

* product-specific

2.3 Branch Processor Instructions

2.3.1 System Linkage Instructions

These instructions provide the means by which a program can call upon the system to perform a service, and by which the system can return from performing a service or from processing an interrupt.

The *System Call* instructions are described in Book I, *PowerPC AS User Instruction Set Architecture*, but only at the level required by an application programmer. A complete description of these instructions appears below.

System Call SC-form

sc LEV

[POWER mnemonic: svca]

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

$SRR0 \leftarrow_{iea} CIA +_{iea} 4$
$SRR1_{33:36\ 42:47} \leftarrow 0$
$SRR1_{0:32\ 37:41\ 48:63} \leftarrow MSR_{0:32\ 37:41\ 48:63}$
$MSR \leftarrow$ new_value (see below)
$NIA \leftarrow$ 0x0000_0000_0000_0C00

The effective address of the instruction following the *System Call* instruction is placed into SRR0. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of SRR1, and bits 33:36 and 42:47 of SRR1 are set to undefined values.

Then a System Call interrupt is generated. The interrupt causes the MSR to be altered as described in Section 7.5, “Interrupt Definitions” on page 62.

The interrupt causes the next instruction to be fetched from effective address 0x0000_0000_0000_0C00.

The contents of the LEV field must be 0 or 1; otherwise the results are boundedly undefined.

This instruction is context synchronizing.

Special Registers Altered:
SRR0 SRR1 MSR

Programming Note

sc serves as both a basic and an extended mnemonic. The Assembler will recognize an ***sc*** mnemonic with one operand as the basic form, and an ***sc*** mnemonic with no operand as the extended form. In the extended form the LEV operand is omitted and assumed to be 0.

Programming Note

If LEV=1 the hypervisor is invoked.

If LPES=1, executing this instruction with LEV=1 is the only way that executing an instruction can cause hypervisor state to be entered.

Because this instruction is not privileged, it is possible for application software to invoke the hypervisor. However, such invocation should be considered a programming error.

Engineering Note

$LEV_{0:5}$ must be ignored by the processor.

Architecture Note

The requirement that $LEV_{0:5}$ contain zeros and be ignored by the processor permits these bits to be assigned a meaning in the future if that proves desirable.

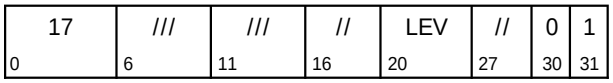
Compatibility Note

For a discussion of POWER compatibility with respect to instruction bits 16:19 and 27:29, see the appendix entitled “Incompatibilities with the POWER Architecture” in Book I, *PowerPC AS User Instruction Set Architecture*. For compatibility with future versions of the PowerPC AS Architecture, these bits should be coded as zeros.

System Call Vectored SC-form

scv LEV

[POWER mnemonic: svcl]




```
LR ← CIA +tia 4
CTR33:36 42:47 ← undefined
CTR0:32 37:41 48:63 ← MSR0:32 37:41 48:63
MSR ← new_value (see below)
NIA ← 0xFFFF_FFFF_FF00_3 || LEV || 0b0_0000
```

The effective address of the instruction following the *System Call Vectored* instruction is placed into the Link Register. Bits 0:32, 37:41, and 48:63 of the MSR are placed into the corresponding bits of Count Register, and bits 33:36 and 42:47 of Count Register are set to undefined values.

Then a System Call Vectored interrupt is generated. The interrupt causes the MSR to be altered as described in Section 7.5, “Interrupt Definitions” on page 62.

The interrupt causes the next instruction to be fetched from effective address 0xFFFF_FFFF_FF00_3 || LEV || 0b0_0000.

The SRRs are not affected.

This instruction is context synchronizing.

This instruction is available in *tags active* mode only. In *tags inactive* mode this is an illegal instruction.

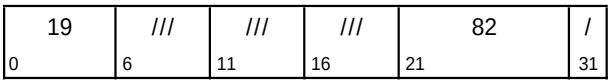
Special Registers Altered:

LR CTR MSR

Return From System Call Vectored XL-form

rfscv

[POWER mnemonic: rfsvc]




```
on some implementations MSR3 ← 0
MSR58 ← CTR58 | CTR49
MSR59 ← CTR59 | CTR49
MSR0:2 4:32 37:41 48:50 52:57 60:63 ← CTR0:2 4:32 37:41 48:50 52:57 60:63
NIA ← LR0:61 || 0b00
```

The result of ORing bits 58 and 49 of the Count Register is placed into MSR_{58}. The result of ORing bits 59 and 49 of the Count Register is placed into MSR_{59}. Bits 0:2, 4:32, 37:41, 48:50, 52:57, and 60:63 of the Count Register are placed into the corresponding bits of the MSR. It is implementation-dependent whether MSR_3 is set to 0 or is unchanged.

Then the next instruction is fetched, under the the control of the new MSR value, from the address $LR_{0:61}$ || 0b00.

This instruction is privileged and context synchronizing.

This instruction is available in *tags active* mode only. In *tags inactive* mode this is an illegal instruction.

Special Registers Altered:

MSR

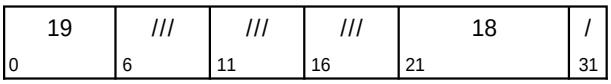
Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{IR} and MSR_{DR} to 1. This instruction does not alter MSR_{ME}.

This instruction should not be executed in hypervisor state, so the fact that some implementations do not set MSR_{HV} to 0 does not matter.

Return From Interrupt Doubleword XL-form

rfid




```
MSR51 ← (MSR3 & SRR151) | ((¬MSR3) & MSR51)
MSR3 ← MSR3 & SRR13
MSR58 ← SRR158 | SRR149
MSR59 ← SRR159 | SRR149
MSR0:2 4:32 37:41 48:50 52:57 60:63 ← SRR10:2 4:32 37:41 48:50 52:57 60:63
NIA ←iea SRR00:61 || 0b00
```

If MSR_3= 1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. The result of ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 0:2, 4:32, 37:41, 48:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or $^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 by the interrupt processing mechanism (see Section 7.4, “Interrupt Processing” on page 61) is the address of the instruction that would have been executed next had the interrupt not occurred.

This instruction is privileged and context synchronizing.

Special Registers Altered:

MSR

Programming Note

If this instruction sets MSR_{PR} to 1, it also sets MSR_{IR} and MSR_{DR} to 1.

This instruction is the only instruction that can be used to set MSR_{HV} to 0 on all implementations. This instruction is the only instruction that can be used to alter MSR_{ME}. These bits can be altered by this instruction only if it is executed in hypervisor state.

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the details concerning the registers and the privileged instructions implemented in the Fixed-Point Processor that are not covered in Book I, *PowerPC AS User Instruction Set Architecture*.

3.2 Special Purpose Registers

Special Purpose Registers (SPRs) are read and written using the ***mfspr*** (page 20) and ***mtspr*** (page 19) instructions. Most SPRs are defined in other chapters of this book; see the index to locate those definitions.

3.3 Fixed-Point Processor Registers

3.3.1 Data Address Register

The Data Address Register (DAR) is a 64-bit register | that is set by Data Storage, Data Segment, and Align- | ment interrupts. See Section 7.5.3, “Data Storage | Interrupt” on page 64, Section 7.5.4, “Data Segment | Interrupt” on page 65, and Section 7.5.8, “Alignment | Interrupt” on page 67. When one of these interrupts † occurs, the DAR is set to an effective address associ- † ated with the storage access caused by the interrupting instruction. If the interrupt occurs in 32-bit mode, the high-order 32 bits of the DAR are set to 0.

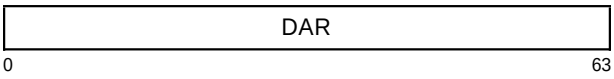


Figure 5. Data Address Register

|

3.3.2 Data Storage Interrupt Status Register

The Data Storage Interrupt Status Register (DSISR) is a 32-bit register that defines the cause of Data Storage and Alignment interrupts. See Section 7.5.3, “Data Storage Interrupt” on page 64 and Section 7.5.8, “Alignment Interrupt” on page 67.

Figure 6. Data Storage Interrupt Status Register

3.3.3 Software-Use SPRs

SPRG0 through SPRG3 are 64-bit registers provided for use by privileged software.

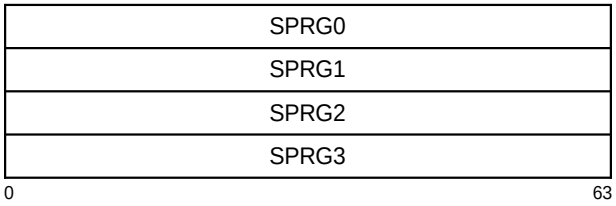


Figure 7. Software-use SPRs

The following list describes the conventional uses of SPRG0 through SPRG3.

SPRG0

Hypervisor software may keep a unique real address in this register to identify an area of storage reserved for use by the hypervisor first-level interrupt handler. This area must be unique for each processor in the system.

SPRG0 is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 4.

SPRG1

This register may be used as a scratch register by the first-level interrupt handler to save the contents of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to storage.

SPRG2

This register may be used by the operating system as needed.

SPRG3

This register may be used by the operating system as needed.

It is optional whether SPRG3 can be read in problem state. On implementations that provide this ability, SPRG3 may be used for information, such as a “thread-id”, that the operating system makes available to application programs.

Programming Note

On implementations for which SPRG3 can be read in problem state, operating systems must ensure that no sensitive data are left in SPRG3 when a problem state program is dispatched, and operating systems for secure systems must ensure that SPRG3 cannot be used to implement a “covert channel” between problem state programs. These requirements can be satisfied by clearing SPRG3 before passing control to a program that will run in problem state.

On such implementations, SPRG3 can be used “orthogonally” for both the purpose described for it above and the purpose described for SPRG1. If this is done, SPRG1 can be used for some other purpose.

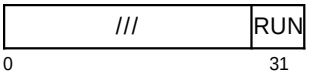
Engineering Note

The ability to read SPRG3 in problem state is being phased into the architecture, and will become required in a future version of the architecture.

3.3.4 Control Register

The Control Register (CTRL) is a 32-bit register that controls an external I/O pin. This signal may be used for the following:

- driving the RUN Light on a system operator panel
- External interrupt routing
- Performance Monitor event counting (see Appendix E, “Example Performance Monitor (Optional)” on page 105)



Bit	*Name*	*Description*
31	RUN	Run state bit

All other fields are implementation-dependent.

Figure 8. Control Register

The CTRL RUN can be used by the operating system to indicate when the processor is doing useful work.

The contents of the CTRL can be written by the ***mtspr*** instruction and read by the ***mfspr*** instruction. Write access to the CTRL is privileged. Reads can be performed in privileged or problem state.

3.3.5 Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register that contains a value identifying the version and revision level of the processor. The contents of the PVR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PVR is privileged; write access is not provided.

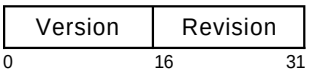


Figure 9. Processor Version Register

The PVR distinguishes between processors that differ in attributes that may affect software. It contains two fields.

Version A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.

Revision A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, such as clock rate and Engineering Change level.

Version numbers are assigned by the PowerPC AS Architecture process. Revision numbers are assigned by an implementation-defined process.

Engineering Note

Although the classification of a given difference between processors as "major" or "minor" is somewhat arbitrary, the following are examples of differences that generally should be considered "major".

- number and types of execution units
- optional facilities and instructions supported
- level of support of instructions (hard-wired or emulated)
- size, geometry, and management of caches and of TLBs

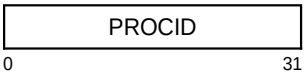
The following are examples of differences that generally should be considered "minor".

- remapping a processor to a new technology
- redesigning a critical path to increase clock rate
- fixing bugs

In general, any change to a processor should cause a new PVR value to be assigned. Even a seemingly trivial change that is not expected to be apparent to software should cause a new revision number to be assigned, in case the change is later discovered to have introduced an error that software must circumvent.

3.3.6 Processor Identification Register

† The Processor Identification Register (PIR) is a 32-bit register that contains a value that can be used to distinguish the processor from other processors in the system. The contents of the PIR can be copied to a GPR by the ***mfspr*** instruction. Read access to the PIR is privileged; write access, if provided, is described in the Book IV, *PowerPC AS Implementation Features* document for the implementation.



Bits	*Name*	*Description*
0:31	PROCID	Processor ID

Figure 10. Processor Identification Register

The means by which the PIR is initialized are implementation-dependent (see Book IV).

| The PIR is a hypervisor resource; see Section 1.7, | "Logical Partitioning (LPAR)" on page 4.

3.4 Fixed-Point Processor Privileged Instructions

3.4.1 Move To/From System Register Instructions

The *Move To Special Purpose Register* and *Move From Special Purpose Register* instructions are described in Book I, *PowerPC AS User Instruction Set Architecture*, but only at the level available to an † application programmer. For example, no mention is made there of registers that can be accessed only in | privileged state. The descriptions of these | instructions given below extend the descriptions given | in Book I, but do not list Special Purpose Registers | that are defined in Book IV, *PowerPC AS Implementation Features*. In the descriptions of these instructions | given below, the "defined" SPR numbers are the SPR | numbers shown in the figure for the instruction and | the SPR numbers defined in Book IV for the instruction, and similarly for "defined" registers.

Extended mnemonics

† Extended mnemonics are provided for the ***mtspr*** and ***mfspr*** instructions so that they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. See Appendix A, "Assembler Extended Mnemonics" on page 93.

Engineering Note

SPR numbers that are not shown in Figure 11 or Figure 12 and are in the ranges shown below are reserved for implementation-specific uses.

848 - 863
880 - 895
976 - 991
1008 - 1023

Implementation-specific registers must be privileged, and must comply with the other guidelines and limitations given in the Preface of Book I. SPR numbers for implementation-specific registers must be registered in advance with the person responsible for the technical content of this document (see the cover page).

Architecture Note

SPR numbers that are in the ranges 28-29, 80-82, 136-142, 144-159, 276-279, 512-639, and 972-973 are used in some early implementations for implementation-specific purposes. These SPR numbers will not be assigned a meaning in the PowerPC AS Architecture except after careful consideration of the effect of such assignment on existing implementations.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  SPREG(n) ← (RS)
else
  SPREG(n) ← (RS)32:63
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 11. The contents of register RS are placed into the designated Special Purpose Register. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RS are placed into the SPR.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

$spr_0=1$ if and only if writing the register is privileged. Execution of this instruction specifying a defined and † privileged register when $MSR_{PR}=1$ causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if $spr_0=0$: boundedly undefined results
- if $spr_0=1$:
  - if $MSR_{PR}=1$: Privileged Instruction type Program interrupt
  - if $MSR_{PR}=0$ and $MSR_{HV}=0$: boundedly undefined results
  - if $MSR_{PR}=0$ and $MSR_{HV}=1$: undefined results

If the SPR field contains a value that is shown in Figure 11 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were not shown in the figure.

Special Registers Altered:

See Figure 11

Compiler and Assembler Note

For the ***mtspr*** and ***mfspr*** instructions, the SPR number coded in assembler language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15. This maintains compatibility with POWER SPR encodings, in which these two instructions have only a 5-bit SPR field occupying bits 11:15.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
1	00000 00001	XER	no
8	00000 01000	LR	no
9	00000 01001	CTR	no
18	00000 10010	DSISR	yes
19	00000 10011	DAR	yes
22	00000 10110	DEC	yes
25	00000 11001	SDR1 [6]	hypv
26	00000 11010	SRR0	yes
27	00000 11011	SRR1	yes
29	00000 11101	ACCR	yes
152	00100 11000	CTRL	yes
272	01000 10000	SPRG0 [6]	hypv
273	01000 10001	SPRG1	yes
274	01000 10010	SPRG2	yes
275	01000 10011	SPRG3	yes
280	01000 11000	ASR [3]	yes
282	01000 11010	EAR [2,6]	hypv
284	01000 11100	TBL [6]	hypv
285	01000 11101	TBU [6]	hypv
784-799	11000 1xxxx	perf_mon[4]	yes
1013	11111 10101	DABR [5,6]	hypv

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] Part of the optional External Control facility (see Section 10.1).
[3] Part of the optional "Bridge" facility (see Section 11.1).
[4] Part of the optional Performance Monitor facility (see Appendix E).
[5] Part of the optional Data Address Breakpoint facility (see Section 10.2).
[6] This register is a hypervisor resource, and can be modified by this instruction only in hypervisor state (see Section 1.7).

All SPR numbers not shown above, or in Figure 12, or in Book IV are reserved.

Figure 11. SPR encodings for mtspr

Programming Note

For a discussion of software synchronization requirements when altering certain Special Purpose Registers, see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79.

Compatibility Note

For a discussion of POWER compatibility with respect to SPR numbers not shown in the instruction descriptions for ***mtspr*** and ***mfspr***, see the appendix entitled "Incompatibilities with the POWER Architecture" in Book I, *PowerPC AS User Instruction Set Architecture*.

Engineering Note

Causing an interrupt if this instruction is executed specifying an SPR number that is not defined for the implementation facilitates the debugging of software.

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if length(SPREG(n)) = 64 then
  RT ← SPREG(n)
else
  RT ← 32 0 || SPREG(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 12. The contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

$spr_0=1$ if and only if reading the register is privileged. Execution of this instruction specifying a defined and privileged register when $MSR_{PR}=1$ causes a Privileged Instruction type Program interrupt.

Execution of this instruction specifying an SPR number that is not defined for the implementation causes either an Illegal Instruction type Program interrupt or one of the following.

- if $spr_0=0$: boundedly undefined results
- if $spr_0=1$:
  - — if $MSR_{PR}=1$: Privileged Instruction type Program interrupt
  - — if $MSR_{PR}=0$: boundedly undefined results

If the SPR field contains a value that is shown in Figure 12 but corresponds to an optional Special Purpose Register that is not provided by the implementation, the effect of executing this instruction is the same as if the SPR number were not shown in the figure.

Special Registers Altered:

None

Note

See the Notes that appear with ***mtspr***.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
1	00000 00001	XER	no
8	00000 01000	LR	no
9	00000 01001	CTR	no
18	00000 10010	DSISR	yes
19	00000 10011	DAR	yes
22	00000 10110	DEC	yes
25	00000 11001	SDR1	yes
26	00000 11010	SRR0	yes
27	00000 11011	SRR1	yes
29	00000 11101	ACCR	yes
136	00100 01000	CTRL	no
272	01000 10000	SPRG0	yes
273	01000 10001	SPRG1	yes
274	01000 10010	SPRG2	yes
259,275	01000 n0011	SPRG3 [6,7]	no,yes
280	01000 11000	ASR [3]	yes
282	01000 11010	EAR [2]	yes
287	01000 11111	PVR	yes
768-799	11000 nxxxx	perf_mon [4,7]	no,yes
1013	11111 10101	DABR [5]	yes
1023	11111 11111	PIR	yes

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] Part of the optional External Control facility (see Section 10.1).
[3] Part of the optional "Bridge" facility (see Section 11.1).
[4] Part of the optional Performance Monitor facility (see Appendix E).
[5] Part of the optional Data Address Breakpoint facility (see Section 10.2).
[6] The ability to read SPRG3 in problem state is optional (see Section 3.3.3). If this ability is not provided by the implementation, SPR number 259 is treated as if it corresponded to an optional SPR that is not provided by the implementation.
[7] Reading the SPR is privileged if and only if n=1.

Moving from the Time Base (TB and TBU) is accomplished with the ***mftb*** instruction, described in Book II.

All SPR numbers not shown above, or in Figure 11, or in Book IV are reserved.

Figure 12. SPR encodings for mfspr

Move To Machine State Register Doubleword X-form

mtmsrd RS

31	RS	///	///	178	/
0	6	11	16	21	31

| $MSR_{58} \leftarrow (RS)_{58} \mid (RS)_{49}$
| $MSR_{59} \leftarrow (RS)_{59} \mid (RS)_{49}$
| $MSR_{0:2\ 4:50\ 52:57\ 60:63} \leftarrow (RS)_{0:2\ 4:50\ 52:57\ 60:63}$

| The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 0:2, 4:50, 52:57, and 60:63 of register RS are placed into the corresponding bits of the MSR.

This instruction is privileged. This instruction is execution synchronizing except with respect to alterations to the LE bit; see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79.

In addition, alterations to the EE and RI bits are effective as soon as the instruction completes. Thus if MSR_{EE}= 0 and an External or Decrementer interrupt is pending, executing an ***mtmsrd*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to be taken before the next instruction is executed, if no higher priority exception exists (see Section 7.8, "Interrupt Priorities" on page 73).

Special Registers Altered:
MSR

| **Programming Note**

| If this instruction sets MSR_{PR} to 1, it also sets MSR_{IR} and MSR_{DR} to 1.

| This instruction does not alter MSR_{HV} or MSR_{ME}.

† **Programming Note**

† For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 9.

Move From Machine State Register X-form

mfmsr RT

31	RT	///	///	83	/
0	6	11	16	21	31

RT ← MSR

The contents of the MSR are placed into register RT.

This instruction is privileged.

Special Registers Altered:
None

Chapter 4. Storage Control, Tags Active

4.1 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 4.3, “Address Translation Overview” on page 30 and following sections. The real address is what is presented to the storage subsystem. See Figure 13 on page 30.

For a complete discussion of storage addressing and effective address calculation, see the section entitled “Storage Addressing” in Book I, *PowerPC AS User Instruction Set Architecture*.

† Tags Active vs. Tags Inactive

The selection between *tags active* and *tags inactive* operation is made by MSR_{TA}. This chapter describes storage control in *tags active* mode.

† Storage Control Overview

- Real address space size is 2^m bytes, $m \leq 62$; see Note 1.
- Real page size is 2^{12} bytes (4 KB).
- † Effective address space size is 2^{64} bytes.
- There are two ways to translate an effective address to a virtual address. (A virtual address is always translated to a real address via the Page Table.)
  - — A Process Local Storage (PLS) effective address is translated via the Segment Look-aside Buffer (SLB) to a virtual address.
    - — Virtual address space size is 2^n bytes, $65 \leq n \leq 80$; see Note 2.
    - — Segment size is 2^{28} bytes (256 MB).
    - — Number of virtual segments is 2^{n-28}; see Note 2.
    - — Virtual page size is 2^p bytes, $12 \leq p \leq 28$; two sizes are supported simultaneously, 4 KB (p=12) and a larger size; see Note 3.
  - — A Single Level Storage (SLS) effective address is used directly as the virtual address (no SLB lookup).
    - — Virtual address space size is $2^{64} - 2^{48}$ bytes.
    - — Segment size is 2^{24} bytes (16 MB).
    - — Number of virtual segments is $2^{40} - 2^{24}$.
    - — Virtual page size is 2^{12} bytes (4 KB).

Notes:

1. The value of m is implementation-dependent (subject to the maximum given above). When used to address storage, the high-order 62– m bits of the "62-bit" real address must be zeros.
2. The value of n is implementation-dependent (subject to the range given above). In references to 80-bit virtual addresses elsewhere in this Book, the high-order 80– n bits of the "80-bit" virtual address are assumed to be zeros.
3. The value of p for the larger virtual page size is implementation-dependent (subject to the range given above).

4.2 Storage Model

The storage model provides the following features.

1. The architecture allows the storage implementations to take advantage of the performance benefits of weak ordering of storage accesses between processors or between processors and I/O devices.
2. The architecture provides instructions that allow the programmer to ensure a consistent and ordered storage state.

   - ***dcbf***
   - ***dcbst***
   - ***eieio***
   - ***icbi***
   - ***isync***
   - ***ldarx***
   - ***lwarx***
   - ***lwsync***
   - ***stdcx.***
   - ***stwcx.***
   - ***sync***
   - ***tlbsync***

3. Storage accesses appear to be performed in program order with respect to the processor performing them but, in general, may be performed in different orders with respect to other processors and mechanisms.
4. Storage consistency between processors, and between a processor and an I/O device, is controlled by software using the "WIM" storage control bits (see Section 4.7). These bits allow software to control whether a given storage location has any of the following attributes.
   - Write Through Required (W)
   - Caching Inhibited (I)
   - Memory Coherence Required (M)

Engineering Note

The architecture does not suggest or preclude any implementation of storage consistency supporting the features listed above. In particular, the implementation may be a snoopy bus design, a centralized cache directory design, or other design.

4.2.1 Storage Exceptions

A *storage exception* is an exception that causes an Instruction Storage interrupt, an Instruction Segment interrupt, a Data Storage interrupt, a Data Segment interrupt, or an Alignment interrupt. Attempting to fetch or execute an instruction causes a storage exception if certain conditions apply. Such conditions include the following.

- The appropriate relocate bit in the MSR is set to 1 and the effective address cannot be translated to a real address.
- The access is not permitted by the storage protection mechanism.
- The access causes a Data Address Compare match or a Data Address Breakpoint match.

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a *Load* or *Store* instruction. See the section entitled "Instruction Restart" in Book II, *PowerPC AS Virtual Environment Architecture*, and Section 7.6, "Partially Executed Instructions" on page 72 in this Book.

4.2.2 Instruction Fetch

Instructions are fetched under control of MSR_{IR}.

MSR_{IR}= 0

The effective address of the instruction is interpreted as described in Section 4.2.5, "Real Addressing Mode" on page 27.

MSR_{IR}= 1

The effective address of the instruction is translated by the Address Translation mechanism. (If it cannot be translated, a storage exception occurs.)

4.2.2.1 Implicit Branch

Explicitly altering certain MSR bits (using ***mtmsr***[***d***]), or explicitly altering SLB entries, Page Table entries, or certain System Registers, may have the side effect of changing the addresses, effective or real, from which the current instruction stream is being fetched. This side effect is called an *implicit branch*. For example, an ***mtmsrd*** instruction that changes the value of MSR_{SF} may change the effective addresses from which the current instruction stream is being fetched. The MSR bits and System Registers for which alteration can cause an implicit branch are indicated as such in Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79. Implicit branches are not supported by the PowerPC AS Architecture. If an implicit branch occurs, the results are boundedly undefined.

4.2.3 Data Access

Data accesses are controlled by MSR_{DR}.

MSR_{DR}= 0

The effective address of the data is interpreted as described in Section 4.2.5, "Real Addressing Mode" on page 27.

MSR_{DR}= 1

The effective address of the data is translated by the Address Translation mechanism. (If it cannot be translated, a storage exception occurs.)

4.2.4 Performing Operations Out-of-Order

An operation is said to be performed "in-order" if, at the time that it is performed, it is known to be required by the sequential execution model. An operation is said to be performed "out-of-order" if, at the time that it is performed, it is not known to be required by the sequential execution model.

Architecture Note

In earlier versions of the architecture specification, "speculative" was used instead of "out-of-order". The terminology was changed to be consistent with the technical literature, where "speculative execution" often means the execution of instructions past unresolved branches and "out-of-order execution" means execution of an instruction before it is known to be required by the sequential execution model. Because the meaning of "speculative" in the literature differs from ordinary English usage the term would cause confusion no matter how the architecture specification defined it, so the term is no longer used here at all.

Operations are performed out-of-order by the hardware on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are really needed is contingent on everything that might divert the control flow away from the instruction, such as *Branch, Trap, System Call, System Call Vectored*, ***rfid***, and ***rfscv*** instructions, and interrupts, and on everything that might change the context in which the instruction is executed.

Typically, the hardware performs operations out-of-order when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, the processor abandons any results of the operation (except as described below).

In the remainder of this section, including its subsections, "*Load* instruction" includes the *Cache Management* and other instructions that are stated in the instruction descriptions to be "treated as a *Load*", and similarly for "*Store* instruction".

Most operations can be performed out-of-order, as long as the machine appears to follow the sequential execution model. Certain out-of-order operations are restricted, as follows.

- **Stores**

  Stores are performed in-order (even if the *Store* instructions that caused them were executed out-of-order).

- **Accessing Guarded Storage**

  The restrictions for this case are given in Section 4.2.4.2.

No error of any kind other than Machine Check may be reported due to an operation that is performed out-of-order, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effects (other than Machine Check) of performing an operation out-of-order are the following.

- Reference, Change, and Tag Set bits may be set as described in Section 4.8, "Reference, Change, and Tag Set Recording" on page 40.
- Non-Guarded storage locations that could be fetched into a cache by in-order execution may be fetched out-of-order into that cache.

Engineering Note

Out-of-order execution of the ***stwcx.*** and ***stdcx.*** instructions is extremely complex and is not recommended.

Engineering Note

Because an External or Decrementer exception can become pending at any time, it might seem that if MSR_{EE}= 1 then fetching or executing any instruction beyond the current instruction is an out-of-order operation. However, these operations need not be treated as out-of-order if the taking of the interrupt is delayed until after they have completed. Similar considerations apply to Floating-Point Enabled Exception type Program interrupts when one of the Imprecise floating-point exception modes is in effect.

Engineering Note

Implementations that perform operations out-of-order must take care to obey the sequential execution model except as permitted by the architecture. Examples of cases that may require special attention include the following.

- Changes of control flow, including ***sc***, ***scv***, *Trap*, ***rfid***, ***rfscv***, and interrupts as well as branches.
- Changes of context due to changes of control flow. For example, the code at a branch target location, or the handler for System Call, System Call Vectored, or Trap interrupts, may change the context and then return, so that the instructions immediately following the *Branch*, ***sc***, ***scv***, or *Trap* execute in a new context.
- Changes to resources that affect address translation, storage protection, or storage control attributes, when the change is followed by the appropriate software synchronization. Such resources include $MSR_{SF\ TA\ PR\ US\ IR\ DR}$, SDR1, EAR, Page Tables, SLBs, and TLBs.
- Execution synchronizing and context synchronizing operations.

4.2.4.1 Guarded Storage

Storage is said to be "well-behaved" if the corresponding real storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

† Storage is said to be Guarded if either of the following conditions is satisfied.

- † MSR bit IR or DR is 1 for instruction fetches or data accesses respectively, and the G bit is 1 in the relevant Page Table Entry.
- MSR bit IR or DR is 0 for instruction fetches or data accesses respectively, and the optional Real Mode Storage Control facility (see Section 10.3) is not implemented. In this case all of storage is Guarded for the corresponding accesses.

In general, storage that is not well-behaved should be Guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a Machine Check.

The following rules apply to in-order execution of *Load* and *Store* instructions for which the first byte of the storage operand is in storage that is both Caching Inhibited and Guarded.

- *Load* or *Store* instruction that causes an atomic access

  If any portion of the storage operand has been accessed and an External, Decrementer, or Imprecise mode Floating-Point Enabled exception is pending, the instruction completes before the interrupt occurs.

- *Load* or *Store* instruction that causes an Alignment exception, or that causes a Data Storage exception for reasons other than Data Address Compare match or Data Address Breakpoint match

  The portion of the storage operand that is in Caching Inhibited and Guarded storage is not accessed.

  (The corresponding rules for instructions that cause a Data Address Compare match or Data Address Breakpoint match are given in Sections 4.6 and 10.2 respectively.)

Architecture Note

The rules for accessing Guarded storage when an Imprecise mode Floating-Point Enabled exception is pending should be revisited when the architecture is clarified with respect to those modes. For example, it may be acceptable to require software synchronization between any instruction that could cause a floating-point enabled exception in Imprecise mode and a subsequent instruction that accesses Guarded storage. (A *Floating-Point Status and Control Register* instruction might provide sufficient synchronization.)

4.2.4.2 Out-of-Order Accesses to Guarded Storage

In general, Guarded storage is not accessed out-of-order. The only exceptions to this rule are the following.

Load Instruction

† If a copy of any byte of the storage operand is in a
† cache then that byte may be accessed in the cache or in main storage.

Instruction Fetch

If $MSR_{IR}=0$ then an instruction may be fetched if any of the following conditions are met.

1. The instruction is in a cache. In this case it may be fetched from the cache or from main storage.
2. The instruction is in a real page from which an instruction has previously been fetched, except that if that previous fetch was based on condition
† 1 then the previously fetched instruction must have been in the instruction cache.
3. The instruction is in the same real page as an instruction that is required by the sequential execution model, or is in the real page immediately following such a page.

Programming Note

Software should ensure that only well-behaved storage is copied into a cache, either by accessing as Caching Inhibited (and Guarded) all storage that may not be well-behaved, or by accessing such storage as not Caching Inhibited (but Guarded) and referring only to cache blocks that are well-behaved.

If a real page contains instructions that will be
† executed when $MSR_{IR}=0$, software should ensure that this real page and the next real page contain only well-behaved storage (or, if the optional Real Mode Storage Control Facility is implemented, that this real page is not Guarded).
†

Engineering Note

† When $MSR_{IR}=0$ or $MSR_{DR}=0$, performance may be significantly degraded because all of storage defaults to being Guarded for the corresponding accesses. If it is important to avoid this degradation, a means of specifying portions of real storage that are treated as non-Guarded in real
† addressing mode should be provided as described in Section 10.3, “Real Mode Storage Control” on page 86.

4.2.5 Real Addressing Mode

Instruction fetches are performed in “real addressing mode” if instruction address translation is disabled ($MSR_{IR}=0$). Data accesses are performed in real addressing mode if data address translation is disabled ($MSR_{DR}=0$). Storage accesses in real addressing mode are performed in a manner that depends on the contents of MSR_{HV}, LPES, and the RMLR and RMOR (see Section 1.7, “Logical Partitioning (LPAR)” on page 4), as described below. In all cases, bits 0:1 of the effective address are ignored and, on implementations that support a real address size of only m bits, $m<62$, bits 2:63–m of the effective address may be ignored.

- If $MSR_{HV}=1$, bits 2:63 of the effective address are used as the real address for the access.

- If MSR_{HV}= 0 and LPES=0, the access causes a storage exception as described in Section 4.9.3, “Storage Protection, Address Translation Disabled” on page 43.
- If MSR_{HV}= 0 and LPES=1, the Offset Real Mode Address mechanism, described in Section 4.2.5.1, controls the access.

4.2.5.1 Offset Real Mode Address

If MSR_{HV}= 0 and LPES=1, the access is controlled by the contents of the Real Mode Limit Register and Real Mode Offset Register, as follows.

Real Mode Limit Register (RMLR)

If bits 2:63 of effective address for the access are greater than or equal to the value (limit) represented by the contents of the RMLR, the access causes a storage exception (see Section 4.9.3). The RMLR supports effective address limits that are powers of 2. The number and values of the limits supported are implementation-dependent.

Real Mode Offset Register (RMOR)

If the access is permitted by the RMLR, the effective address for the access is ORed with the offset represented by the contents of the RMOR and the low-order m bits of the result are used as the real address for the access. The number and values of the offsets supported are implementation-dependent.

Programming Note

The offset specified by the RMOR should be a non-zero multiple of the limit specified by the RMLR. If these registers are set thus, ORing the effective address with the offset produces a result that is equivalent to adding the effective address and the offset. (The offset must not be zero, because real page 0 contains the fixed interrupt vectors and real pages 1 and 2 may be used for implementation-specific purposes; see Section 4.2.6, “Address Ranges Having Defined Uses” on page 29.)

Engineering Note

Ignoring bits 2:63– m of the effective address simplifies the real mode limit check. Specifically, if the minimum limit value supported by the implementation is 2^k, only bits 64– m:63– k of the effective address need be checked.

4.2.5.2 Storage Control Attributes for Real Addressing Mode and for Implicit Storage Accesses

Storage accesses in real addressing mode are performed as though all of storage had the following storage control attributes, except as modified by the optional Real Mode Storage Control facility (see Section 10.3) if that facility is implemented. (The storage control attributes are defined in Book II, *PowerPC AS Virtual Environment Architecture*.)

- not Write Through Required
- not Caching Inhibited, for instruction fetches
- not Caching Inhibited, for data accesses if the Real Mode Caching Inhibited bit is set to 0; Caching Inhibited, for data accesses if the Real Mode Caching Inhibited bit is set to 1
- Memory Coherence Required, for data accesses
- Guarded

Implicit accesses to the Page Table by the processor in performing address translation and in recording reference, change, and tag set information are performed as though the storage occupied by the Page Table had the following storage control attributes.

- not Write Through Required
- not Caching Inhibited
- Memory Coherence Required
- not Guarded

These implicit accesses are ordered by the ***sync*** instruction in the same manner as are explicit storage accesses.

Software must ensure that any data storage location that is accessed with the Real Mode Caching Inhibited bit set to 1 is not in the caches.

Software must ensure that the Real Mode Caching Inhibited bit contains 0 whenever data address translation is enabled and whenever the processor is not in hypervisor state.

Programming Note

Because storage accesses in real addressing mode do not use the SLB or the Page Table, accesses in this mode bypass all checking and recording of information contained therein (e.g., storage protection checks that use information contained therein are not performed, and reference, change, and tag set information is not recorded).

The Real Mode Caching Inhibited bit can be used to permit a control register on an I/O device to be accessed without permitting the corresponding storage location to be copied into the caches. The bit should normally contain 0. Software would set the bit to 1 just before accessing the control register, access the control register as needed, and then set the bit back to 0.

4.2.6 Address Ranges Having Defined Uses

The address ranges described below have uses that are defined by the architecture.

- Fixed interrupt vectors

  Except for the first 256 bytes, which are reserved for software use, the real page beginning at real address 0x0000_0000_0000_0000 is either used for interrupt vectors or reserved for future interrupt vectors.

- Implementation-specific use

  The two contiguous real pages beginning at real address 0x0000_0000_0000_1000 are reserved for implementation-specific purposes.

- Offset Real Mode interrupt vectors

  The real page beginning at the real address specified by the RMOR is used similarly to the page for the fixed interrupt vectors.

- *System Call Vectored* interrupt vectors

  The virtual page containing the byte addressed by effective address 0xFFFF_FFFF_FF00_3000 contains the interrupt vectors that are invoked by the *System Call Vectored* instruction.

- Page Table

  A contiguous sequence of real pages beginning at the real address specified by SDR1 contains the Page Table.

4.2.7 Invalid Real Address

A storage access (including an access that is performed out-of-order; see Section 4.2.4) may cause a Machine Check if the accessed storage location contains an uncorrectable error or does not exist. In the latter case the Checkstop state may be entered. See Section 7.5.2, “Machine Check Interrupt” on page 63.

Programming Note

Hypervisor software must ensure that a storage access by a program in one partition will not cause a Checkstop or other system-wide event that could affect the integrity of other partitions (see Section 1.7, “Logical Partitioning (LPAR)” on page 4). For example, such an event could occur if a real address placed in a Page Table Entry or made accessible to a partition using the Offset Real Mode Address mechanism (see Section 4.2.5.1) does not exist.

4.3 Address Translation Overview

Figure 13 gives an overview of the address translation process in *tags active* mode.

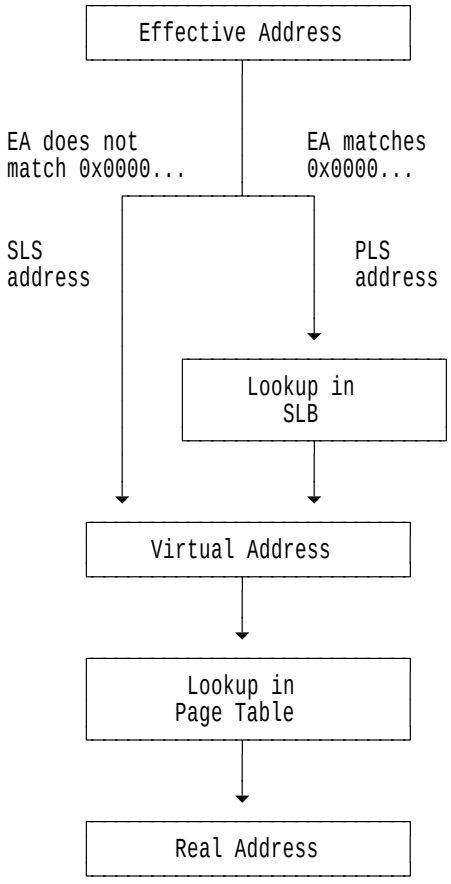


Figure 13. PowerPC AS address translation, tags active

The effective address (EA) is the address generated by the processor for an instruction fetch or for a data access. If address translation is enabled (MSR_{IR}= 1 or MSR_{DR}= 1 as appropriate), this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 4.4. The second step, conversion of the virtual address to a real address (RA), is described in Section 4.5.

If the effective address cannot be translated, a storage exception (see Section 4.2.1) occurs.

4.4 Virtual Address Generation

PLS effective addresses ($EA_{0:15}$=0x0000) in *tags active* mode and all effective addresses in *tags inactive* mode are translated to virtual addresses as described in Section 4.4.1. SLS effective addresses ($EA_{0:15}\neq$ 0x0000) are translated to virtual addresses as described in Section 4.4.2.

4.4.1 Virtual Address Generation, Tags Inactive Mode or PLS Address

For a PLS effective address in *tags active* mode and for any effective address in *tags inactive* mode, conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB) as shown in Figure 14.

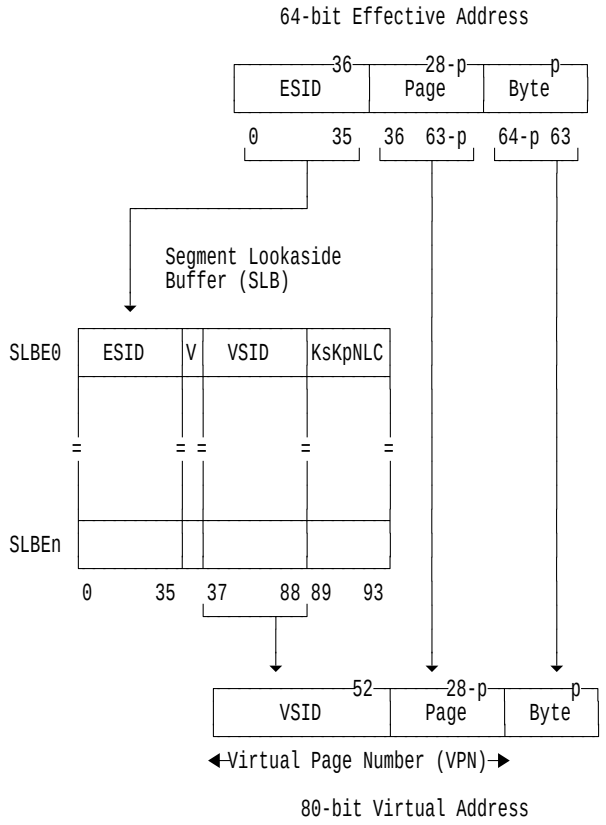


Figure 14. Translation of 64-bit effective address to 80-bit virtual address, tags inactive mode or PLS address

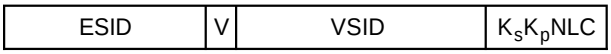
4.4.1.1 Segment Lookaside Buffer (SLB)

The Segment Lookaside Buffer (SLB) specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries.

The contents of the SLB are managed by software, using the instructions described in Section 6.1.2.1, "SLB Management Instructions" on page 50. See Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79 for the rules that software must follow when updating the SLB.

SLB Entry

Each SLB entry (SLBE) maps one ESID to one VSID. Figure 15 shows the layout of an SLB entry.



Bit(s)	*Name*	*Description*
0:35	ESID	Effective Segment ID
36	V	Entry valid (V=1) or invalid (V=0)
37:88	VSID	Virtual Segment ID
89	K_s	Supervisor (privileged) state storage key
90	K_p	Problem state storage key
91	N	No-execute segment if N=1
92	L	Virtual pages are large (L=1) or 4 KB (L=0)
93	C	Class

Figure 15. SLB Entry

On implementations that support a virtual address size of only n bits, $n < 80$, bits 0:79– n of the VSID field are treated as reserved bits, and software must set them to zeros.

A No-execute segment (N=1) contains data that should not be executed.

The L bit selects between two virtual page sizes, 4 KB (p=12) and "large". The large page size is an implementation-dependent value that is a power of 2 and is in the range 8 KB : 256 MB ($13 \le p \le 28$). Some implementations may provide a means by which software can select the large page size from a set of several implementation-dependent sizes during system initialization.

If "large page" is used in reference to real storage, it means the sequence of contiguous real (4 KB) pages to which a large virtual page is mapped.

The Class field is used in conjunction with the ***slbie*** instruction (see Section 6.1.2.1).

Software must ensure that the SLB contains at most one entry that translates a given effective address (i.e., that a given ESID is contained in no more than one SLB entry).

Programming Note

Because the virtual page size is used both in searching the Page Table and in forming the real address using the matching Page Table Entry (PTE) (see Section 4.5, "Virtual to Real Translation" on page 33), and PTEs contain no indication of the virtual page size, the virtual page size must be the same for all address translations that use a given VSID value. This has the following consequences, which apply collectively to all processors that use the same Page Table.

- The value of the L bit must be the same in all SLB entries that contain a given VSID value.
- If a given PTE is used to translate both SLS addresses and non-SLS addresses, the value of the L bit must be 0 in all SLB entries that contain the corresponding VSID value.
- Before changing the value of the L bit in an SLB entry, software must invalidate all SLB entries, TLB entries, and PTEs that contain the corresponding VSID value.

Engineering Note

It is suggested that implementations provide a mechanism by which software can select one of three different large page sizes. For example, an implementation might provide large page sizes of 64 KB, 1 MB, and 16 MB. Because this selection will be changed very infrequently (i.e., only during system initialization), the selection mechanism need not be directly accessible to software.

Architecture Note

If additional SLB entry fields are defined in the future, consideration should be given to retaining the potential to enlarge the Class field. Such enlargement would be in the low-order direction (i.e., the current Class bit would become the high-order bit of the enlarged Class field). Related considerations affect the ***slbie***, ***slbmte***, and ***slbmfev*** instructions.

Consideration should also be given to retaining the property that the Class value returned by ***slbmfev*** can be inserted into the register containing the ESID for ***slbie*** using a single instruction.

4.4.1.2 SLB Search

When the hardware searches the SLB, all entries are tested for a match with the EA. For a match to exist, the following must be true:

- $SLBE_V = 1$
- $SLBE_{ESID} = EA_{0:35}$

If the SLB search succeeds, the virtual address (VA) is formed by concatenating the VSID from the matching SLB entry with bits 36:63 of the EA.

The Virtual Page Number (VPN) is bits 0:79– p of the virtual address.

If the SLB search fails, a *segment fault* occurs. This is an Instruction Segment exception or a Data Segment exception, depending on whether the effective address is for an instruction fetch or for a data access.

4.4.2 Virtual Address Generation, SLS Address

For an SLS effective address (*tags active* mode), conversion of a 64-bit effective address to a virtual address (VA) is done by extending the effective address on the left with 16 0 bits. (The SLB is not used.)

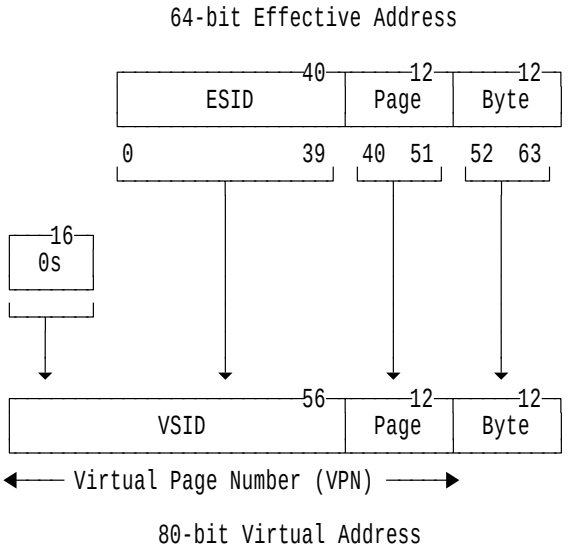


Figure 16. Translation of 64-bit effective address to 80-bit virtual address, SLS address

The virtual page size is 4 KB (p=12). The Virtual Page Number (VPN) is bits 0:67 of the virtual address.

4.5 Virtual to Real Translation

For all virtual addresses (PLS or SLS virtual address in *tags active* mode, any virtual address in *tags inactive* mode), conversion of an 80-bit virtual address to a real address is done by searching the Page Table as shown in Figure 17.

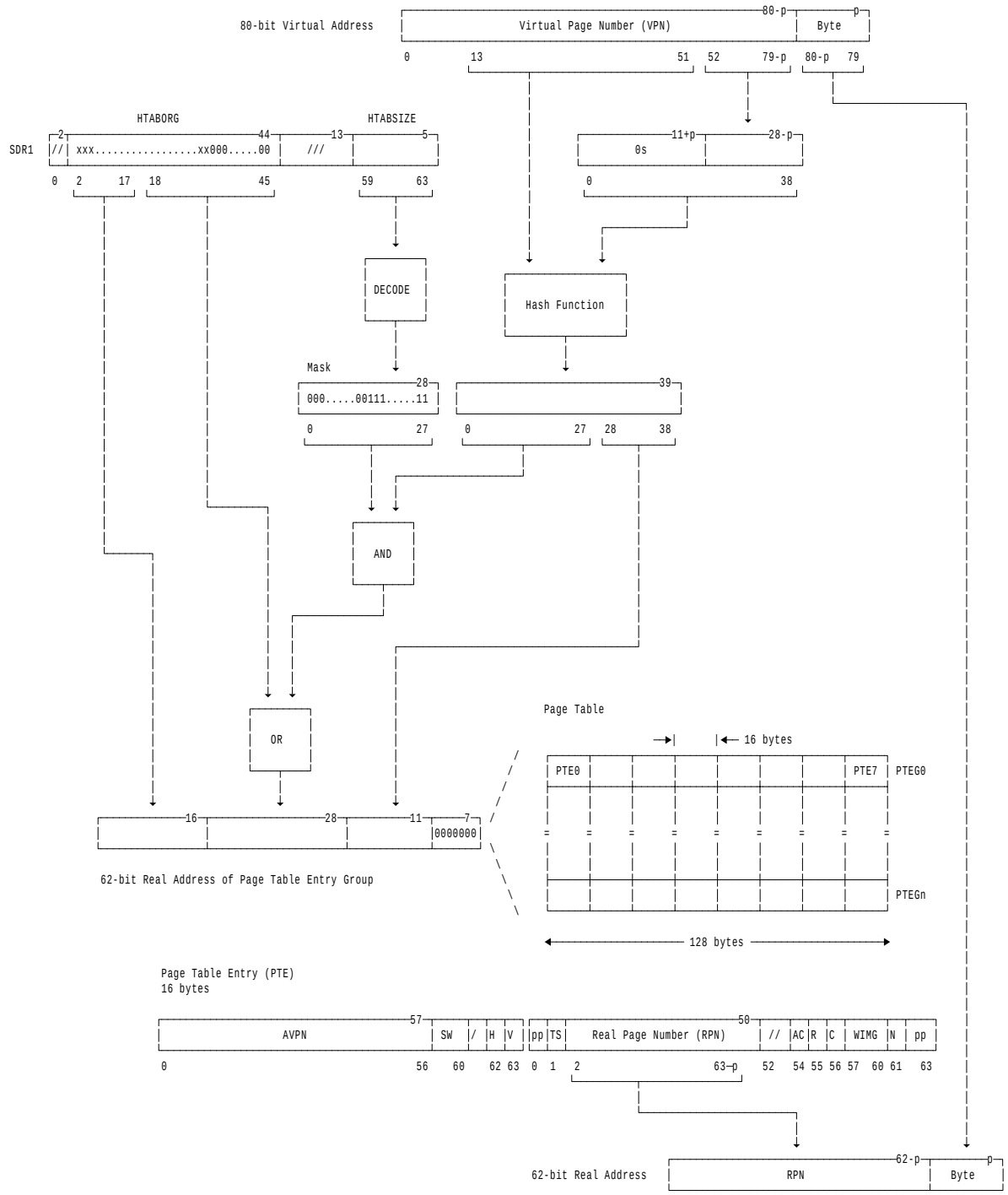


Figure 17. Translation of 80-bit virtual address to 62-bit real address

4.5.1 Page Table

The Hashed Page Table (HTAB) is a variable-sized data structure that specifies the mapping between Virtual Page Numbers and Real Page Numbers. The HTAB's size must be a multiple of 4 KB, its starting address must be a multiple of its size, and it must be located in storage having the storage control attributes that are used for implicit accesses to it (see Section 4.2.5.2).

The HTAB contains Page Table Entry Groups (PTEGs). A PTEG contains 8 Page Table Entries (PTEs) of 16 bytes each; each PTEG is thus 128 bytes long. PTEGs are entry points for searches of the Page Table.

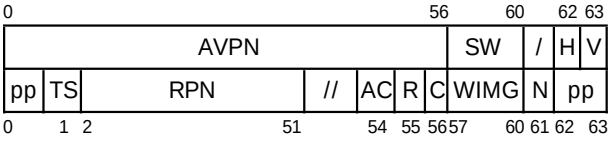
See Section 6.2, “Page Table Update Synchronization Requirements” on page 57 for the rules that software must follow when updating the Page Table.

Programming Note

The Page Table must be treated as a hypervisor resource (see Section 1.7, “Logical Partitioning (LPAR)” on page 4), and therefore must be placed in real storage to which only the hypervisor has write access. Moreover, the contents of the Page Table must be such that non-hypervisor software cannot modify storage that contains hypervisor programs or data. Finally, to protect against incorrect use of the L bit of SLB entries by non-hypervisor software, real storage that is mapped by the Page Table must be allocated to partitions in units each of which has a size that is a multiple of 2^P bytes and is aligned at a 2^P byte boundary, where 2^P is the maximum large page size for any processor in the system. (Incorrect use of the L bit could cause the virtual address for a large virtual page to be translated using a PTE that was created to translate a 4 KB virtual page. If 2^P were the maximum large page size for the partition, instead of for the system, it might be necessary to change a processor's large page size as part of reassigning the processor to a different partition.)

Page Table Entry

Each Page Table Entry (PTE) maps one VPN to one RPN. Figure 18 shows the layout of a PTE.



Dword	*Bit(s)*	*Name*	*Description*
0	0:56	AVPN	Abbreviated Virtual Page Number
	57:60	SW	Available for software use
	62	H	Hash function identifier
	63	V	Entry valid (V=1) or invalid (V=0)
1	0	pp	Page Protection bit 0
	1	TS	Tag Set bit
	2:51	RPN	Real Page Number
	54	AC	Address Compare bit
	55	R	Reference bit
	56	C	Change bit
	57:60	WIMG	Storage control bits
	61	N	No-execute page if N=1
	62:63	pp	Page protection bits 1:2

All other fields are reserved.

Figure 18. Page Table Entry

If $p \leq 23$, the Abbreviated Virtual Page Number (AVPN) field contains bits 0:56 of the VPN. Otherwise bits 0:79–p of the AVPN field contain bits 0:79–p of the VPN, and bits 80–p:56 of the AVPN field must be zeros.

Programming Note

If $p \leq 23$, the AVPN field omits the low-order 23–p bits of the VPN. These bits are not needed in the PTE, because the low-order 11 bits of the VPN are always used in selecting the PTEGs to be searched (see Section 4.5.3).

On implementations that support a virtual address size of only n bits, $n < 80$, bits 0:79–n of the AVPN field must be zeros.

The RPN field contains the page number of the real page that contains the first byte of the block of real storage to which the virtual page is mapped. If $p > 12$, the low-order p–12 bits of the RPN field (bits 64–p:51 of doubleword 1 of the PTE) must be 0. On implementations that support a real address size of only m bits, $m < 62$, bits 0:61–m of the RPN field must be zeros.

Programming Note

For a large virtual page, the high-order 62– p bits of the RPN field (bits 0:61– p) comprise the large real page number.

Engineering Note

The requirement that if p> 12 the low-order p– 12 bits of the RPN field must be 0 permits bits 34:49 of the 62-bit real address to be formed by ORing $RPN_{34:49}$ with $^{28-p}0$ || $VPN_{80-p:67}$ (equivalently, by ORing $RPN_{34:49}$ with $^{28-p}0$ || $EA_{64-p:51}$), instead of by concatenating as described in Section 4.5.3. (To protect against incorrect use of the L bit of SLB entries by non-hypervisor software, bits 34:49 of the 62-bit real address must not be formed by adding the two components.)

A No-execute page (N=1) contains data that should not be executed.

Page Table Size

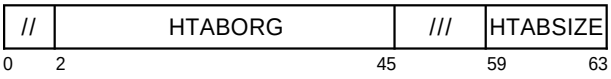
The number of entries in the Page Table directly affects performance because it influences the hit ratio in the Page Table and thus the rate of page faults. If the table is too small, it is possible that not all the virtual pages that actually have real pages assigned can be mapped via the Page Table. This can happen if too many hash collisions occur and there are more than 16 entries for the same primary/secondary pair of PTEGs. While this situation cannot be guaranteed not to occur for any size Page Table, making the Page Table larger than the minimum size (see Section 4.5.2) will reduce the frequency of occurrence of such collisions.

Programming Note

If large pages are not used, it is recommended that the number of PTEGs in the Page Table be at least half the number of real pages to be accessed. For example, if the amount of real storage to be accessed is 2^{31} bytes (2 GB), then we have $2^{31-12} = 2^{19}$ real pages. The minimum recommended Page Table size would be 2^{18} PTEGs, or 2^{25} bytes (32 MB).

4.5.2 Storage Description Register 1

The SDR1 register is shown in Figure 19.



Bits	*Name*	*Description*
2:45	HTABORG	Real address of Page Table
59:63	HTABSIZE	Encoded size of Page Table

All other fields are reserved.

Figure 19. SDR1

SDR1 is a hypervisor resource; see Section 1.7, "Logical Partitioning (LPAR)" on page 4.

The HTABORG field in SDR1 contains the high-order 44 bits of the 62-bit real address of the Page Table. The Page Table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. At least 11 bits from the hash function (see Figure 17 on page 33) are used to index into the Page Table. The minimum size Page Table is 256 KB (2^{11} PTEGs of 128 bytes each).

The Page Table can be any size 2^n bytes where $18 \le n \le 46$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0.

The HTABSIZE field in SDR1 contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the Page Table index. This number must not exceed 28. HTABSIZE is used to generate a mask of the form 0b00...011...1, which is a string of 28 – HTABSIZE 0-bits followed by a string of HTABSIZE 1-bits. The 1-bits determine which additional bits (beyond the minimum of 11) from the hash are used in the index (see Figure 17 on page 33). The number of low-order 0 bits in HTABORG must be greater than or equal to the value in HTABSIZE.

On implementations that support a real address size of only m bits, m< 62, bits 0:61– m of the HTABORG field are treated as reserved bits, and software must set them to zeros.

Programming Note

If neither PLS addresses nor *tags inactive* mode are used, let n=64. If either PLS addresses or *tags inactive* mode are used, let n equal the virtual address size (in bits) supported by the implementation. If n<67, software should set the HTABSIZE field to a value that does not exceed n–39. Because the high-order 80–n bits of the VSID are zeros (SLS address) or are assumed to be zeros (PLS address or *tags inactive* mode), the hash value used in the Page Table search will have the high-order 67–n bits either all 0s (primary hash; see Section 4.5.3) or all 1s (secondary hash). If HTABSIZE>n–39, some of these hash value bits will be used to index into the Page Table, with the result that certain PTEGs will not be searched.

Engineering Note

Because software must ensure that the number of low-order 0 bits in HTABORG is greater than or equal to the value in HTABSIZE, the 62-bit real address of the PTEG can be formed by ORing the various components.

Example:

Suppose that the Page Table is 16,384 (2^{14}) 128-byte PTEGs, for a total size of 2^{21} bytes (2 MB). A 14-bit index is required. Eleven bits are provided from the hash to start with, so 3 additional bits from the hash must be selected. Thus the value in HTABSIZE must be 3 and the value in HTABORG must have its low-order 3 bits (bits 43:45 of SDR1) equal to 0. This means that the Page Table must begin on a $2^{3+11+7} = 2^{21} =$ 2 MB boundary.

4.5.3 Page Table Search

When the hardware searches the Page Table, the accesses are performed as described in Section 4.2.5, “Real Addressing Mode” on page 27.

An outline of the HTAB search process is shown in Figure 17 on page 33. The detailed algorithm is as follows.

1. **Primary Hash:**
   A 39-bit hash value is computed by Exclusive ORing bits 13:51 of the VPN with a 39-bit value formed by concatenating 11+p 0-bits with the low-order 28–p bits of the VPN. The 62-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 2:17 of SDR1 (the high-order 16 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies a particular PTEG, called the “primary PTEG”, whose eight PTEs will be tested.
2. **Secondary Hash:**
   A 39-bit hash value is computed by taking the one's complement of the Exclusive OR of bits 13:51 of the VPN with a 39-bit value formed by concatenating 11+p 0-bits with the low-order 28–p bits of the VPN. The 62-bit real address of a PTEG is formed by concatenating the following values:
   - Bits 2:17 of SDR1 (the high-order 16 bits of HTABORG).
   - Bits 0:27 of the 39-bit hash value ANDed with the mask generated from bits 59:63 of SDR1 (HTABSIZE) and then ORed with bits 18:45 of SDR1 (the low-order 28 bits of HTABORG).
   - Bits 28:38 of the 39-bit hash value.
   - Seven 0-bits.

   This operation identifies the “secondary PTEG”.
3. As many as 16 PTEs in the two identified PTEGs are tested for a match with the VPN. Let q = minimum(5, 28–p). For a match to exist, the following must be true:
   - $PTE_H = 0$ for the primary PTEG, 1 for the secondary PTEG
   - $PTE_V = 1$
   - $PTE_{AVPN_{0:51}} = VA_{0:51}$
   - if p<28, $PTE_{AVPN_{52:51+q}} = VA_{52:51+q}$

   If one or more matches are found, the search is successful; otherwise it fails. If more than one

match is found, the matching entries must be identical in all defined fields with the exception of SW, H, AC, R, C, and TS. If they are, one of the matching entries is used, for the translation, Data Address Compare, and the setting of the R, C, and TS bits. If they are not, the translation and Data Address Compare are undefined, as is the setting of the R, C, and TS bits in the matching entries, and the remainder of this section does not apply.

If the Page Table search succeeds, the real address (RA) is formed by concatenating bits 0:61– p of the RPN from the matching PTE with bits 64– p:63 of the effective address (the byte offset).

$$RA = RPN_{0:61-p} \,||\, EA_{64-p:63}$$

For SLS addresses, the N (No-execute) value used for the storage access is the N bit of the matching PTE. For PLS addresses and *tags inactive* mode addresses, the N value used for the storage access is the result of ORing the N bit from the matching PTE with the N bit from the SLB entry that was used to translate the effective address.

† If the Page Table search fails, a *page fault* occurs. † This is an Instruction Storage exception or a Data † Storage exception, depending on whether the effective address is for an instruction fetch or for a data access.

Programming Note

To obtain the best performance, Page Table Entries should be allocated beginning with the first empty entry in the primary PTEG, or with the first empty entry in the secondary PTEG if the primary PTEG is full.

Translation Lookaside Buffer

Conceptually, the Page Table is searched by the address relocation hardware to translate every reference. For performance reasons, the hardware usually keeps a Translation Lookaside Buffer (TLB) that holds PTEs that have recently been used. The TLB is searched prior to searching the Page Table. As a consequence, when software makes changes to the Page Table it must perform the appropriate TLB invalidate operations to maintain the consistency of the † TLB with the Page Table (see Section 6.2).

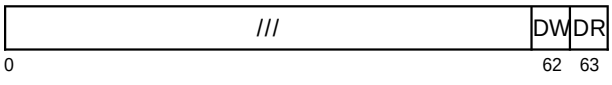
Programming Notes

1. Page Table entries may or may not be cached in a TLB.
2. It is possible that the hardware implements more than one TLB, such as one for data and one for instructions. In this case the size and shape of the TLBs may differ, as may the values contained therein.
3. Use the ***tlbie*** or ***tlbia*** instruction to ensure that the TLB no longer contains a mapping for a particular virtual page.

4.6 Data Address Compare

The Data Address Compare mechanism provides a means of detecting load and store accesses to a virtual page.

† The Data Address Compare mechanism is controlled † by the Address Compare Control Register (ACCR), and by a bit in each Page Table Entry (PTE_{AC}).



Bit	*Name*	*Description*
† 62	DW	Data Write Enable
† 63	DR	Data Read Enable

All other fields are reserved.

Figure 20. Address Compare Control Register

† A Data Address Compare match occurs for a *Load* or † *Store* instruction if, for any byte accessed,

† ■ $PTE_{AC}=1$ for the PTE that translates the virtual † address, and

† ■ the instruction is a *Store* and $ACCR_{DW}=1$, or the † instruction is a *Load* and $ACCR_{DR}=1$.

If the above conditions are satisfied, a match also occurs for ***dcbz***, ***eciwx***, and ***ecowx***. For the purpose

of determining whether a match occurs, ***eciwx*** is treated as a *Load*, and ***dcbz*** and ***ecowx*** are treated as *Stores*.

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed.
- The instruction is a *Load/Store String* of zero length.

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Compare match causes a Data Storage exception (see Section 7.5.3, "Data Storage Interrupt" on page 64). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store*, ***dcbz***, or ***ecowx*** instruction causes the match, the bytes of the storage operand that are in a virtual page with $PTE_{AC}=1$ are not altered.

Programming Note

† The Data Address Compare mechanism does not
† apply to instruction fetches, or to data accesses in real addressing mode ($MSR_{DR}=0$).

If a Data Address Compare match occurs for a *Load* instruction for which any byte of the storage operand is in storage that is both Caching Inhibited and Guarded, or for an ***eciwx*** instruction, it may not be safe for software to restart the instruction.

Engineering Note

In the case of a Data Address Compare match, it is preferable not to access any bytes of the storage operand at or after the first matching byte. This makes the Data Address Compare mechanism more useful for debugging.

4.7 Storage Control Bits

When address translation is enabled, each storage access is performed under the control of the Page Table Entry used to translate the effective address. Each Page Table Entry contains storage control bits that specify the presence or absence of the corresponding storage control attribute (see the section entitled "Storage Control Attributes" in Book II, *PowerPC AS Virtual Environment Architecture*) for all accesses translated by the entry, as shown in Figure 21. The bits are called W, I, M, and G.

Bit	Storage Control Attribute
W[1]	0 – not Write Through Required 1 – Write Through Required
I	0 – not Caching Inhibited 1 – Caching Inhibited
M[2]	0 – not Memory Coherence Required 1 – Memory Coherence Required
G	0 – not Guarded 1 – Guarded

1. Support for the 1 value of the W bit is optional. Implementations that do not support the 1 value treat the bit as reserved and assume its value to be 0.
2. Support for the 0 value of the M bit is optional. Implementations that do not support the 0 value assume the value of the bit to be 1, and may either preserve the value of the bit or write it as 1.

Figure 21. Storage control bits

Instructions are not fetched from storage for which the G bit in the Page Table Entry is set to 1 (see Section 4.9, "Storage Protection" on page 42).

Programming Note

In a uniprocessor system in which only the processor has caches, correct coherent execution does not require the processor to access storage as Memory Coherence Required, and accessing storage as not Memory Coherence Required may give better performance.

Engineering Note

Mechanisms other than processors (e.g., I/O devices) usually issue memory requests that are coherent. Such a mechanism may use the same coherence protocol that the processors use. In this case, the mechanism's use of the coherence protocol for storage that is shared with the processors may be independent of whether the processors access that storage as Memory Coherence Required.

Engineering Note

Because instruction storage need not be consistent with data storage, it is permissible for an implementation to ignore the M bit for instruction fetches.

Treating instruction fetches as noncoherent may result in better performance in an implementation in which a coherent storage request has greater latency or overhead than a noncoherent storage request. However, care must be taken to avoid using a copy of a storage location that was fetched noncoherently (in response to an instruction fetch) to satisfy a subsequent coherent data request caused by a *Load, Store*, or *Cache Management* instruction. Also, care must be taken to ensure that the instruction sequence for instruction modification that is shown in the section entitled “Instruction Cache Instruction” in Book II has the effects described there.

In system designs, consideration must be given to whether instruction fetches are to be noncoherent and, if so, how this choice affects the implementation of I/O subsystems and I/O caches. For example, if the processor ignores the M bit for instruction fetches, the system could ensure that instructions being copied into main storage have been flushed from any I/O cache before the program using them is restarted.

4.7.1 Storage Control Bit Restrictions

All combinations of W, I, M, and G values are supported except those for which both W and I are 1.

Programming Note

If an application program requests both the Write Through Required and the Caching Inhibited attributes for a given storage location, the operating system should set the I bit to 1 and the W bit to 0.

The value of the I bit must be the same for all accesses to a given real page.

The value of the W bit must be the same for all accesses to a given real page.

4.7.2 Altering the Storage Control Bits

When changing the value of the I bit for a given real page from 0 to 1, software must set the I bit to 1 and then flush all copies of locations in the page from the caches using ***dcbf*** and ***icbi*** before permitting any other accesses to the page.

When changing the value of the W bit for a given real page from 0 to 1, software must ensure that no processor modifies any location in the page until after all copies of locations in the page that are considered to be modified in the data caches have been copied to main storage using ***dcbst*** or ***dcbf***.

When changing the value of the M bit for a given real page, software must ensure that all data caches are consistent with main storage. The actions required to do this to are system-dependent.

Programming Note

For example, when changing the M bit in some directory-based systems, software may be required to execute ***dcbf*** instructions on each processor to flush all storage locations accessed with the old M value before permitting the locations to be accessed with the new M value.

Additional requirements for changing the storage control bits are given in Section 6.2.1, “Page Table Updates” on page 57 and in Chapter 9, “Synchronization Requirements for Special Registers and for Lookaside Buffers” on page 79.

4.8 Reference, Change, and Tag Set Recording

If address translation is enabled (MSR_{IR}=1 or MSR_{DR}=1), Reference (R), Change (C), and Tag Set (TS) bits are maintained in the Page Table Entry that is used to translate the virtual address. If the storage operand of a *Load* or *Store* instruction crosses a virtual page boundary, the accesses to the components of the operand in each page are treated as separate and independent accesses to each of the pages for the purpose of setting the Reference, Change, and Tag Set bits.

† Reference, Change, and Tag Set bits are set by the processor as described below. Setting the bits need not be atomic with respect to performing the access that caused the bits to be updated. An attempt to access storage may cause one or more of the bits to be set (as described below) even if the access is not performed. The bits are updated in the Page Table Entry if the new value would otherwise be different from the old, as determined by examining either the Page Table Entry or any corresponding lookaside information maintained by the processor (e.g., in a TLB).

Reference Bit

The Reference bit is set to 1 if the corresponding access (load, store, or instruction fetch) is required by the sequential execution model and is performed. Otherwise the Reference bit may be set to 1 if the corresponding access is attempted, either in-order or out-of-order, even if the attempt causes an exception.

Change Bit

The Change bit is set to 1 if a *Store* instruction is executed and the store is performed. Otherwise the Change bit may be set to 1 if a *Store* instruction is executed and the store is permitted by the storage protection mechanism and would not cause an EAO exception and, if the *Store* instruction is executed out-of-order, the instruction would be required by the sequential execution model in the absence of the following kinds of interrupts:

- system-caused interrupts (i.e., System Reset, Machine Check, External, and Decrementer interrupts)
- Floating-Point Enabled Exception type Program interrupts when the processor is in an Imprecise mode

Programming Note

Even though the execution of a *Store* instruction causes the Change bit to be set to 1, the store might not be performed or might be only partially performed in cases such as the following.

- A *Store Conditional* instruction (***stwcx.*** or ***stdcx.***) is executed, but no store is performed.
- A *Store String Word Indexed* instruction (***stswx***) or *Store String Doubleword Indexed* instruction (***stsdx***) is executed, but the length is zero.
- The *Store* instruction causes a Data Storage exception (for which setting the Change bit is not prohibited).
- The *Store* instruction causes an Alignment exception.
- The Page Table Entry that translates the virtual address of the storage operand is altered such that the new contents of the Page Table Entry preclude performing the store (e.g., the PTE is made invalid, or the PP bits are changed).

  For example, when executing a *Store* instruction, the processor may search the Page Table for the purpose of setting the Change bit and then reexecute the instruction. When reexecuting the instruction, the processor may search the Page Table a second time. If the Page Table Entry has meanwhile been altered, by a program executing on another processor, the second search may obtain the new contents, which may preclude the store.
- A system-caused interrupt occurs before the store has been performed.

Tag Set Bit

There are two implementation alternatives for this bit.

† 1. The Tag Set bit is not altered by the processor.

2. If a ***stq*** instruction is executed when XER_{43}=1, the Tag Set bit is or may be set to 1 under the same conditions as those in which the Change bit is or may be set to 1.

Figure 22 on page 41 summarizes the rules for setting the Reference, Change, and Tag Set bits. The table applies to each atomic storage reference. It should be read from the top down; the first line matching a given situation applies. For example, if ***stwcx.*** fails due to both a storage protection violation and the lack of a reservation, the Change bit is not altered.

In the figure, the "*Load*-type" instructions are the *Load* instructions described in Books I and II, ***eciwx***, and the *Cache Management* instructions that are treated as *Loads*. The "*Store*-type" instructions are the *Store* instructions described in Books I and II, ***ecowx***, and the *Cache Management* instructions that are treated as *Stores*. The "ordinary" *Load* and *Store* instructions are those described in Books I and II. "set" means "set to 1".

Status of Access	R	C	TS
Effective Address Overflow exception	Acc[1]	No	No
Storage protection violation	Acc[1]	No	No
Out-of-order I-fetch or *Load*-type inst'n	Acc	No	No
Out-of-order *Store*-type inst'n			
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3]	Acc	Acc[1 2]	Acc[1 4 6]
All other cases	Acc	No	No
In-order *Load*-type or *Store*-type inst'n, access not performed			
Load-type inst'n	Acc	No	No
Store-type inst'n	Acc	Acc[2]	Acc[4 6]
Other in-order access			
I-fetch	Yes	No	No
Ordinary *Load*, ***eciwx***	Yes	No	No
stq	Yes	Yes	Yes[5]
Other ordinary *Store*, ***ecowx***, ***dcbz***	Yes	Yes	No
icbi, dcbt, dcbtst, dcbst, dcbf	Acc	No	No

"Acc" means that it is acceptable to set the bit.

[1] It is preferable not to set the bit.

[2] If C is set, R is also set unless it is already set.

[3] For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.

[4] If TS is set, R and C are also set unless they are already set.

[5] TS is set only if $XER_{43}=1$.

[6] TS may be set only if the instruction is ***stq*** and $XER_{43}=1$.

Figure 22. Setting the Reference, Change, and Tag Set bits

Engineering Note

Any implementation-specific interrupt used to emulate instructions in software can be handled in a manner similar to a system-caused interrupt. That is, if the hardware can determine that the instruction to be emulated will not cause a precise architected interrupt then the Change and Tag Set bits can be set out-of-order past the instruction to be emulated under the same conditions as these bits can be set past a potential system-caused interrupt.

When the hardware updates the Reference, Change, and Tag Set bits in the Page Table Entry, the accesses are performed as described in Section 4.2.5, "Real Addressing Mode" on page 27. The accesses may be performed using operations equivalent to a store to a byte, halfword, word, or doubleword, and are not necessarily performed as an atomic read/modify/write of the affected bytes.

These Reference, Change, and Tag Set bit updates are not necessarily immediately visible to software. Executing a ***sync*** instruction ensures that all Reference, Change, and Tag Set bit updates associated

with address translations that were performed, by the processor executing the ***sync*** instruction, before the ***sync*** instruction is executed will be performed with respect to that processor before the ***sync*** instruction's memory barrier is created. There are additional requirements for synchronizing Reference, Change, and Tag Set bit updates in multiprocessor systems; see Section 6.2.1, “Page Table Updates” on page 57.

Programming Note

Because the ***sync*** instruction is execution synchronizing, the set of Reference, Change, and Tag set bit updates that are performed with respect to the processor executing the ***sync*** instruction before the memory barrier is created includes all Reference, Change, and Tag Set bit updates associated with instructions preceding the ***sync*** instruction.

If software refers to a Page Table Entry when $MSR_{DR}=1$, Reference, Change, and Tag Set bits in the associated Page Table Entries are set as for ordinary loads and stores. See Section 6.2.1 for the rules software must follow when updating Reference, Change, and Tag Set bits.

Engineering Note

If the hardware updates a Reference, Change, or Tag Set bit in the Page Table Entry without using an atomic read/modify/write operation, care must be taken to avoid overwriting an update to any of these bits by another processor. Thus the datum written to the Page Table Entry must not contain a 0 value for any of these bits.

Subject to the preceding requirement, when the hardware updates the Reference, Change, or Tag Set bit in the Page Table Entry it is permissible to store the corresponding byte, halfword, word, or doubleword, with the relevant subset of these three bits updated, from any lookaside information (e.g., TLB) maintained by the processor.

Engineering Note

Since most TLB reloads do not require altering the Reference, Change, or Tag Set bit in the Page Table Entry (PTE), it is suggested that on a TLB miss the search for the PTE be done without fetching the PTEs for exclusive access. This will reduce cache thrashing due to TLB reloads. It is assumed that a nonexclusive request for a PTE will be returned with exclusive access if no other processor has a copy.

4.9 Storage Protection

The storage protection mechanism provides a means for selectively granting instruction fetch access, granting read access, granting read/write access, and prohibiting access to areas of storage based on a number of control criteria.

The operation of the protection mechanism depends on whether address translation is enabled ($MSR_{IR}=1$ or $MSR_{DR}=1$, as appropriate for the access) or disabled ($MSR_{IR}=0$ or $MSR_{DR}=0$, as appropriate for the access) and, if address translation is enabled, on whether the processor is in *tags active* mode or *tags inactive* mode.

If an instruction fetch is not permitted by the protection mechanism, an Instruction Storage exception is generated. If a data access is not permitted by the protection mechanism, a Data Storage exception is generated. (See Section 4.2.1, “Storage Exceptions” on page 24.)

When address translation is enabled, a *protection domain* is a range of unmapped effective addresses, a virtual page, or a segment that is not an SLS segment. When address translation is disabled and LPES=1 there are two protection domains: the set of effective addresses that are less than the value specified by the RMLR, and all other effective addresses. When address translation is disabled and LPES=0 the entire effective address space comprises a single protection domain. A *protection boundary* is a boundary between protection domains.

4.9.1 Storage Protection, Address Translation Enabled, Tags Active

When address translation is enabled and the processor is in *tags active* mode, the protection mechanism is controlled by the following.

- MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state
- MSR_{US}, which distinguishes between system state and user state
- PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address
- For instruction fetches only:
  - the N (No-execute) value used for the access (see Section 4.5.3)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if $PTE_G=1$.

2. For any access except an instruction fetch that is not permitted by rule 1, Figure 23 is applied. An instruction fetch is permitted for any entry in the figure except “no access”; it is implementation-dependent whether an instruction fetch is permitted for an entry with “no access”. A load is permitted for any entry except “no access”. A store is permitted only for entries with “read/write”.

PR=1 & US=1	PR=1 & US=0	PR=0 & US=1	PR=0 & US=0	PP
no access	read/write	no access	read/write	000
read only	read/write	read only	read/write	001
read/write	read/write	read/write	read/write	010
read only	read only	read only	read only	011
no access	no access	read/write	read/write	100
read only	read only	read/write	read/write	101

All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.

Figure 23. PP bit protection states, address translation enabled, tags active

4.9.2 Storage Protection, Address Translation Enabled, Tags Inactive

When address translation is enabled and the processor is in *tags inactive* mode, the protection mechanism is controlled by the following.

- MSR_{PR}, which distinguishes between supervisor (privileged) state and problem state
- K_s and K_p, the supervisor (privileged) state and problem state storage key bits in the SLB entry used to translate the effective address
- PP, page protection bits 0:2 in the Page Table Entry used to translate the effective address
- For instruction fetches only:
  - the N (No-execute) value used for the access (see Section 4.5.3)
  - PTE_G, the G (Guarded) bit in the Page Table Entry used to translate the effective address

Using the above values, the following rules are applied.

1. For an instruction fetch, the access is not permitted if the N value is 1 or if PTE_G=1.
2. For any access except an instruction fetch that is not permitted by rule 1, a “Key” value is computed using the following formula:

$$Key \leftarrow (K_p \ \& \ MSR_{PR}) \ | \ (K_s \ \& \ \neg MSR_{PR})$$

   Using the computed Key, Figure 24 is applied. An instruction fetch is permitted for any entry in the figure except “no access”. A load is permitted for any entry except “no access”. A store is permitted only for entries with “read/write”.

Key	PP	Access Authority
0	- 00	read/write
0	- 01	read/write
0	010	read/write
0	011	read only
1	- 00	no access
1	- 01	read only
1	010	read/write
1	011	read only

- PP_0 may be 0 or 1.

All PP encodings not shown above are reserved. The results of using reserved PP encodings are boundedly undefined.

Figure 24. PP bit protection states, address translation enabled, tags inactive

4.9.3 Storage Protection, Address Translation Disabled

When address translation is disabled, the protection mechanism is controlled by the following (see Section 1.7, “Logical Partitioning (LPAR)” on page 4 and Section 4.2.5, “Real Addressing Mode” on page 27).

- LPES, which distinguishes between the two modes of using the LPAR facility
- MSR_{HV}, which distinguishes between hypervisor state and other privilege states
- RMLR, which specifies the real mode limit value

Using the above values, Figure 25 is applied. The access is permitted for any entry in the figure except “no access”.

LPES	HV	Access Authority
0	0	no access
0	1	read/write
1	0	read/write or no access[1]
1	1	read/write

1. If the effective address for the access is less than the value specified by the RMLR the access authority is read/write; otherwise the access is not permitted.

Figure 25. Protection states, address translation disabled

Programming Note

The comparison described in note 1 in Figure 25 ignores bits 0:1 of the effective address and may ignore bits 2:63– m; see Section 4.2.5.

Chapter 5. Storage Control, Tags Inactive

5.1 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Load*, *Store*, *Branch*, or *Cache Management* instruction, or when it fetches the next sequential instruction. The effective address is translated to a real address according to procedures described in Section 5.3, “Address Translation Overview” on page 47 and following sections. The real address is what is presented to the storage subsystem. See Figure 26 on page 47.

For a complete discussion of storage addressing and effective address calculation, see the section entitled “Storage Addressing” in Book I, *PowerPC AS User Instruction Set Architecture*.

† Tags Active vs. Tags Inactive

| The selection between *tags active* and *tags inactive* | operation is made by MSR_{TA}. This chapter describes | storage control in *tags inactive* mode.

Storage Control Overview

| ■ Real address space size is 2^m bytes, $m \leq 62$; see | Note 1.

| ■ Real page size is 2^{12} bytes (4 KB).

■ Effective address space size is 2^{64} bytes.

| ■ Virtual address space size is 2^n bytes, $65 \leq n \leq 80$; | see Note 2.

■ Segment size is 2^{28} bytes (256 MB).

| ■ Number of virtual segments is 2^{n-28}; see Note 2.

| ■ Virtual page size is 2^p bytes, $12 \leq p \leq 28$; two sizes | are supported simultaneously, 4 KB (p=12) and a | larger size; see Note 3.

| **Notes:**

| 1. The value of m is implementation-dependent | (subject to the maximum given above). When | used to address storage, the high-order 62–m | bits of the “62-bit” real address must be zeros.
| 2. The value of n is implementation-dependent | (subject to the range given above). In references | to 80-bit virtual addresses elsewhere in this Book, | the high-order 80–n bits of the “80-bit” virtual | address are assumed to be zeros.
| 3. The value of p for the larger virtual page size is | implementation-dependent (subject to the range | given above).

5.2 Storage Model

The storage model provides the following features.

1. The architecture allows the storage implementations to take advantage of the performance benefits of weak ordering of storage accesses between processors or between processors and I/O devices.
2. The architecture provides instructions that allow the programmer to ensure a consistent and ordered storage state.

   - ***dcbf***
   - ***dcbst***
   - ***eieio***
   - ***icbi***
   - ***isync***
   - ***ldarx***
   - ***lwarx***
   - ***lwsync***
   - ***stdcx.***
   - ***stwcx.***
   - ***sync***
   - ***tlbsync***

3. Storage accesses appear to be performed in program order with respect to the processor performing them but, in general, may be performed in different orders with respect to other processors and mechanisms.
4. Storage consistency between processors, and between a processor and an I/O device, is controlled by software using the "WIM" storage control bits (see Section 4.7). These bits allow software to control whether a given storage location has any of the following attributes.

   - Write Through Required (W)
   - Caching Inhibited (I)
   - Memory Coherence Required (M)

Engineering Note

The architecture does not suggest or preclude any implementation of storage consistency supporting the features listed above. In particular, the implementation may be a snoopy bus design, a centralized cache directory design, or other design.

5.2.1 Storage Exceptions

A *storage exception* is an exception that causes an Instruction Storage interrupt, an Instruction Segment interrupt, a Data Storage interrupt, a Data Segment interrupt, or an Alignment interrupt. Attempting to fetch or execute an instruction causes a storage exception if certain conditions apply. Such conditions include the following.

- The appropriate relocate bit in the MSR is set to 1 and the effective address cannot be translated to a real address.
- The access is not permitted by the storage protection mechanism.
- The access causes a Data Address Compare match or a Data Address Breakpoint match.

In certain cases a storage exception may result in the "restart" of (re-execution of at least part of) a *Load* or *Store* instruction. See the section entitled "Instruction Restart" in Book II, *PowerPC AS Virtual Environment Architecture*, and Section 7.6, "Partially Executed Instructions" on page 72 in this Book.

5.2.2 Instruction Fetch

Instruction fetch, for both *tags active* mode and *tags inactive* mode, is described in Section 4.2.2.

5.2.3 Data Access

Data access, for both *tags active* mode and *tags inactive* mode, is described in Section 4.2.3.

5.2.4 Performing Operations Out-of-Order

The limits on performing operations out-of-order, for both *tags active* mode and *tags inactive* mode, are described in Section 4.2.4.

5.2.5 32-Bit Mode

The computation of the 64-bit effective address is independent of mode. In 32-bit mode (MSR_{SF}=0), the high-order 32 bits of the 64-bit effective address are treated as zeros for the purpose of addressing storage. This applies to both data accesses and instruction fetches. It applies independent of whether address translation is enabled or disabled. This truncation of the effective address is the only respect in which storage accesses are mode-dependent.

Programming Note

Treating the high-order 32 bits of the effective address as zeros effectively truncates the 64-bit effective address to a 32-bit effective address such as would have been generated on a 32-bit implementation of the PowerPC Architecture. Thus, for example, the ESID in 32-bit mode is the high-order four bits of this truncated effective address; the ESID thus lies in the range 0-15. When address translation is enabled, these four bits would select a Segment Register on a 32-bit implementation of the PowerPC Architecture. On PowerPC AS the SLB entries that translate these 16 ESIDs can be used to emulate the PowerPC 32-bit implementation's Segment Registers.

5.2.6 Real Addressing Mode

Real addressing mode, for both *tags active* mode and *tags inactive* mode, is described in Section 4.2.5.

5.2.7 Real Storage Locations Having Defined Uses

The defined uses of real storage, for both *tags active* mode and *tags inactive* mode, are described in Section 4.2.6.

5.2.8 Invalid Real Address

The results of attempting to access an invalid real address, for both *tags active* mode and *tags inactive* mode, are described in Section 4.2.7.

5.3 Address Translation Overview

Figure 26 gives an overview of the address translation process in *tags inactive* mode.

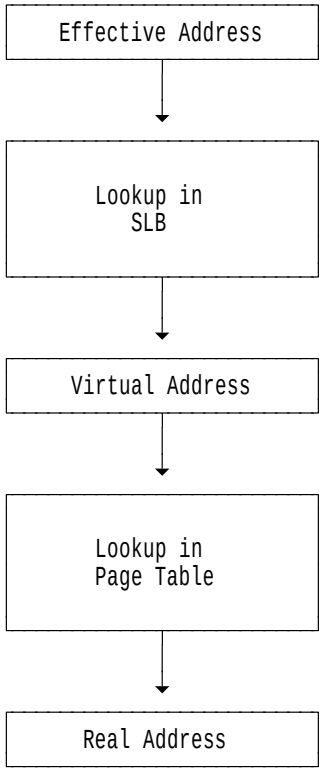


Figure 26. PowerPC AS address translation, tags inactive

The effective address (EA) is the address generated by the processor for an instruction fetch or for a data access. If address translation is enabled (MSR_{IR}= 1 or MSR_{DR}= 1 as appropriate), this address is passed to the Address Translation mechanism, which attempts to convert the address to a real address which is then used to access storage.

The first step in address translation is to convert the effective address to a virtual address (VA), as described in Section 4.4.1, “Virtual Address Generation, Tags Inactive Mode or PLS Address” on page 30. The second step, conversion of the virtual address to a real address (RA), is described in Section 4.5.

If the effective address cannot be translated, a storage exception (see Section 4.2.1) occurs.

5.4 Data Address Compare

The Data Address Compare mechanism, for both *tags active* mode and *tags inactive* mode, is described in Section 4.6.

5.5 Storage Control Bits

The storage control bits, for both *tags active* mode and *tags inactive* mode, are described in Section 4.7.

5.6 Reference and Change Recording

Reference and Change recording, for both *tags active* mode and *tags inactive* mode, is described in Section 4.8.

5.7 Storage Protection

Storage protection, for both *tags active* mode and *tags inactive* mode, is described in Section 4.9.

Chapter 6. Storage Control Instructions and Table Updates

6.1 Storage Control Instructions

6.1.1 Cache Management Instructions

This section describes aspects of cache management that are relevant only to operating systems.

For a ***dcbz*** instruction that causes the target block to be newly established in the data cache without being fetched from main storage, the processor need not verify that the associated real address is valid. The existence of a data cache block that is associated with an invalid real address (see Section 4.2.7) can cause a delayed Machine Check interrupt or a delayed Checkstop.

Each implementation provides an efficient means by which software can ensure that all blocks that are considered to be modified in the data cache have been copied to main storage before the processor enters any power conserving mode in which data cache contents are not maintained. The means are described in the Book IV, *PowerPC AS Implementation Features* document for the implementation.

6.1.2 Lookaside Buffer Management

All implementations have a Segment Lookaside Buffer (SLB), and provide the *SLB Management* instructions described in Section 6.1.2.1.

For performance reasons, most implementations have a Translation Lookaside Buffer (TLB), which is a cache of recently used Page Table Entries (PTEs). The TLB is not necessarily kept consistent with the Page Table in main storage. When software alters the contents of a PTE, it must also invalidate all corresponding TLB entries.

Each implementation that has a TLB provides a means by which software can do the following.

- Invalidate the TLB entry that translates a given effective address
- Invalidate all TLB entries

An implementation may provide one or more of the *TLB Management* instructions described in Section 6.1.2.2 in order to satisfy requirements in the preceding list. Alternatively, an algorithm may be given that performs one of the functions listed above (a † loop invalidating individual TLB entries may be used † to invalidate the entire TLB, for example), or different instructions may be provided. Such algorithms or instructions are described in Book IV, *PowerPC AS Implementation Features*. Because most implementations have a TLB and also provide instructions similar or identical to the *TLB Management* instructions described in Section 6.1.2.2, other sections of the Books assume that the TLB exists and that the instructions described in Section 6.1.2.2 are provided.

An implementation that does not have a TLB treats the corresponding instructions (***tlbie***, ***tlbia***, and ***tlbsync***) either as no-ops or as illegal instructions.

Programming Note

Because the presence, absence, and exact semantics of the *TLB Management* instructions are implementation-dependent, it is recommended that system software "encapsulate" uses of these instructions into subroutines to minimize the impact of moving from one implementation to another.

Programming Note

The function of all the instructions described in Sections 6.1.2.1 and 6.1.2.2 is independent of whether address translation is enabled or disabled.

For a discussion of software synchronization requirements when invalidating SLB and TLB entries, see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79.

Engineering Note

It is possible for the hardware to implement more than one TLB, such as one for data and one for instructions. If this approach is taken, the requirement for an instruction that invalidates a TLB entry may be satisfied by a single instruction for all TLBs or by separate instructions for each TLB.

Engineering Note

Primary opcode 31, extended opcode 308, can be used for a privileged implementation-specific TLB invalidation function.

Primary opcode 31, extended opcodes 978 and 1010, can be used for a privileged implementation-specific TLB reload function for data and instructions respectively.

6.1.2.1 SLB Management Instructions

Programming Note

Accesses to a given SLB entry caused by the instructions described in this section obey the sequential execution model with respect to the contents of the entry and with respect to data dependencies on those contents. That is, if an instruction sequence contains two or more of these instructions, when the sequence has completed, the final state of the SLB entry and of General Purpose Registers is as if the instructions had been executed in program order.

However, software synchronization is required in order to ensure that any alterations of the entry take effect correctly with respect to address translation; see Chapter 9.

SLB Invalidate Entry X-form

slbie RB

31	///	///	RB	434	/
0	6	11	16	21	31

```
esid ← (RB)0:35
class ← (RB)36
if class = SLBEC for SLB entry that translates
   or most recently translated esid
  then for SLB entry (if any) that translates esid
    SLBEV ← 0
    all other fields of SLBE ← undefined
  else translation of esid ← undefined
```

Let the Effective Segment ID (ESID) be $(RB)_{0:35}$. Let the class be $(RB)_{36}$. The class value must be the same as the Class value in the SLB entry that translates the ESID, or the Class value that was in the SLB entry that most recently translated the ESID if the translation is no longer in the SLB; if the class value is not the same, the results of translating effective addresses for which $EA_{0:35}$=ESID are undefined, and the next paragraph need not apply.

If the SLB contains an entry that translates the specified ESID, the V bit in that entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values.

$(RB)_{37:63}$ must be zeroes.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction is privileged.

Special Registers Altered:

None

Programming Note

The only SLB entry that is invalidated is the entry (if any) that translates the specified ESID.

slbie does not affect SLBs on other processors.

Programming Note

The reason the class value specified by ***slbie*** must be the same as the Class value that is or was in the relevant SLB entry is that the processor may use these values to optimize invalidation of implementation-specific lookaside information used in address translation. If the value specified by ***slbie*** differs from the value that is or was in the relevant SLB entry, these optimizations may produce incorrect results. (An example of implementation-specific address translation lookaside information is the set of recently used translations of effective addresses to real addresses that some processors maintain in an Effective to Real Address Translation (ERAT) lookaside buffer.)

The recommended use of the Class field is to classify SLB entries according to the expected longevity of the translations they contain, or a similar property such as whether the translations are used by all programs or only by a single program. If this is done and the processor invalidates certain implementation-specific lookaside information based only on the specified class value, an ***slbie*** instruction that invalidates a short-lived translation will preserve such lookaside information for long-lived translations.

If the optional “Bridge” facility is implemented (see Section 11.1), the *Move To Segment Register* instructions create SLB entries in which the Class value is 0.

Engineering Note

$(RB)_{37:63}$ must be ignored by the processor.

Preserving the contents of the SLB entry (other than the V bit) when an ***slbie*** instruction is executed, and returning the contents of the SLB entry when an ***slbmfev*** or ***slbmfee*** instruction is executed that specifies an invalid SLB entry, facilitates the debugging of software.

Engineering Note

An example of how the class value can be used to optimize invalidation of implementation-specific address translation lookaside information is as follows. (The class value has no architecturally defined use, nor does the Class field of SLB entries.)

On implementations that have an Effective to Real Address Translation lookaside buffer (ERAT), the class value can be used to select the ERAT entries to invalidate when an ***slbie*** instruction is executed. (Invalidating only ERAT entries in which the Class value is equal to the specified class value is likely to provide better performance than invalidating all ERAT entries.) If ERAT entries are used to translate effective addresses in real addressing mode, those entries can be treated as if they contain a Class value that lies outside the range supported by the SLB entry, so that ***slbie*** does not invalidate them.

Architecture Note

† Bits 11:15 of the ***slbie*** instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using † (RA|0)+(RB) address arithmetic for ***slbie***.

The requirement that $RB_{37:63}$ contain zeros and be ignored by the processor permits the Class field of the SLB entry and the class value supplied by ***slbie*** to be enlarged in the future if that proves desirable.

The requirement that $RB_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***slbie***.

SLB Invalidate All X-form

slbia

31	///	///	///	498	/
0	6	11	16	21	31

```
for each SLB entry except SLB entry 0
  SLBE_V ← 0
  all other fields of SLBE ← undefined
```

For all SLB entries except SLB entry 0, the V bit in the entry is set to 0, making the entry invalid, and the remaining fields of the entry are set to undefined values. SLB entry 0 is not altered.

This instruction is privileged.

Special Registers Altered:

None

Programming Note

slbia does not affect SLBs on other processors.

Programming Note

If ***slbia*** is executed when instruction address translation is enabled (MSR_{IR}=1), software can ensure that attempting to fetch the instruction following the ***slbia*** does not cause an Instruction Segment interrupt by placing the ***slbia*** and the subsequent instruction in the effective segment mapped by SLB entry 0. (The preceding assumes that no other interrupts occur between executing the ***slbia*** and executing the subsequent instruction.)

SLB Move To Entry X-form

slbmte RS,RB

31	RS	///	RB	402	/
0	6	11	16	21	31

The SLB entry specified by bits 52:63 of register RB is loaded from register RS and from the remainder of register RB. The contents of these registers are interpreted as shown in Figure 27.

RS

VSID	K_sK_pNLC	0s
0	52 56	63

RB

ESID	V	0s	index
0	35 37	52	63

$RS_{0:51}$ VSID
RS_{52} K_s
RS_{53} K_p
RS_{54} N
RS_{55} L
RS_{56} C
$RS_{57:63}$ must be 0b000_0000

$RB_{0:35}$ ESID
RB_{36} V
$RB_{37:51}$ must be 0b000 || 0x000
$RB_{52:63}$ index, which selects the SLB entry

Figure 27. GPR contents for slbmte

On implementations that support a virtual address size of only n bits, $n < 80$, $(RS)_{0:79-n}$ must be zeros.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

If this instruction is executed in 32-bit mode, $(RB)_{0:31}$ must be zeros (i.e., the ESID must be in the range 0-15).

This instruction cannot be used to invalidate an SLB entry.

This instruction is privileged.

Special Registers Altered:
None

Programming Note

The reason ***slbmte*** cannot be used to invalidate an SLB entry is that it does not necessarily affect implementation-specific address translation look-aside information. ***slbie*** (or ***slbia***) must be used for this purpose.

Engineering Note

$(RS)_{57:63}$ must be ignored by the processor.

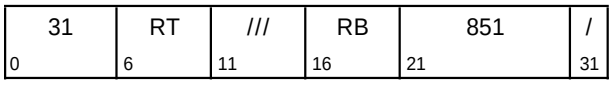
Architecture Note

The requirement that $RS_{57:63}$ contain zeros and be ignored by the processor permits the Class field of the SLB entry to be enlarged in the future if that proves desirable.

The requirement that $RB_{0:31}$ contain zeros in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***slbmte***.

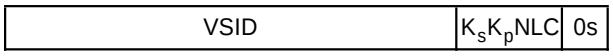
SLB Move From Entry VSID X-form

slbmfev RT,RB

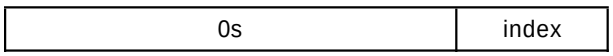


If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the VSID, K_s, K_p, N, L, and C fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 28.

RT



RB



$RT_{0:51}$ VSID
RT_{52} K_s
RT_{53} K_p
RT_{54} N
RT_{55} L
RT_{56} C
$RT_{57:63}$ set to 0b000_0000

$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 28. GPR contents for slbmfev

On implementations that support a virtual address size of only n bits, $n < 80$, $RT_{0:79-n}$ are set to zeros.

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), the contents of register RT are undefined.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

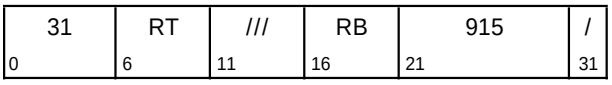
Special Registers Altered:
None

Architecture Note

The requirement that $RT_{57:63}$ be set to zeros permits the Class field of the SLB entry to be enlarged in the future if that proves desirable.

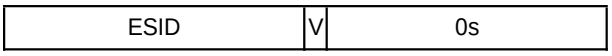
SLB Move From Entry ESID X-form

slbmfee RT,RB

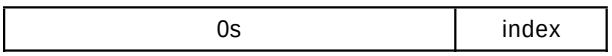


If the SLB entry specified by bits 52:63 of register RB is valid (V=1), the contents of the ESID and V fields of the entry are placed into register RT. The contents of these registers are interpreted as shown in Figure 29.

RT



RB



$RT_{0:35}$ ESID
RT_{36} V
$RT_{37:63}$ set to 0b000 || 0x00_0000

$RB_{0:51}$ must be 0x0_0000_0000_0000
$RB_{52:63}$ index, which selects the SLB entry

Figure 29. GPR contents for slbmfee

If the SLB entry specified by bits 52:63 of register RB is invalid (V=0), RT_{36} is set to 0 and the contents of $RT_{0:35}$ and $RT_{37:63}$ are undefined.

High-order bits of $(RB)_{52:63}$ that correspond to SLB entries beyond the size of the SLB provided by the implementation must be zeros.

This instruction is privileged.

Special Registers Altered:
None

6.1.2.2 TLB Management Instructions (Optional)

TLB Invalidate Entry X-form

tlbie RB,L

[POWER mnemonic: tlbi]

31	///	L	///	RB	306	/
0	6	10	11	16	21	31

```
if L = 0
  then pg_size ← 4 KB
  else pg_size ← large page size
p ← log_base_2(pg_size)
if (RB)0:15 = 0x0000
  then seg_type = non-SLS
  else seg_type = SLS
for each TLB entry
  if (entry_VPN32:79–p = (RB)16:63–p) &
     (entry_pg_size = pg_size) &
     (entry_seg_type = seg_type)
  then TLB entry ← invalid
```

If $(RB)_{0:15}$=0x0000 let the segment type be non-SLS (PLS segment or *tags inactive* mode segment); otherwise let the segment type be SLS. If the L field of the instruction is 1 let the page size be large; otherwise let the page size be 4 KB.

All TLB entries that have all of the following properties are made invalid on all processors.

- The entry translates a virtual address for which $VPN_{32:79-p}$ is equal to $(RB)_{16:63-p}$.
- The segment type of the entry matches the segment type specified by $(RB)_{0:15}$.
- The page size of the entry matches the page size specified by the L field of the instruction.

Additional TLB entries may also be made invalid on any processor.

MSR_{SF} must be 1 when this instruction is executed; otherwise the results are undefined.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync***) instruction with respect to a subsequent ***tlbsync*** instruction executed by the processor executing the ***tlbie*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a third set of operations, which is independent of the other two sets that ***eieio*** orders.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

See Section 6.2.1, “Page Table Updates” on page 57 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:

None

Programming Note

If the same VPN is used for both an SLS segment and a non-SLS segment, two ***tlbie*** instructions must be executed in order to invalidate the translation, one with $(RB)_{0:15}$=0x0000 and one with $(RB)_{0:15} \neq$ 0x0000.

Architecture Note

† Bits 11:15 of the ***tlbie*** instruction (ordinarily the position of an RA field) must be zero. This provides implementations the option of using † (RA|0)+(RB) address arithmetic for ***tlbie***.

The requirement that ***tlbie*** be executed only in 64-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***tlbie***. (If ***tlbie*** were executed in 32-bit mode, on an implementation that does normal EA computation for ***tlbie*** the high-order 16 bits of the specified VPN bits would be treated as zeros.)

The requirement that ***tlbie*** (and ***tlbsync***) be executed only in hypervisor state reduces implementation complexity, by avoiding the need to ensure that violation of the requirements described in Section 6.2.1 by non-hypervisor software does not cause a Checkstop or other significant system-wide event.

Architecture Note

Cumulative ordering is moot for the memory barrier created by ***eieio*** for ***tlbie*** and ***tlbsync***, because at most one processor should execute these instructions at a time (see Section 6.2.1).

Engineering Note

Causing a Privileged Instruction type Program interrupt if ***tlbie*** or ***tlbsync*** is executed in privileged but non-hypervisor state facilitates the debugging of software.

TLB Invalidate All X-form

tlbia

31	///	///	///	370	/
0	6	11	16	21	31

`all TLB entries ← invalid`

† All TLB entries are made invalid on the processor
† executing the ***tlbia*** instruction.

†

This instruction is privileged.

This instruction is optional.

Special Registers Altered:
None

†

Programming Note

† ***tlbia*** does not affect TLBs on other processors.

TLB Synchronize X-form

tlbsync

31	///	///	///	566	/
0	6	11	16	21	31

The ***tlbsync*** instruction provides an ordering function for the effects of all ***tlbie*** instructions executed by the processor executing the ***tlbsync*** instruction, with respect to the memory barrier created by a subsequent ***sync*** instruction executed by the same processor. Executing a ***tlbsync*** instruction ensures that all of the following will occur.

- All TLB invalidations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction will have completed on any other processor before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that processor.
- All storage accesses by other processors for which the address was translated using the translations being invalidated, and all Reference, Change, and Tag Set bit updates associated with address translations that were performed by other processors using the translations being invalidated, will have been performed with respect to the processor executing the ***sync*** instruction, to the extent required by the associated Memory Coherence Required attributes, before the ***sync*** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the ***eieio*** (or ***sync***) instruction with respect to preceding ***tlbie*** instructions executed by the processor executing the ***tlbsync*** instruction. The operations caused by ***tlbie*** and ***tlbsync*** are ordered by ***eieio*** as a third set of operations, which is independent of the other two sets that ***eieio*** orders.

The ***tlbsync*** instruction may complete before operations caused by ***tlbie*** instructions preceding the ***tlbsync*** instruction have been performed.

This instruction is privileged, and can be executed only in hypervisor state. If it is executed in privileged but non-hypervisor state either a Privileged Instruction type Program interrupt occurs or the results are boundedly undefined.

This instruction is optional.

See Section 6.2.1, “Page Table Updates” on page 57 for a description of other requirements associated with the use of this instruction.

Special Registers Altered:
None

Architecture Note

See the first Architecture Note in the ***tlbie*** instruction description for an explanation of why ***tlbsync*** can be executed only in hypervisor state.

6.2 Page Table Update Synchronization Requirements

This section describes rules that software should follow when updating the Page Table, and includes suggested sequences of operations for some representative cases.

In the sequences of operations shown in the following subsections, any alteration of a Page Table Entry (PTE) that corresponds to a single line in the sequence is assumed to be done using a *Store* instruction for which the access is atomic. Appropriate modifications must be made to these sequences if this assumption is not satisfied (e.g., if a † store doubleword operation is done using two *Store Word* instructions).

Sequences that use the ***tlbie*** instruction may require a context synchronizing operation before and/or after the sequence; see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79. Similarly, sequences that add a PTE require a context synchronizing operation after the sequence if the new entry is needed in order to translate the effective addresses of subsequent instructions.

Page Table Entries must not be changed in a manner that causes an implicit branch.

6.2.1 Page Table Updates

TLBs are non-coherent caches of the HTAB. TLB entries must be invalidated explicitly with one of the *TLB Invalidate* instructions.

Unsynchronized lookups in the HTAB continue even while it is being modified. Any processor, including the processor modifying the HTAB, may look in the HTAB at any time in an attempt to reload a TLB entry. When altering a PTE, software must ensure that the PTE's Valid bit is 0 if the PTE is inconsistent (e.g., if the RPN field is not correct for the current AVPN field).

Updates of Reference, Change, and Tag Set bits by the processor are not synchronized with the accesses that cause the updates. When modifying the low-order half of a PTE, software must take care to avoid overwriting a processor update of these bits and to avoid having the value written by a *Store* instruction overwritten by a processor update. The processor does not alter any other fields of the PTE.

In a multiprocessor system, when one or more ***tlbie*** instructions have been executed by a processor in a given partition, the following sequence of instructions must be executed by that processor before a ***tlbie*** or ***tlbsync*** instruction is executed by another processor in that partition.

eieio
tlbsync
sync

Other instructions may be interleaved with this sequence of instructions, but these instructions must appear in the order shown.

Programming Note

The ***eieio*** instruction prevents the reordering of ***tlbie*** instructions previously executed by the processor with respect to the subsequent ***tlbsync*** instruction. The ***tlbsync*** instruction and the subsequent ***sync*** instruction together ensure that all storage accesses for which the address was translated using the translations being invalidated, and all Reference, Change, and Tag Set bit updates associated with address translations that were performed using the translations being invalidated, will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that processor or mechanism.

Similarly, when a ***tlbsync*** instruction has been executed by a processor in a given partition, a ***sync*** instruction must be executed by that processor before a ***tlbie*** or ***tlbsync*** instruction is executed by another processor in that partition.

The sequences of operations shown in the following subsections assume a multiprocessor environment. In a uniprocessor environment the ***tlbsync*** can be omitted, as can the ***eieio*** that separates the ***tlbie*** from the ***tlbsync***.

6.2.1.1 Adding a Page Table Entry

This is the simplest Page Table case. The Valid bit of the old entry is assumed to be 0. The following sequence can be used to create a PTE, maintain a consistent state, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE(TS,RPN,AC,R,C,WIMG,N,PP) ← new values
eieio          /* order 1st update before 2nd    */
PTE(AVPN,SW,H,V) ← new values (V=1)
sync           /* order updates before next
                  storage accesses               */
```

6.2.1.2 Modifying a Page Table Entry

General Case

If a valid entry is to be modified and the translation instantiated by the entry being modified is to be invalidated, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes. (The sequence is equivalent to deleting the PTE and then adding a new one; see Sections 6.2.1.3 and 6.2.1.1.)

```
PTE_V ← 0          /* (other fields don't matter)    */
sync               /* order update before tlbie and
                      before next storage accesses   */
tlbie(old_seg_type,old_VPN_32:79–p,old_L) /* invali-
                      date old translation           */
eieio              /* order tlbie before tlbsync     */
tlbsync            /* order tlbie before sync        */
sync               /* order tlbie and tlbsync before
                      next storage accesses          */
PTE_TS,RPN,AC,R,C,WIMG,N,PP ← new values
eieio              /* order 2nd update before 3rd    */
PTE_AVPN,SW,H,V ← new values (V=1)
sync               /* order 2nd and 3rd updates before
                      next storage accesses          */
```

Resetting the Reference Bit

If the only change being made to a valid entry is to set the Reference bit to 0, a simpler sequence suffices because the Reference bit need not be maintained exactly.

```
oldR ← PTE_R       /* get old R                      */
if oldR = 1 then
  PTE_R ← 0        /* store byte (R=0, other bits
                      unchanged)                     */
  tlbie(seg_type,VPN_32:79–p,L) /* invalidate entry*/
  eieio            /* order tlbie before tlbsync     */
  tlbsync          /* order tlbie before sync        */
  sync             /* order tlbie, tlbsync, and
                      update before next storage
                      accesses                       */
```

Modifying the Virtual Address

If the virtual address translated by a valid PTE is to be modified and the new virtual address hashes to the same two PTEGs as does the old virtual address, the following sequence can be used to modify the PTE, maintain a consistent state, ensure that the translation instantiated by the old entry is no longer available, and ensure that a subsequent reference to the virtual address translated by the new entry will use the correct real address and associated attributes.

```
PTE_AVPN,SW,H,V ← new values (V=1)
sync               /* order update before tlbie and
                      before next storage accesses   */
tlbie(old_seg_type,old_VPN_32:79–p,old_L) /* invali-
                      date old translation           */
eieio              /* order tlbie before tlbsync     */
tlbsync            /* order tlbie before sync        */
sync               /* order tlbie and tlbsync before
                      next storage accesses          */
```

To modify the AC, N, or PP bits without overwriting a Reference, Change, or Tag Set bit update being performed by the processor or by some other processor in the system, a sequence similar to that shown above can be used except that the first line would be replaced by a ***sync*** instruction followed by a loop containing a ***ldarx***/***stdcx.*** pair that emulates an atomic "Compare and Swap" of the low-order doubleword of the PTE. (See the section entitled "Atomic Update Primitives" in Book II, *PowerPC AS Virtual Environment Architecture* for a description of "Compare and Swap".)

6.2.1.3 Deleting a Page Table Entry

The following sequence can be used to ensure that the translation instantiated by an existing entry is no longer available.

```
PTE_V ← 0          /* (other fields don't matter)    */
sync               /* order update before tlbie and
                      before next storage accesses   */
tlbie(old_seg_type,old_VPN_32:79–p,old_L) /* invali-
                      date old translation           */
eieio              /* order tlbie before tlbsync     */
tlbsync            /* order tlbie before sync        */
sync               /* order tlbie and tlbsync before
                      next storage accesses          */
```

Chapter 7. Interrupts

7.1 Overview

The PowerPC AS architecture provides an interrupt mechanism to allow the processor to change state as a result of external signals, errors, or unusual conditions arising in the execution of instructions.

System Reset and Machine Check interrupts are not ordered. All other interrupts are ordered such that only one interrupt is reported, and when it is processed (taken) no program state is lost. Since save/restore registers SRR0 and SRR1 are serially reusable resources used by most interrupts, program state may be lost when an unordered interrupt is taken.

7.2 Interrupt Synchronization

When an interrupt occurs, SRR0 is set to point to an instruction such that all preceding instructions have completed execution, no subsequent instruction has begun execution, and the instruction addressed by SRR0 may or may not have completed execution, depending on the interrupt type.

With the exception of System Reset and Machine Check interrupts, all interrupts are context synchronizing as defined in Section 1.6.1, “Context Synchronization” on page 3. System Reset and Machine Check interrupts are context synchronizing if they are recoverable (i.e., if bit 62 of SRR1 is set to 1 by the interrupt). If a System Reset or Machine Check interrupt is not recoverable (i.e., if bit 62 of SRR1 is set to 0 by the interrupt), it acts like a context synchronizing operation with respect to subsequent instructions. That is, a non-recoverable System Reset or Machine Check interrupt need not satisfy items 1 through 3 of Section 1.6.1, but does satisfy items 4 and 5.

7.3 Interrupt Classes

Interrupts are classified by whether they are directly caused by the execution of an instruction or are caused by some other system exception. Those that are "system-caused" are:

- System Reset
- Machine Check
- External
- Decrementer

External and Decrementer are maskable interrupts. While $MSR_{EE}=0$, the interrupt mechanism ignores the exceptions that generate these interrupts. Therefore, software may delay the generation of these interrupts by setting $MSR_{EE}=0$ or by failing to set $MSR_{EE}=1$ after processing an interrupt. When any interrupt is taken, MSR_{EE} is set to 0 by the interrupt mechanism, delaying the recognition of any further exceptions causing these interrupts.

System Reset and Machine Check exceptions are not maskable. These exceptions will be recognized regardless of the setting of the MSR.

"Instruction-caused" interrupts are further divided into two classes, *precise* and *imprecise*.

7.3.1 Precise Interrupt

Except for the Imprecise Mode Floating-Point Enabled Exception type Program interrupt, all instruction-caused interrupts are precise. When the fetching or execution of an instruction causes a precise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.
2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some storage accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
3. The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have been partially executed, or may have completed, depending on the interrupt type.
4. Architecturally, no subsequent instruction has begun execution.

7.3.2 Imprecise Interrupt

This architecture defines one imprecise interrupt, the Imprecise Mode Floating-Point Enabled Exception type Program interrupt.

When the execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point.

1. SRR0 addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
2. An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.
3. If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another interrupt (e.g., Alignment, Data Storage), then SRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed (see Section 7.6, "Partially Executed Instructions" on page 72).
4. If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than ***sync*** or ***isync***, then SRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for forcing the imprecise interrupt). If the imprecise interrupt is forced by a ***sync*** or ***isync*** instruction, then SRR0 may address either the ***sync*** or ***isync*** instruction, or the following instruction.
5. If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by SRR0 appears not to have begun execution, if it is not the excepting instruction.
6. No instruction following the instruction addressed by SRR0 appears to have begun execution.

All Floating-Point Enabled Exception type Program interrupts are maskable using the MSR bits FE0 and FE1. Although these interrupts are maskable, they differ significantly from the other maskable interrupts in that the masking of these interrupts is usually controlled by the application program, whereas the masking of External and Decrementer interrupts is controlled by the operating system.

Architecture Note

An implementation may define one or more additional interrupts to be imprecise. If this is done, then a complete description of how such imprecise interrupts are implemented by the processor and how they are to be handled by the operating system can be found in the Book IV, *PowerPC AS Implementation Features* document for the implementation. Such an implementation must provide a means of forcing the processor to process interrupts in a precise fashion as described here, perhaps with reduced performance.

The discussion here assumes that only the Imprecise Mode Floating-Point Enabled Exception type Program interrupt is imprecise.

7.4 Interrupt Processing

Associated with each kind of interrupt is an *interrupt vector*, which contains the initial sequence of instructions that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in other registers, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt will occur, the following actions are performed. The handling of Machine Check and System Call Vectored interrupts (see Sections 7.5.2 and 7.5.15 respectively) differs from the description given below in several respects.

1. SRR0 is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. Bits 33:36 and 42:47 of SRR1 are loaded with information specific to the interrupt type.
3. Bits 0:32, 37:41, and 48:63 of SRR1 are loaded with a copy of the corresponding bits of the MSR.
4. The MSR is set as shown in Figure 30 on page 62. In particular, MSR bits IR and DR are set to 0, disabling relocation, and MSR bit SF is set to 1, selecting 64-bit mode. The new values take effect beginning with the first instruction executed following the interrupt.
5. Instruction fetch and execution resumes, using the new MSR value, at the effective address specific to the interrupt type. These effective addresses are shown in Figure 31 on page 62.

Interrupts do not clear reservations obtained with ***lwarx*** or ***ldarx***.

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a "new" program.

- ***stwcx.*** or ***stdcx.***, to clear the reservation if one is outstanding, to ensure that a ***lwarx*** or ***ldarx*** in the interrupted program is not paired with a ***stwcx.*** or ***stdcx.*** in the "new" program.
- ***sync***, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another processor before the program is resumed on that other processor.
- ***isync*** or ***rfid***, to ensure that the instructions in the "new" program execute in the "new" context.

Programming Note

In order to handle Machine Check and System Reset interrupts correctly, the operating system should manage MSR_{RI} as follows.

- In the Machine Check and System Reset interrupt handlers, interpret SRR1 bit 62 (where MSR_{RI} is placed) as:
  - 0: interrupt is not recoverable
  - 1: interrupt is recoverable
- In each interrupt handler, when enough state has been saved that a Machine Check or System Reset interrupt can be recovered from, set MSR_{RI} to 1.
- In each interrupt handler, do the following (in order) just before returning.
  1. Set MSR_{RI} to 0.
  2. Set SRR0 and SRR1 to the values to be used by ***rfid***. The new value of SRR1 should have bit 62 set to 1 (which will happen naturally if SRR1 is restored to the value saved there by the interrupt, because the interrupt handler will not be executing this sequence unless the interrupt is recoverable).
  3. Execute ***rfid***.

MSR_{RI} can be managed similarly to handle interrupts other than Machine Check and System Reset that occur within interrupt handlers.

This Note describes only the management of MSR_{RI}. It is not intended to be a full description of the requirements for an interrupt handler.

Engineering Note

Implementations that use emulation assists must report in SRR0 the effective address of the instruction being emulated, and in the DAR if applicable the effective address that would have been computed by the instruction being emulated.

7.5 Interrupt Definitions

Figure 30 shows all the types of interrupts and the values assigned to the MSR for each. Figure 31 shows the effective address of the interrupt vector for each interrupt type. (Section 4.2.6 on page 29 summarizes all architecturally defined uses of effective addresses, including those implied by Figure 31.)

Interrupt Type	MSR Bit							
	IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	-	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	0	0	0	0	0	0	-	m
Data Segment	0	0	0	0	0	0	-	m
Instruction Storage	0	0	0	0	0	0	-	m
Instruction Segment	0	0	0	0	0	0	-	m
External	0	0	0	0	0	0	-	m
Alignment	0	0	0	0	0	0	-	m
Program	0	0	0	0	0	0	-	m
FP Unavailable	0	0	0	0	0	0	-	m
Decrementer	0	0	0	0	0	0	-	m
System Call	0	0	0	0	0	0	-	s
Trace	0	0	0	0	0	0	-	m
Performance Monitor	0	0	0	0	0	0	-	m
System Call Vectored	1	1	-	-	-	-	-	v

0 bit is set to 0
1 bit is set to 1
- bit is not altered
m if LPES=0, set to 1; otherwise (implementation-dependent) set to 0 or unchanged
s if LEV=1 or LPES=0, set to 1; otherwise (implementation-dependent) set to 0 or unchanged
v set to 0 or unchanged (implementation-dependent)

Bits BE, FP, PMM, PR, and SE are set to 0.

In *tags active* mode, the US bit is not altered.

In *tags inactive* mode, the US bit is treated as reserved and is set as if written as 0.

If the optional Little-Endian facility is implemented (see the section entitled “Little-Endian” in Book I), the bits associated with the facility are set as follows. The ILE bit is not altered. The LE bit is copied from the ILE bit except for System Call Vectored interrupt. For System Call Vectored interrupt, LE is not altered.

Bit SF is set to 1.

Bit TA is not altered.

Reserved bits are set as if written as 0.

Figure 30. MSR setting due to interrupt

Programming Note

For all the cases in which it is implementation-dependent whether the interrupt sets MSR_{HV} to 0 or leaves it unchanged, the bit should already be 0, so the fact that some implementations do not set it to 0 does not matter.

Effective Address[1]	Interrupt Type
00..0000_0100	System Reset
00..0000_0200	Machine Check
00..0000_0300	Data Storage
00..0000_0380	Data Segment
00..0000_0400	Instruction Storage
00..0000_0480	Instruction Segment
00..0000_0500	External
00..0000_0600	Alignment
00..0000_0700	Program
00..0000_0800	Floating-Point Unavailable
00..0000_0900	Decrementer
00..0000_0A00	Reserved
00..0000_0B00	Reserved
00..0000_0C00	System Call
00..0000_0D00	Trace
00..0000_0E00	Reserved
00..0000_0E10	Reserved
. . .	. . .
00..0000_0EFF	Reserved
00..0000_0F00	Performance Monitor
00..0000_0F10	Reserved
. . .	. . .
00..0000_0FFF	Reserved
FF..FF00_3000	System Call Vectored
FF..FF00_3020	System Call Vectored
. . .	. . .
FF..FF00_3FE0	System Call Vectored
FF..FF00_3FFF	(end of ***scv*** interrupt vectors)

[1] The values in the Effective Address column are interpreted as follows.
- 00..0000_nnnn *means* 0x0000_0000_0000_nnnn
- FF..FF00_nnnn *means* 0xFFFF_FFFF_FF00_nnnn

[2] Effective addresses 0x0000_0000_0000_0000 through 0x0000_0000_0000_00FF are used by software and will not be assigned as interrupt vectors.

Figure 31. Effective address of interrupt vector by interrupt type

Programming Note

When address translation is disabled, use of any of the effective addresses that are shown as reserved in Figure 31 risks incompatibility with future implementations.

7.5.1 System Reset Interrupt

If a System Reset exception causes an interrupt that is not context synchronizing, or causes the loss of a Machine Check exception, an External exception, or a Floating-Point Enabled Exception type Program exception, the interrupt is not recoverable.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 Set to 0.

42:47 Set to 0.

62 Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0100.

Each implementation provides a means for software to distinguish power-on Reset from other types of System Reset, and describes it in the Book IV, *PowerPC AS Implementation Features* document for the implementation.

Engineering Note

Every attempt should be made to allow continuing execution.

If the result of a System Reset interrupt is the same as that produced by an External interrupt with the exception of where execution resumes, the interrupt is recoverable. This condition exists if none of the specified exceptions have been lost and if the state of the processor has not been corrupted by an error in the processor.

7.5.2 Machine Check Interrupt

The causes of Machine Check interrupts are implementation-dependent. For example, a Machine Check interrupt may be caused by a reference to a storage location that contains an uncorrectable error or does not exist (see Section 4.2.7, “Invalid Real Address” on page 29), or by an error in the storage subsystem.

Machine Check interrupts are enabled when $MSR_{ME}=1$. If $MSR_{ME}=0$ and a Machine Check occurs, the processor enters the Checkstop state. The Checkstop state may also be entered if an access is attempted to a storage location that does not exist (see Section 4.2.7).

Disabled Machine Check (Checkstop State)

When a processor is in Checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. Some implementations may preserve some or all of the internal state of the processor when entering Checkstop state, so that the state can be analyzed as an aid in problem determination.

Enabled Machine Check

If a Machine Check exception causes an interrupt that is not context synchronizing, or causes the loss of an External exception or a Floating-Point Enabled Exception type Program exception, the interrupt is not recoverable.

In some systems, the operating system may attempt to identify and log the cause of the Machine Check.

The following registers are set:

SRR0 Set on a “best effort” basis to the effective address of some instruction that was executing or was about to be executed when the Machine Check exception occurred. For further details see the Book IV, *PowerPC AS Implementation Features* document for the implementation.

SRR1

62 Loaded from bit 62 of the MSR if the processor is in a recoverable state; otherwise set to 0.

Others See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0200.

Programming Note

If a Machine Check interrupt is caused by an error in the storage subsystem, the storage subsystem may return incorrect data, which may be placed into registers. This corruption of register contents may occur even if the interrupt is recoverable.

Engineering Note

Every attempt should be made to allow continuing execution.

If the result of a Machine Check interrupt is the same as that produced by an External interrupt with the exception of where execution resumes, the interrupt is recoverable. This condition exists if none of the specified exceptions have been lost and if the state of the processor has not been corrupted by an error in the processor. A load operation that places data, possibly corrupted by the storage subsystem, into a GPR or FPR does not make the interrupt unrecoverable.

7.5.3 Data Storage Interrupt

A Data Storage interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- Data address translation is enabled ($MSR_{DR}=1$) and the virtual address of any byte of the storage location specified by a *Load, Store,* ***icbi, dcbz, dcbst, dcbf, eciwx,*** or ***ecowx*** instruction cannot be translated to a real address.
- The effective address specified by a ***lq, stq, lwarx, ldarx, stwcx.***, or ***stdcx.*** instruction refers to storage that is Write Through Required or Caching Inhibited.
- The access violates storage protection.
- A Data Address Compare match or a Data Address Breakpoint Register (DABR) match occurs.
- Execution of an ***eciwx*** or ***ecowx*** instruction is disallowed because $EAR_E=0$.
- The effective address calculation of a *Load*, *Store*, ***icbi, dcbz, dcbst***, or ***dcbf*** instruction results in an Effective Address Overflow (EAO) exception (see Book I, *PowerPC AS User Instruction Set Architecture*).

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Storage interrupt, and either (a) the specified effective address refers to storage that is Write Through Required or Caching Inhibited, or (b) a non-conditional *Store* to the specified effective address would cause a Data Storage interrupt, it is implementation-dependent whether a Data Storage interrupt occurs.

If a *Move Assist* instruction has a length of zero (in the XER), a Data Storage interrupt does not occur for reasons of Effective Address Overflow, address translation, or storage protection, regardless of the effective address.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

DSISR

0 Set to 0.

1 Set to 1 if $MSR_{DR}=1$ and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.

2:3 Set to 0.

4 Set to 1 if the access is not permitted by the storage protection mechanism; otherwise set to 0.

Programming Note

The only cases in which $DSISR_4$ can be set to 1 for an access that occurs when $MSR_{DR}=0$ are those described in Figure 25. These cases can be distinguished from other causes of data storage protection violations by examining $SRR1_{59}$ (the bit in which MSR_{DR} was saved by the interrupt).

5 Set to 1 if the access is due to a ***lq, stq, lwarx, ldarx, stwcx.***, or ***stdcx.*** instruction that addresses storage that is Write Through Required or Caching Inhibited; otherwise set to 0.

6 Set to 1 for a *Store*, ***dcbz,*** or ***ecowx*** instruction; otherwise set to 0.

7:8 Set to 0.

9 Set to 1 if a Data Address Compare match or a DABR match occurs; otherwise set to 0.

10 Set to 0.

11 Set to 1 if execution of an ***eciwx*** or ***ecowx*** instruction is attempted when $EAR_E=0$; otherwise set to 0.

12:14 Set to 0.

15 Set to 1 if $MSR_{DR}=1$, the translation for an attempted access is found in the SLB, the translation is not found in the primary PTEG or in the secondary PTEG, and $SLBE_L=1$; otherwise set to 0.

16:30 Set to 0.

31 Set to 1 if an Effective Address Overflow (EAO) exception caused the interrupt.

DAR Set to the effective address of a storage element as described in the following list. The list should be read from the top down; the DAR is set as described by the first item that corresponds to an exception that is reported in the DSISR. For example, if a *Load* instruction causes a storage protection violation and a DABR match (and both are reported in the DSISR), the DAR is set to the effective address of a byte in the first aligned doubleword for which access was attempted in the page that caused the exception.

- undefined, for an EAO exception
- a Data Storage exception occurs for reasons other than DABR match or, for ***eciwx*** and ***ecowx***, $EAR_E = 0$
  - — a byte in the block that caused the exception, for a *Cache Management* instruction
  - — a byte in the first aligned doubleword for which access was attempted in the page that caused the exception, for a *Load, Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 7.7)
- undefined, for a DABR match, or if ***eciwx*** or ***ecowx*** is executed when $EAR_E = 0$

If the interrupt occurs in 32-bit mode, the high-order 32 bits of the DAR are set to 0.

† If multiple Data Storage exceptions occur for a given effective address, any one or more of the bits corresponding to these exceptions may be set to 1 in the DSISR, subject to the requirement that if Effective Address Overflow occurs for this effective address then bit 31 is set to 1.

Programming Note

More than one bit may be set to 1 in the DSISR in the following combinations.

1, {s+}
1, 15, {s+}
4, {s+}
4, 5, {s}
5, {s}
{s+}

In this list, "{s}" represents any combination of the set of bits {6, 9, 31} and "{s+}" adds bit 11 to this set.

Execution resumes at effective address 0x0000_0000_0000_0300.

Engineering Note

† For initial hardware debug it is often useful to run with cache disabled. In some ways cache disabled mode is similar to Caching Inhibited storage. Although ***lq*** and ***stq*** need not be supported by an implementation for storage that is Caching Inhibited, support for ***lq*** and ***stq*** with cache disabled should be considered.

7.5.4 Data Segment Interrupt

A Data Segment interrupt occurs when no higher priority exception exists and a data access cannot be performed because data address translation is enabled ($MSR_{DR} = 1$) and the effective address of any byte of the storage location specified by a *Load*, *Store*, ***icbi, dcbz, dcbst, dcbf, eciwx,*** or ***ecowx*** instruction cannot be translated to a virtual address.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of a Data Segment interrupt, and a non-conditional *Store* to the specified effective address would cause a Data Segment interrupt, it is implementation-dependent whether a Data Segment interrupt occurs.

If a *Move Assist* instruction has a length of zero (in the XER), a Data Segment interrupt does not occur, regardless of the effective address.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 30 on page 62.

DAR Set to the effective address of a storage element as described in the following list.

- a byte in the block that caused the Data Segment interrupt, for a *Cache Management* instruction
- a byte in the first aligned doubleword for which access was attempted in the segment that caused the Data Segment interrupt, for a *Load, Store*, ***eciwx***, or ***ecowx*** instruction ("first" refers to address order; see Section 7.7)

If the interrupt occurs in 32-bit mode, the high-order 32 bits of the DAR are set to 0.

Execution resumes at effective address 0x0000_0000_0000_0380.

Programming Note

A Data Segment interrupt cannot occur for an SLS address. For a PLS address or in *tags inactive* mode it occurs if $MSR_{DR}=1$ and the translation of the effective address of any byte of the specified storage location is not found in the SLB.

7.5.5 Instruction Storage Interrupt

An Instruction Storage interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched for any of the following reasons.

- Instruction address translation is enabled ($MSR_{IR}=1$) and the virtual address cannot be translated to a real address.
- The fetch access violates storage protection.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33 Set to 1 if $MSR_{IR}=1$ and the translation for an attempted access is not found in the primary PTEG or in the secondary PTEG; otherwise set to 0.

34 Set to 0.

35 Set to 1 if the access occurs when $MSR_{IR}=1$ and is to No-execute storage or to Guarded storage; otherwise set to 0.

36 Set to 1 if the access is not permitted by Figure 23, 24, or 25, as appropriate; otherwise set to 0.

Programming Note

The only cases in which $SRR1_{36}$ can be set to 1 for an access that occurs when $MSR_{IR}=0$ are those described in Figure 25. These cases can be distinguished from other causes of instruction storage protection violations that set $SRR1_{36}$ to 1 by examining $SRR1_{58}$ (the bit in which MSR_{IR} was saved by the interrupt).

42:46 Set to 0.

47 Set to 1 if $MSR_{IR}=1$, the translation for an attempted access is found in the SLB, the translation is not found in the primary PTEG or in the secondary PTEG, and $SLBE_L=1$; otherwise set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

If multiple Instruction Storage exceptions occur due to attempting to fetch a single instruction, any one or more of the bits corresponding to these exceptions may be set to 1 in SRR1.

Programming Note

More than one bit may be set to 1 in SRR1 in the following combinations.

33, 35
33, 47
33, 35, 47
35, 36

Execution resumes at effective address 0x0000_0000_0000_0400.

7.5.6 Instruction Segment Interrupt

An Instruction Segment interrupt occurs when no higher priority exception exists and the next instruction to be executed cannot be fetched because instruction address translation is enabled ($MSR_{IR}=1$) and the effective address cannot be translated to a virtual address.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 is set to the branch target address).

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0480.

Programming Note

An Instruction Segment interrupt cannot occur for an SLS address. For a PLS address or in *tags inactive* mode it occurs if $MSR_{IR}=1$ and the translation of the effective address of the next instruction to be executed is not found in the SLB.

7.5.7 External Interrupt

An External interrupt occurs when no higher priority exception exists, an External interrupt exception is presented to the interrupt mechanism, and $MSR_{EE}=1$. The occurrence of the interrupt does *not* cancel the request.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0500.

7.5.8 Alignment Interrupt

An Alignment interrupt occurs when no higher priority exception exists and a data access cannot be performed for any of the following reasons.

- The operand of a floating-point *Load* or *Store* is not word-aligned, or crosses a virtual page boundary.
- The operand of ***lq, stq, lmw, lmd, stmw, stmd, lwarx, ldarx, stwcx., stdcx., eciwx***, or ***ecowx*** is not aligned.
- The operand of a single-register *Load* or *Store* is not aligned and the processor is in Little-Endian mode.
- The instruction is ***lq, stq, lmw, lmd, stmw, stmd, lswi, lswx, lsdi, lsdx, stswi, stswx, stsdi,*** or ***stsdx***, and the operand is in storage that is Write Through Required or Caching Inhibited, or the processor is in Little-Endian mode.
- The operand of a *Load* or *Store* crosses a segment boundary, or crosses a boundary between virtual pages that have different storage control attributes.
- The operand of a *Load* or *Store* is not aligned and is in storage that is Write Through Required or Caching Inhibited.
- The operand of ***dcbz, lwarx, ldarx, stwcx.,*** or ***stdcx.*** is in storage that is Write Through Required or Caching Inhibited.

Engineering Note

Early implementations have additional requirements for ***lmw*** and ***stmw***, for reasons of compatibility with the POWER Architecture. See the section entitled “Load/Store Multiple Instructions” in the “Incompatibilities with the POWER Architecture” appendix of Book I.

If a ***stwcx.*** or ***stdcx.*** would not perform its store in the absence of an Alignment interrupt and the specified effective address refers to storage that is Write Through Required or Caching Inhibited, it is implementation-dependent whether an Alignment interrupt occurs.

Setting the DSISR and DAR as described below is optional for implementations on which Alignment interrupts occur rarely, if ever, for cases that the Alignment interrupt handler emulates. For such implementations, if the DSISR and DAR are not set as described below they are set to undefined values.

Engineering Note

For a given implementation, decisions regarding whether to set the DSISR and DAR as described in the remainder of this section, and what potential causes of Alignment interrupts actually cause Alignment interrupts, must include consideration of the cases that the Alignment interrupt handler would emulate, and of the effect of such emulation on software performance.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.
42:47 Set to 0.
Others Loaded from the MSR.

MSR See Figure 30 on page 62.

DSISR

0:11 Set to 0.
12:13 Set to bits 30:31 of the instruction if DS-form.
Set to 0b00 if D-, DQ-, or X-form.

14 Set to 0.

15:16 Set to bits 29:30 of the instruction if X-form. Set to 0b00 if D-, DS-, or DQ-form.

17 Set to bit 25 of the instruction if X-form. Set to bit 5 of the instruction if D-, DS-, or DQ-form.

18:21 Set to bits 21:24 of the instruction if X-form. Set to bits 1:4 of the instruction if D-, DS-, or DQ-form.

22:26 Set to bits 6:10 of the instruction (RT/RS/FRT/FRS), except undefined for ***dcbz***.

27:31 Set to bits 11:15 of the instruction (RA) for update form instructions; set to either bits 11:15 of the instruction or to any register number not in the range of registers to be loaded for a valid form ***lmw***, a valid form ***lswi***, or a valid form ***lswx*** for which neither RA nor RB is in the range of registers to be loaded; otherwise undefined.

Engineering Note

The requirement for ***lmw, lswi***, and ***lswx*** ensures that the program that emulates these instructions when they cause an Alignment interrupt on the 601 can also be used on subsequent PowerPC AS implementations. (601 implements POWER semantics for these instructions, preserving RA when it is in the range to be loaded and is not 0. Therefore the 601 Alignment interrupt handler must do the same. Software wants to use the same Alignment interrupt handler for all PowerPC AS implementations. This requires that the "RA field" saved in the DSISR for post-601 implementations not be in the range that would be loaded if the effective address were aligned.)

For ***lmw***, the requirement can be met either by storing zeros or by storing the RT field with 1 subtracted from it. For ***lswi*** and ***lswx***, it can be met by storing the RT field with 1 subtracted from it (the *Load String* instructions wrap from GPR 31 to GPR 0, so simply storing zeros is not adequate).

† **DAR** Set to the effective address computed by the instruction, except that if the interrupt occurs in 32-bit mode the high-order 32 bits of the DAR are set to 0.

† For an X-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the same value that would have resulted if the corresponding D- or DS-form instruction had caused the interrupt. Similarly, for a D- or DS-form *Load* or *Store*, it is acceptable for the processor to set the DSISR to the value that would have resulted for the corresponding X-form instruction. For example, an unaligned ***lwax*** (that crosses a protection boundary) would normally, following the description above, cause the DSISR to be set to binary:

```
000000000000 00 0 01 0 0101 ttttt ?????
```

where "ttttt" denotes the RT field, and "?????" † denotes an undefined 5-bit value. However, it is acceptable if it causes the DSISR to be set as for ***lwa***, which is

```
000000000000 10 0 00 0 1101 ttttt ?????
```

If there is no corresponding alternative form instruction (e.g., for ***lwaux***), the value described above is set in the DSISR.

The instruction pairs that may use the same DSISR value are:

lhz/lhzx	lhzu/lhzux	lha/lhax	lhau/lhaux
lwz/lwzx	lwzu/lwzux	lwa/lwax	
ld/ldx	ldu/ldux		
sth/sthx	sthu/sthux	stw/stwx	stwu/stwux
std/stdx	stdu/stdux		
lfs/lfsx	lfsu/lfsux	lfd/lfdx	lfdu/lfdux
stfs/stfsx	stfsu/stfsux	stfd/stfdx	stfdu/stfdux

Execution resumes at effective address 0x0000_0000_0000_0600.

Programming Note

The architecture does not support the use of an unaligned effective address by ***lwarx, ldarx, stwcx., stdcx., eciwx***, and ***ecowx***. If an Alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the Alignment interrupt handler must not attempt to simulate the instruction, but instead should treat the instruction as a programming error.

7.5.9 Program Interrupt

A Program interrupt occurs when no higher priority exception exists and one of the following exceptions arises during execution of an instruction:

Floating-Point Enabled Exception

A Floating-Point Enabled Exception type Program interrupt is generated when the expression

$$(MSR_{FE0} \mid MSR_{FE1}) \ \& \ FPSCR_{FEX}$$

is 1. $FPSCR_{FEX}$ is set to 1 by the execution of a floating-point instruction that causes an enabled exception, including the case of a *Move To FPSCR* instruction that causes an exception bit and the corresponding enable bit both to be 1.

Illegal Instruction

An Illegal Instruction type Program interrupt is generated when execution is attempted of an illegal instruction, or of a reserved or optional instruction that is not provided by the implementation.

An Illegal Instruction type Program interrupt may be generated when execution is attempted of any of the following kinds of instruction.

- an instruction that is in invalid form
- an ***lswx*** instruction for which RA or RB is in the range of registers to be loaded
- an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values, or an ***mftb*** instruction with a TBR field that does not contain one of the defined values

Engineering Note

Early implementations have additional requirements for instructions that are reserved because they correspond to non-privileged POWER instructions that are not in PowerPC AS, and for ***mtspr*** and ***mfspr***, for reasons of compatibility with the POWER Architecture. See the sections entitled "Move To/From SPR" and "Discontinued Opcodes" in the "Incompatibilities with the POWER Architecture" appendix of Book I.

Privileged Instruction

The following applies if the instruction is executed when MSR_{PR} = 1.

A Privileged Instruction type Program interrupt is generated when execution is attempted of a privileged instruction, or of an ***mtspr*** or ***mfspr*** instruction with an SPR field that contains one of the defined values having spr_0=1. It may be generated when execution is attempted of an ***mtspr*** or ***mfspr*** instruction with an SPR field that does not contain one of the defined values but has spr_0=1, or when execution is attempted of an ***mftb*** instruction with a TBR field that does not contain one of the defined values but has tbr_0=1.

The following applies if the instruction is executed when $MSR_{HV\ PR}$ = 0b00.

A Privileged Instruction type Program interrupt may be generated when execution is attempted of an ***mtspr*** instruction with an SPR field that designates a hypervisor resource (see Section 1.7, "Logical Partitioning (LPAR)" on page 4), or when execution of a ***tlbie*** or ***tlbsync*** instruction is attempted.

Programming Note

These are the only cases in which a Privileged Instruction type Program interrupt can be generated when MSR_{PR}=0. They can be distinguished from other causes of Privileged Instruction type Program interrupts by examining $SRR1_{49}$ (the bit in which MSR_{PR} was saved by the interrupt).

Trap

A Trap type Program interrupt is generated when any of the conditions specified in a *Trap* instruction is met.

The following registers are set:

SRR0 For all Program interrupts except a Floating-Point Enabled Exception when in one of the Imprecise modes, set to the effective address of the instruction that caused the Program interrupt.

For an Imprecise Mode Floating-Point Enabled Exception, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. If SRR0 points to a subsequent instruction, that instruction has not been executed. If a subsequent instruction is ***isync*** or ***sync***, SRR0 will not point more than four bytes beyond the ***isync*** or ***sync*** instruction.

If $FPSCR_{FEX}$=1 but Floating-Point Enabled Exception type Program interrupts are disabled by having both MSR_{FE0} and MSR_{FE1} = 0, a Floating-Point Enabled Exception type Program interrupt will occur prior to or at the next synchronizing event if these MSR bits are altered by any instruction that can set the MSR so that the expression

$$(MSR_{FE0} \mid MSR_{FE1})\ \&\ FPSCR_{FEX}$$

is 1. When this occurs, SRR0 is loaded with the address of the instruction that would have executed next, not with the address of the instruction that modified the MSR causing the interrupt.

SRR1

- **33:36** Set to 0.
- **42** Set to 0.
- **43** Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.
- **44** Set to 1 for an Illegal Instruction type Program interrupt; otherwise set to 0.
- **45** Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.
- **46** Set to 1 for a Trap type Program interrupt; otherwise set to 0.

47 Set to 0 if SRR0 contains the address of the instruction causing the exception, and to 1 if SRR0 contains the address of a subsequent instruction.

Others Loaded from the MSR.

Only one of bits 43:46 can be set to 1.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0700.

Engineering Note

If the Imprecise Recoverable Mode Floating-Point Enabled Exception type Program interrupt is implemented as imprecise, the hardware must provide, at the minimum, the address at which to resume the interrupted process (this is given in SRR0), the excepting instruction's opcode, extended opcode, and record bit, the source values or registers, and the target register. This information can be provided directly in registers or by means of a pointer to the excepting instruction. The manner in which it is provided is described in the Book IV, *PowerPC AS Implementation Features* document for the implementation.

7.5.10 Floating-Point Unavailable Interrupt

A Floating-Point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and $MSR_{FP}=0$.

The following registers are set:

SRR0 Set to the effective address of the instruction that caused the interrupt.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0800.

7.5.11 Decrementer Interrupt

A Decrementer interrupt occurs when no higher priority exception exists, the Decrementer exception exists, and $MSR_{EE}=1$. The occurrence of the interrupt cancels the request.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0900.

7.5.12 System Call Interrupt

A System Call interrupt occurs when a *System Call* instruction is executed.

The following registers are set:

SRR0 Set to the effective address of the instruction following the *System Call* instruction.

SRR1

33:36 Set to 0.

42:47 Set to 0.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0C00.

Programming Note

An attempt to execute an ***sc*** instruction with LEV=1 in problem state should be treated as a programming error.

7.5.13 Trace Interrupt

A Trace interrupt occurs when no higher priority exception exists and either MSR_{SE}=1 and any instruction except ***rfid*** or ***rfscv*** is successfully completed, or MSR_{BE}=1 and a *Branch* instruction is completed. Successful completion means that the instruction caused no other interrupt. Thus a Trace interrupt never occurs for a *System Call* instruction, or for a *Trap* instruction that traps. The instruction that causes a Trace interrupt is called the "traced instruction".

When a Trace interrupt occurs, the following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 and 42:47 See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

Others Loaded from the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0x0000_0000_0000_0D00.

Extensions to the Trace facility are described in Appendix F, "Example Trace Extensions (Optional)" on page 115.

Programming Note

The following instructions are not traced.

- ***rfid***
- ***rfscv***
- ***sc***, ***scv***, and *Trap* instructions that trap
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler
- instructions that are emulated by software

In general, interrupt handlers can achieve the effect of tracing these instructions.

Architecture Note

If a Trace interrupt were permitted after an ***rfid*** or ***rfscv***, the Trace interrupt handler would never be able to return to a program for which MSR_{SE}=1.

7.5.14 Performance Monitor Interrupt (Optional)

The Performance Monitor interrupt is part of the optional Performance Monitor facility; see Appendix E. If the Performance Monitor facility is not implemented or does not use this interrupt, the corresponding interrupt vector (see Figure 31 on page 62) is treated as reserved.

7.5.15 System Call Vectored Interrupt

A System Call Vectored interrupt occurs when a *System Call Vectored* instruction is executed in *tags active* mode.

The following registers are set:

LR Set to the effective address of the instruction following the *System Call Vectored* instruction.

CTR

33:36 undefined

42:47 undefined

Others Loaded from corresponding bits of the MSR.

MSR See Figure 30 on page 62.

Execution resumes at effective address 0xFFFF_FFFF_FF00_3 || LEV || 0b0_0000, where LEV is the 7-bit value specified by the *System Call Vectored* instruction.

Programming Note

Because the System Call Vectored interrupt sets MSR_{IR} to 1, the effective address described above is translated to a real address before being used to access storage. If the effective address cannot be translated, or if instructions cannot be fetched from the addressed storage location (e.g., the access would violate storage protection, or would be to No-execute storage), an Instruction Storage interrupt occurs before the first instruction at the effective address is executed.

Because the System Call Vectored interrupt uses save/restore registers that differ from those used by other interrupts, the System Call Vectored interrupt handler can run with address translation enabled and External interrupts enabled. Similarly, the Programming Note about managing MSR_{RI} in Section 7.4, "Interrupt Processing" on page 61 does not apply to the System Call Vectored interrupt handler (the System Call Vectored interrupt does not alter MSR_{RI}).

7.6 Partially Executed Instructions

If a system-caused, Data Storage, Data Segment, or Alignment exception occurs while a *Load* or *Store* instruction is executing, the instruction may be aborted. In such cases the instruction is not completed, but may have been partially executed in the following respects.

- Some of the bytes of the storage operand may have been accessed, except that if access to a given byte of the storage operand would cause Effective Address Overflow or would violate storage protection, that byte is neither copied to a register by a *Load* instruction nor modified by a *Store* instruction. Also, the rules for storage accesses given in Section 4.2.4.1, "Guarded Storage" on page 26 and in the section entitled "Instruction Restart" in Book II are obeyed.
- Some registers may have been altered as described in the Book II section cited above.
- Reference, Change, and Tag Set bits may have been updated as described in Section 4.8.
- For a ***stwcx.*** or ***stdcx.*** instruction that is executed in-order, CR0 and the FXCC may have been set to undefined values and the reservation may have been cleared.
- For an ***lq*** instruction that is executed in-order, the TGCC may have been set to an undefined value.

The architecture does not support continuation of an aborted instruction but intends that the aborted instruction be re-executed if appropriate.

Programming Note

An exception may result in the partial execution of a *Load* or *Store* instruction. For example, if the Page Table Entry that translates the address of the storage operand is altered, by a program running on another processor, such that the new contents of the Page Table Entry preclude performing the access, the alteration could cause the *Load* or *Store* instruction to be aborted after having been partially executed.

As stated in the Book II section cited above, if an instruction is partially executed the contents of registers are preserved to the extent that the instruction can be re-executed correctly. The consequent preservation is described in the following list. For any given instruction, zero, one, or two items in the list apply.

- For a fixed-point *Load* instruction that is not a multiple or string form, or for an ***eciwx*** instruction, if RT = RA or RT = RB then the contents of register RT are not altered.
- For an ***lq*** instruction, if RT+1 = RA then the contents of register RT+1 are not altered.
- For an update form *Load* or *Store* instruction, the contents of register RA are not altered.

7.7 Exception Ordering

Since multiple exceptions can exist at the same time and the architecture does not provide for reporting more than one interrupt at a time, the generation of more than one interrupt is prohibited. Also some exceptions would be lost if they were not recognized and handled when they occurred. For example, if an External interrupt was generated when a Data Storage exception existed, the Data Storage exception would be lost. If the Data Storage exception was caused by a *Store Multiple* instruction for which the storage operand crosses a virtual page boundary and the exception was a result of attempting to access the second virtual page, the store could have modified locations in the first virtual page even though it appeared that the *Store Multiple* instruction was never executed.

In addition, the architecture defines imprecise interrupts that must be recoverable, cannot be lost, and can occur at any time with respect to the executing instruction stream. Some of the maskable and nonmaskable exceptions are persistent and can be deferred. The following exceptions persist even though some other interrupt is generated:

- Floating-Point Enabled Exceptions
- External
- Decrementer

For the above reasons, all exceptions are prioritized with respect to other exceptions that may exist at the same instant to prevent the loss of any exception that is not persistent. Some exceptions cannot exist at the same instant as some others.

Data Storage, Data Segment, and Alignment exceptions occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0).

7.7.1 Unordered Exceptions

The exceptions listed here are unordered, meaning that they may occur at any time regardless of the state of the interrupt processing mechanism. These exceptions are recognized and processed when presented.

1. System Reset
2. Machine Check

7.7.2 Ordered Exceptions

The exceptions listed here are ordered with respect to the state of the interrupt processing mechanism.

System-Caused or Imprecise

1. Program
   - Imprecise Mode Floating-Point Enabled Exception
2. External
3. Decrementer

Instruction-Caused and Precise

1. Instruction Segment
2. Instruction Storage
3. Program
   - Illegal Instruction
   - Privileged Instruction
4. Function-Dependent

   4.a Fixed-Point

   1a Program
   - Trap

   1b System Call or System Call Vectored

   1c.1 EAO type Data Storage

   1c.2 non-EAO type Data Storage, or Data Segment or Alignment

   2 Trace

   4.b Floating-Point

   1 FP Unavailable

   2a Program
   - Precise Mode Floating-Point Enabled Excep'n

   2b.1 EAO type Data Storage

   2b.2 non-EAO type Data Storage, or Data Segment or Alignment

   3 Trace

For implementations that execute multiple instructions in parallel using pipeline or superscalar techniques, or combinations of these, it can be difficult to understand the ordering of exceptions. To understand this ordering it is useful to consider a model in which each instruction is fetched, then decoded, then executed, all before the next instruction is fetched. In this model, the exceptions a single instruction would generate are in the order shown in the list of instruction-caused exceptions. Exceptions with different numbers have different ordering. Exceptions with the same numbering but different lettering are mutually exclusive and cannot be caused by the same instruction. Where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal ordering.

Even on processors that are capable of executing several instructions simultaneously, or out of order, instruction-caused interrupts (precise and imprecise) occur in program order.

7.8 Interrupt Priorities

This section describes the relationship of nonmaskable, maskable, precise, and imprecise interrupts. In the following descriptions, the interrupt mechanism waiting for all possible exceptions to be reported includes only exceptions caused by previously initiated instructions (e.g., it does not include waiting for the Decrementer to step through zero). The exceptions are listed in order of highest to lowest priority.

1. System Reset

   System Reset exception has the highest priority of all exceptions. If this exception exists, the interrupt mechanism ignores all other exceptions and generates a System Reset interrupt.

   Once the System Reset interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

2. Machine Check

   Machine Check exception is the second highest priority exception. If this exception exists and a System Reset exception does not exist, the interrupt mechanism ignores all other exceptions and generates a Machine Check interrupt.

   Once the Machine Check interrupt is generated, no nonmaskable interrupts are generated due to exceptions caused by instructions issued prior to the generation of this interrupt.

3. Instruction-Dependent

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the processor may generate any one of the three interrupts for which an exception exists).

   A. Fixed-Point Loads and Stores

      a. EAO type Data Storage
      b. non-EAO type Data Storage, or Data Segment or Alignment
      c. Trace

B. Floating-Point Loads and Stores

a. Floating-Point Unavailable
b. EAO type Data Storage
c. non-EAO type Data Storage, or Data Segment or Alignment
d. Trace

C. Other Floating-Point Instructions

a. Floating-Point Unavailable
b. Program - Precise Mode Floating-Point Enabled Exception
c. Trace

D. ***rfid***, ***rfscv***, and ***mtmsr***[***d***]

a. Program - Privileged Instruction
b. Program - Precise Mode Floating-Point Enabled Exception
c. Trace, for ***mtmsr***[***d***] only

If the MSR bits FE0 and FE1 are set such that Precise Mode Floating-Point Enabled Exception type Program interrupts are enabled and FPSCR bit FEX is set, a Program interrupt will result prior to or at the next synchronizing event.

E. Other Instructions

a. These exceptions are mutually exclusive and have the same priority:
   - Program - Trap
   - System Call
   - System Call Vectored
   - Program - Privileged Instruction
   - Program - Illegal Instruction
b. Trace

F. Instruction Storage and Instruction Segment

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

4. Program - Imprecise Mode Floating-Point Enabled Exception

† This exception is the fourth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

5. External

† This exception is the fifth highest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

6. Decrementer

This exception is the lowest priority exception. When this exception is created, the interrupt mechanism waits for all other possible exceptions to be reported. It then generates this interrupt if no higher priority exception exists when the interrupt is to be generated.

Chapter 8. Timer Facilities

8.1 Overview

The Time Base and the Decrementer provide timing functions for the system. Both are volatile resources and must be initialized during startup. The ***mftb*** instruction is used to read the Time Base; the ***mtspr*** and ***mfspr*** instructions are used to write the Time Base and Decrementer and to read the Decrementer.

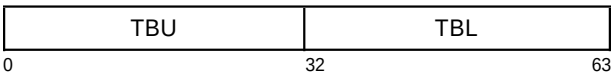
Time Base (TB)
The Time Base provides a long-period counter driven by an implementation-dependent frequency.

Decrementer (DEC)
The Decrementer, a counter that is updated at the same rate as the Time Base, provides a means of signaling an interrupt after a specified amount of time has elapsed unless

- the Decrementer is altered by software in the interim, **or**
- the Time Base update frequency changes.

8.2 Time Base

The Time Base (TB) is a 64-bit register (see Figure 32) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.



Field	*Description*
TBU	Upper 32 bits of Time Base
TBL	Lower 32 bits of Time Base

Figure 32. Time Base

The Time Base is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 4.

There is no automatic initialization of the Time Base; system software must perform this initialization.

The Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no interrupt or other indication when this occurs.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100\ \text{MHz}} = 5.90 \times 10^{12}\ \text{seconds}$$

which is approximately 187,000 years.

The Time Base must be implemented such that the following requirements are satisfied.

1. Loading a GPR from the Time Base shall have no effect on the accuracy of the Time Base.
2. Storing a GPR to the Time Base shall replace the value in the Time Base with the value in the GPR.

The PowerPC AS Architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in an PowerPC AS system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever

the update frequency of the Time Base changes, and a means to determine what the current update frequency is.

- The update frequency of the Time Base is under the control of the system software.

Implementations must provide a means for either preventing the Time Base from incrementing or preventing it from being read in problem state (MSR_{PR}=1). If the means is under software control, it must be accessible only in hypervisor state ($MSR_{HV\ PR}$ = 0b10). There must be a method for getting all processors' Time Bases to start incrementing with values that are identical or almost identical in all processors.

Architecture Note

Disabling the Time Base or making the ***mftb*** instruction privileged prevents the Time Base from being used to implement a "covert channel" in a secure system.

The requirements stated above for the Time Base apply also to any other SPRs that measure time and can be read in problem state (e.g., Performance Monitor registers).

Programming Note

If the hypervisor initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

See the description of the Time Base in Book II, *PowerPC AS Virtual Environment Architecture* for ways to compute time of day in POSIX format from the Time Base.

Architecture Note

It is intended that the Time Base be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing. The Time Base should not "tick" faster than the CPU instruction clock. Driving the Time Base directly from the CPU instruction clock is probably finer granularity than necessary; the instruction clock divided by 8, 16, or 32 would be more appropriate.

The Time Base driving frequency is also used to update the Decrementer (see Section 8.3), which is used by system software to set interval timers ("alarms"). The update frequency chosen should be appropriate for this purpose as well.

Engineering Note

One method that can be used to meet the requirement to synchronize Time Base values in all processors is to have a TB Enable input signal. When this signal is active, the Time Base is allowed to increment. When this signal is inactive, the Time Base does not increment. This signal may also be used to satisfy the requirement either to prevent the Time Base from incrementing or to prevent the Time Base from being read in problem state.

If the TB Enable input signal is implemented, the Decrementer does not decrement when this signal is inactive.

8.2.1 Writing the Time Base

Writing the Time Base is privileged, and can be done only in hypervisor state. Reading the Time Base is *not* privileged; it is discussed in Book II, *PowerPC AS Virtual Environment Architecture*.

It is not possible to write the entire 64-bit Time Base using a single instruction. The ***mttbl*** and ***mttbu*** extended mnemonics write the lower and upper halves of the Time Base (TBL and TBU), respectively, preserving the other half. These are extended mnemonics for the ***mtspr*** instruction; see page 94.

The Time Base can be written by a sequence such as:

```
lwz    Rx,upper   # load 64-bit value for
lwz    Ry,lower   #   TB into Rx and Ry
li     Rz,0
mttbl  Rz         # force TBL to 0
mttbu  Rx         # set TBU
mttbl  Ry         # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the Time Base is being initialized.

Programming Note

The instructions for writing the Time Base are implementation- and mode-independent. Thus code written to set the Time Base will work correctly in either 64-bit or 32-bit mode.

8.3 Decrementer

The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay.

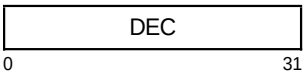


Figure 33. Decrementer

The Decrementer is driven by the same frequency as the Time Base. The period of the Decrementer will depend on the driving frequency, but if the same values are used as given above for the Time Base (see Section 8.2), and if the Time Base update frequency is constant, the period would be

$$T_{DEC} = \frac{2^{32} \times 32}{100\ \text{MHz}} = 1.37 \times 10^{3}\ \text{seconds}$$

which is approximately 23 minutes.

The Decrementer counts down, causing an interrupt (unless masked) when passing through zero. The Decrementer must be implemented such that the following requirements are satisfied.

1. The operation of the Time Base and the Decrementer is coherent, i.e., the counters are driven by the same fundamental time base.
2. Loading a GPR from the Decrementer shall have no effect on the accuracy of the Decrementer.
3. Storing a GPR to the Decrementer shall replace the value in the Decrementer with the value in the GPR.
4. Whenever bit 0 of the Decrementer changes from 0 to 1, an interrupt request is signaled. If multiple Decrementer interrupt requests are received before the first can be reported, only one interrupt is reported. The occurrence of a Decrementer interrupt cancels the request.
5. If the Decrementer is altered by software and the contents of bit 0 are changed from 0 to 1, an interrupt request is signaled.

Programming Note

In systems that change the Time Base update frequency for purposes such as power management, the Decrementer input frequency will also change. Software must be aware of this in order to set interval timers.

8.3.1 Writing and Reading the Decrementer

The contents of the Decrementer can be read or written using the ***mfspr*** and ***mtspr*** instructions, both of which are privileged when they refer to the Decrementer. Using an extended mnemonic (see page 94), the Decrementer may be written from GPR Rx using:

```
mtdec  Rx
```

Programming Note

If the execution of the ***mtdec*** instruction causes bit 0 of the Decrementer to change from 0 to 1, an interrupt request is signaled.

The Decrementer may be read into GPR Rx using:

```
mfdec  Rx
```

Copying the Decrementer to a GPR has no effect on the Decrementer contents or interrupt mechanism.

Chapter 9. Synchronization Requirements for Special Registers and for Lookaside Buffers

† Changing the contents of certain System Registers | and of SLB entries, and invalidating SLB and TLB entries, can have the *side effect* of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions † are executed and data accesses are performed. For example, changing MSR_{IR} from 0 to 1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order, and therefore may require explicit synchronization by software. (Program order is defined in Book II, *PowerPC AS Virtual Environment Architecture*.)

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or † in which instructions are executed or data accesses † are performed, is called a *context-altering instruction*. This chapter covers all the context-altering instructions. The software synchronization required for them is shown in Table 1 (for data access) and Table 2 (for instruction fetch and execution).

The notation "CSI" in the tables means any context | synchronizing instruction (i.e., ***sc, isync,*** or ***rfid***). A context synchronizing interrupt (i.e., any interrupt except non-recoverable System Reset or non-recoverable Machine Check) can be used instead of a context synchronizing instruction. If it is, phrases like "the synchronizing instruction", below, should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required before (after) a context-altering instruction, "the synchronizing instruction before (after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

Programming Note

Sometimes advantage can be taken of the fact that certain instructions that occur naturally in the | program, such as the ***rfid*** at the end of an interrupt handler, provide the required synchronization.

No software synchronization is required before † altering the MSR using ***mtmsr***[***d***] (except perhaps | when altering the LE bit: see the tables), because ***mtmsr***[***d***] is execution synchronizing. No software synchronization is required before most of the other alterations shown in Table 2, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context synchronizing).

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
rfscv	none	none	1
sc	none	none	
scv	none	none	
Trap	none	none	
mtmsrd (SF)	none	CSI	
mtmsrd (TA)	none	CSI	
mtmsr[***d***] (ILE)	none	none	
mtmsr[***d***] (PR)	none	CSI	
mtmsr[***d***] (US)	none	CSI	
mtmsr[***d***] (DR)	none	CSI	
mtmsr[***d***] (LE)	—	—	1
mtsr[***in***]	CSI	CSI	
mtspr (ACCR)	CSI	CSI	
mtspr (SDR1)	***sync***	CSI	4, 5
mtspr (DABR)	—	—	3
mtspr (EAR)	CSI	CSI	
slbie	CSI	CSI	
slbia	CSI	CSI	
slbmte	CSI	CSI	11
tlbie	CSI	***sync***	6, 7
tlbia	CSI	***sync***	6

Table 1. Synchronization requirements for data access

Instruction or Event	Required Before	Required After	Notes
interrupt	none	none	
rfid	none	none	
rfscv	none	none	1
sc	none	none	
scv	none	none	
Trap	none	none	
mtmsrd (SF)	none	CSI	8
mtmsrd (TA)	—	CSI	9
mtmsr[***d***] (ILE)	none	none	
mtmsr[***d***] (EE)	none	none	2
mtmsr[***d***] (PR)	none	CSI	9
mtmsr[***d***] (FP)	none	CSI	
mtmsr[***d***] (FE0, FE1)	none	CSI	
mtmsr[***d***] (SE, BE)	none	CSI	
mtmsr[***d***] (US)	none	CSI	
mtmsr[***d***] (IR)	none	CSI	9
mtmsr[***d***] (RI)	none	none	
mtmsr[***d***] (LE)	—	—	1
mtsr[***in***]	none	CSI	9
mtspr (SDR1)	***sync***	CSI	4, 5
mtspr (DEC)	none	none	10
mtspr (CTRL)	none	none	
slbie	none	CSI	
slbia	none	CSI	
slbmte	none	CSI	9, 11
tlbie	none	CSI or ***sync***	6, 7
tlbia	none	CSI or ***sync***	6

Table 2. Synchronization requirements for instruction fetch and/or execution

Notes:

1. Synchronization requirements for changing from one Endian mode to the other using the ***rfscv*** or ***mtmsr***[***d***] instruction are implementation-dependent, and are specified in the Book IV, *PowerPC AS Implementation Features* document for the implementation.
2. The effect of changing the EE bit is immediate.
   - If an ***mtmsr***[***d***] instruction sets the EE bit to 0, neither an External interrupt nor a Decrementer interrupt occurs after the ***mtmsr***[***d***] is executed.
   - If an ***mtmsr***[***d***] instruction changes the EE bit from 0 to 1 when an External, Decrementer, or higher priority exception exists, the corresponding interrupt occurs immediately after the ***mtmsr***[***d***] is executed, and before the next instruction is executed in the program that set EE to 1.
3. Synchronization requirements for changing the Data Address Breakpoint Register are implementation-dependent, and are specified in the Book IV, *PowerPC AS Implementation Features* document for the implementation.

4. SDR1 must not be altered when MSR_{DR}=1 or MSR_{IR}=1; if it is, the results are undefined.

   Architecture Note

   Altering SDR1 when MSR_{IR}=1 is prohibited because synchronizing Reference bit updates and instruction fetches, based on the old and new contents of SDR1, would be complex for both hardware and software. Altering SDR1 when MSR_{DR}=1 is prohibited because the capability is deemed not to be sufficiently useful to software to warrant verifying it.

   For most operating systems, SDR1 is expected not to be altered after it has been initialized. Therefore there is no need to support altering it quickly.

5. A ***sync*** instruction is required before the ***mtspr*** instruction because SDR1 identifies the Page Table and thereby the location of Reference, Change, and Tag Set bits. To ensure that Reference, Change, and Tag Set bits are updated in the correct Page Table, SDR1 must not be altered until all Reference, Change, and Tag Set bit updates associated with address translations that were performed, by the processor executing the ***mtspr*** instruction, before the ***mtspr*** instruction is executed have been performed with respect to that processor. A ***sync*** instruction guarantees this synchronization of Reference, Change, and Tag Set bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

   Architecture Note

   The reasoning given above might suggest that the architecture should require a ***sync*** instruction before alteration of MSR_{SF} and SLB entries (including alteration of SLB entries caused by ***mtsr***[***in***] instructions; see Section 11.1.2), because they can affect the mapping of effective addresses to virtual addresses and hence can affect which PTE a given effective address maps to. But it is unlikely that any implementation would use MSR_{SF} or an SLB entry more than once per storage access, whereas some implementations might use SDR1 twice — once to determine the real address for the access and once to set the Reference, Change, and Tag Set bits. Therefore the architecture does *not* require a ***sync*** instruction before alteration of MSR_{SF} and SLB entries.

   (An implementation might use SDR1 twice per storage access by keeping in the store queue entry for the Reference, Change, and Tag Set bit update only the offset in the Page Table of the corresponding PTE, instead of the PTE's real address, and then adding the Page Table address from SDR1 to the offset to determine where to update the Reference, Change, and Tag Set bits.)

6. For data accesses, the context synchronizing instruction before the ***tlbie*** or ***tlbia*** instruction ensures that all preceding instructions that access data storage have completed to a point at which they have reported all exceptions they will cause.

   A context synchronizing instruction after the ***tlbie*** or ***tlbia*** ensures that storage accesses associated with fetching instructions following the ***tlbie*** or ***tlbia*** will not use the TLB entry(s) being invalidated. A ***sync*** instruction (or a context synchronizing instruction) after the ***tlbie*** or ***tlbia*** has the corresponding effect for data accesses. A ***sync*** instruction also ensures that all storage accesses associated with instructions preceding the ***sync*** instruction, and all Reference, Change, and Tag Set bit updates associated with address translations that were performed, by the processor executing the ***sync*** instruction, before the ***sync*** instruction is executed, will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before any data accesses caused by instructions following the ***sync*** instruction are performed with respect to that processor or mechanism. If effects described in both the first and the third sentences of this paragraph are needed, both a context synchronizing instruction and a ***sync*** instruction must be used.

   Section 6.2, “Page Table Update Synchronization Requirements” on page 57 gives examples of the synchronization required when using ***tlbie*** in a sequence that alters a Page Table Entry.

   Programming Note

   The following sequence illustrates why it is necessary to ensure that all instructions that precede the ***tlbie*** or ***tlbia*** and access data storage have completed to a point at which they have reported all exceptions they will cause. Assume that valid SLB and Page Table entries exist for the target storage location when the sequence starts.

   1. A program issues a *Load* or *Store* instruction to a page.
   2. The same program marks the entry for the target page invalid in the Page Table.
   3. The same program executes a ***tlbie*** or ***tlbia*** that invalidates the corresponding TLB entry.
   4. The *Load* or *Store* instruction finally executes, and gets a page fault.

   The page fault is semantically incorrect. In order to prevent it, a context synchronizing instruction must be executed between steps 1 and 2.

7. Multiprocessor systems have additional requirements to synchronize “TLB shoot down” (i.e., to invalidate one or more TLB entries on all processors in the multiprocessor system and be able to ensure that the invalidations will have completed and that all side effects of the invalidations will have taken effect before any data accesses caused by subsequent instructions are performed); see Section 6.2.1, “Page Table Updates” on page 57.
8. The alteration must not cause an implicit branch in effective address space. Thus the ***mtmsrd*** instruction and all subsequent instructions, up to and including the next context synchronizing instruction, must have effective addresses that are less than 2^{32}.
9. The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.
10. The elapsed time between the contents of the Decrementer becoming negative and the signaling of the Decrementer exception is not defined.
11. If an ***slbmte*** instruction alters the mapping, or associated attributes, of a currently mapped ESID, the ***slbmte*** must be preceded by an ***slbie*** (or ***slbia***) instruction that invalidates the existing translation. This applies even if the corresponding entry is no longer in the SLB (the translation may still be in implementation-specific address translation lookaside information). No software synchronization is needed between the ***slbie*** and the ***slbmte***, regardless of whether the index of the SLB entry (if any) containing the current translation is the same as the SLB index specified by the ***slbmte***.

    No ***slbie*** (or ***slbia***) is needed if the ***slbmte*** instruction replaces a valid SLB entry with a mapping of a different ESID (e.g., to satisfy an SLB miss). However, the ***slbie*** is needed later if and when the translation that was contained in the replaced SLB entry is to be invalidated.

Chapter 10. Optional Facilities and Instructions

The facilities described in this chapter are optional. An implementation may provide all, some, or none of them.

10.1 External Control

The External Control facility permits a program to communicate with a special-purpose device. The facility consists of a Special Purpose Register, called EAR, and two instructions, called *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***).

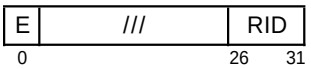
This facility must provide a means of synchronizing the devices with the processor to prevent the use of an address by the device when the translation that produced that address is being invalidated.

Engineering Note

Synchronization of devices with respect to ***tlbie*** instructions previously executed by the processor can be provided by ***tlbsync***.

10.1.1 External Access Register

This 32-bit Special Purpose Register controls access to the External Control facility and, for external control operations that are permitted, identifies the target device.



Bit(s)	*Name*	*Description*
0	E	Enable bit
26:31	RID	Resource ID

All other fields are reserved.

Figure 34. External Access Register

The EAR is a hypervisor resource; see Section 1.7, “Logical Partitioning (LPAR)” on page 4.

The high-order bits of the RID field that correspond to bits of the Resource ID beyond the width of the Resource ID supported by a particular implementation are treated as reserved bits.

Programming Note

The hypervisor can use the EAR to control which programs are allowed to execute *External Access* instructions, when they are allowed to do so, and which devices they are allowed to communicate with using these instructions.

10.1.2 External Access Instructions

The *External Access* instructions, *External Control In Word Indexed* (***eciwx***) and *External Control Out Word Indexed* (***ecowx***), are described in Book II, *PowerPC AS Virtual Environment Architecture*. Additional information about them is given below.

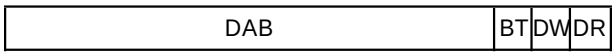
If attempt is made to execute either of these instructions when $EAR_E=0$, a Data Storage interrupt occurs with bit 11 of the DSISR set to 1.

The instructions are supported whenever $MSR_{DR}=1$. If either instruction is executed when $MSR_{DR}=0$ (real addressing mode), the results are boundedly undefined.

10.2 Data Address Breakpoint

The Data Address Breakpoint facility provides a means of detecting load and store accesses to a designated doubleword. The address comparison is done on an effective address, and is done independent of whether address translation is enabled or disabled.

† The Data Address Breakpoint facility is controlled by the Data Address Breakpoint Register (DABR).



Bit(s)	*Name*	*Description*
0:60	DAB	Data Address Breakpoint
61	BT	Breakpoint Translation Enable
62	DW	Data Write Enable
63	DR	Data Read Enable

Figure 35. Data Address Breakpoint Register

The DABR is a hypervisor resource; see Section 1.7, "Logical Partitioning (LPAR)" on page 4.

† A Data Address Breakpoint match occurs for a *Load* or *Store* instruction if, for any byte accessed,

- $EA_{0:60}=DABR_{DAB}$, and
- $MSR_{DR}=DABR_{BT}$, and
- the instruction is a *Store* and $DABR_{DW}=1$, or the instruction is a *Load* and $DABR_{DR}=1$.

In 32-bit mode the high-order 32 bits of the EA are treated as zeros for the purpose of detecting a match.

If the above conditions are satisfied, a match also occurs for ***eciwx*** and ***ecowx***. For the purpose of † determining whether a match occurs, ***eciwx*** is treated † as a *Load*, and ***ecowx*** is treated as a *Store*.

If the above conditions are satisfied, it is undefined whether a match occurs in the following cases.

- The instruction is *Store Conditional* but the store is not performed.
- The instruction is a *Load/Store String* of zero length.
- The instruction is ***dcbz***. (For the purpose of determining whether a match occurs, ***dcbz*** is treated as a *Store*.)

The *Cache Management* instructions other than ***dcbz*** never cause a match.

A Data Address Breakpoint match causes a Data Storage exception (see Section 7.5.3, "Data Storage Interrupt" on page 64). If a match occurs, some or all of the bytes of the storage operand may have been accessed; however, if a *Store* or ***ecowx*** instruction causes the match, the storage operand is not altered if the instruction is one of the following:

- a ***stq*** instruction for which the match occurs for the first doubleword of the storage operand
- any other *Store* instruction that causes an atomic access
- ***ecowx***

Programming Note

† The Data Address Breakpoint facility does not † apply to instruction fetches.

If a Data Address Breakpoint match occurs for a *Load* instruction for which any byte of the storage operand is in storage that is both Caching Inhibited and Guarded, or for an ***eciwx*** instruction, it may not be safe for software to restart the instruction.

Engineering Note

† In the case of a DABR match, it is preferable not † to access or alter any bytes of the storage operand at or after the breakpoint address. This makes the Data Address Breakpoint facility more useful for debugging.

10.3 Real Mode Storage Control

The Real Mode Storage Control facility provides a means of specifying portions of real storage that are treated as non-Guarded in real addressing mode ($MSR_{IR}=0$ or $MSR_{DR}=0$, as appropriate for the type of access). The remaining portions are treated as Guarded in real addressing mode (as is all of storage on implementations that do not provide this means). The means is a hypervisor resource (see Section 1.7, "Logical Partitioning (LPAR)" on page 4), and may also be system-specific.

The facility does not apply to implicit accesses to the Page Table by the hardware in performing address translation or recording reference, change, and tag set information. These accesses are performed as described in Section 4.2.5, "Real Addressing Mode" on page 27.

Programming Note

The preceding capability can be used to improve the performance of software that runs in real addressing mode, by causing accesses to instructions and data that occupy well-behaved storage to be treated as non-Guarded. Because in real addressing mode all storage is not Caching Inhibited, software should not map a Caching Inhibited virtual page to storage that is treated as non-Guarded in real addressing mode. Doing so could permit storage locations in the virtual page to be copied into the cache, which could lead to violations of the requirement given in Section 4.7.2 for changing the value of the I bit.

Engineering Note

The preceding capability should be provided at sufficiently fine granularity that the operating system can specify that kernel space (code and data) is treated as non-Guarded in real addressing mode and can map all application space to real storage that is treated as Guarded in real addressing mode. (This is necessary in order to prevent application code or data from being fetched into a cache when address translation is disabled.) A simple way to provide the capability is to use an implementation-specific SPR to specify the boundary between non-Guarded and Guarded real storage; any address below the value contained in the SPR would be treated as non-Guarded, and any address above the value (inclusive) would be treated as Guarded. It is permissible to treat any boundaries thus provided as if they were protection boundaries, with respect to causing an Alignment interrupt if an access crosses a boundary.

10.4 Move to Machine State Register Instruction

Move To Machine State Register X-form

mtmsr RS

31	RS	///	///	146	/
0	6	11	16	21	31

| $MSR_{58} \leftarrow (RS)_{58} \mid (RS)_{49}$
| $MSR_{59} \leftarrow (RS)_{59} \mid (RS)_{49}$
| $MSR_{32:50\ 52:57\ 60:63} \leftarrow (RS)_{32:50\ 52:57\ 60:63}$

| The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 32:50, 52:57, and 60:63 of register RS are placed into the corresponding bits of the MSR. The high-order 32 bits of the MSR are unchanged.

This instruction is privileged. This instruction is execution synchronizing except with respect to alterations to the LE bit; see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79.

In addition, alterations to the EE and RI bits are effective as soon as the instruction completes. Thus if MSR_{EE}= 0 and an External or Decrementer interrupt is pending, executing an ***mtmsr*** instruction that sets MSR_{EE} to 1 will cause the External or Decrementer interrupt to be taken before the next instruction is executed, if no higher priority exception exists (see Section 7.8, "Interrupt Priorities" on page 73).

Special Registers Altered:

MSR

| **Programming Note**

| If this instruction sets MSR_{PR} to 1, it also sets MSR_{IR} and MSR_{DR} to 1.

| This instruction does not alter MSR_{ME}. (This instruction does not alter MSR_{HV} because it does not alter any of the high-order 32 bits of the MSR.)

† **Programming Note**

† For a discussion of software synchronization requirements when altering certain MSR bits, see Chapter 9.

Programming Note

† There is no need for an analogous version of the ***mfmsr*** instruction, because the existing instruction copies the entire contents of the MSR to the selected GPR.

Chapter 11. Optional Facilities and Instructions that are being Phased Out of the Architecture

The facilities and instructions described in this chapter are optional. An implementation may provide all, some, or none of them.

Warning: These facilities and instructions are being phased out of the architecture.

The facilities and instructions described in this chapter are generally not mentioned elsewhere in Books I – III. Any conflict between this chapter and other parts of the Books is deemed to be resolved in favor of this chapter.

11.1 Bridge to SLB Architecture

The facility described in this section can be used to ease the transition to the current PowerPC AS software-managed Segment Lookaside Buffer (SLB) architecture, from either the Segment Register architecture provided by 32-bit PowerPC implementations or the hardware-accessed Segment Table architecture provided by 64-bit PowerPC implementations and by earlier PowerPC AS implementations.

The facility permits the operating system to continue to use the 32-bit PowerPC implementation's *Segment Register Manipulation* instructions, and to continue to use the Address Space Register (ASR).

Programming Note

Warning: This facility is being phased out of the architecture. It is likely not to be supported on future implementations. New programs should not use it.

Engineering Note

Decisions regarding whether to implement this facility in a given implementation, and how well to make it perform there, must include consideration of migration plans for existing software that uses it.

11.1.1 Address Space Register

The ASR is a 64-bit Special Purpose Register provided for operating system use.

Figure 36. Address Space Register

Programming Note

The ASR can be used to point to a Segment Table.

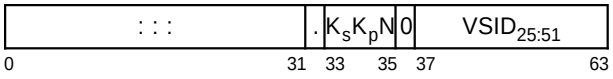
On earlier PowerPC AS implementations and on 64-bit PowerPC implementations, bits 0:51 of the ASR contained the high-order 52 bits of the 64-bit real address of the Segment Table, and bit 63 of the ASR indicated whether the specified Segment Table should (bit 63 = 1) or should not (bit 63 = 0) be searched by the processor when doing address translation.

11.1.2 Segment Register Manipulation Instructions

The instructions described in this section — ***mtsr***, ***mtsrin***, ***mfsr***, and ***mfsrin*** — allow software to associate effective segments 0 through 15 with any of virtual segments 0 through $2^{27}-1$. SLB entries 0:15 serve as virtual Segment Registers, with SLB entry i used to emulate Segment Register i. The ***mtsr*** and ***mtsrin*** instructions move 32 bits from a selected GPR to a selected SLB entry. The ***mfsr*** and ***mfsrin*** instructions move 32 bits from a selected SLB entry to a selected GPR.

The contents of the GPRs used by the instructions described in this section are shown in Figure 37. Fields shown as zeros must be zero for the *Move To Segment Register* instructions. Fields shown as hyphens are ignored. Fields shown as periods are ignored by the *Move To Segment Register* instructions and set to zero by the *Move From Segment Register* instructions. Fields shown as colons are ignored by the *Move To Segment Register* instructions and set to undefined values by the *Move From Segment Register* instructions.

RS/RT



RB

Figure 37. GPR contents for mtsr, mtsrin, mfsr, and mfsrin

Programming Note

The "Segment Register" format used by the instructions described in this section corresponds to the low-order 32 bits of RS and RT shown in the figure. This format is essentially the same as that for the Segment Registers of 32-bit PowerPC implementations. The only differences are the following.

- Bit 36 corresponds to a reserved bit in Segment Registers. Software must supply 0 for the bit because it corresponds to the L bit in SLB entries, and large pages are not supported for SLB entries created by the *Move To Segment Register* instructions.
- VSID bits 25:27 correspond to reserved bits in Segment Registers. Software can use these extra VSID bits to create VSIDs that are larger than those supported by the *Segment Register Manipulation* instructions of 32-bit PowerPC implementations.

Bit 32 of RS and RT corresponds to the T (direct-store) bit of early 32-bit PowerPC implementations. No corresponding bit exists in SLB entries.

Programming Note

The Programming Note in the introduction to Section 6.1.2.1, "SLB Management Instructions" on page 50 applies also to the *Segment Register Manipulation* instructions described in this section, and to any combination of the instructions described in the two sections, except as specified below for ***mfsr*** and ***mfsrin***.

The requirement that the SLB contain at most one entry that translates a given effective address (see Section 4.4.1.1, "Segment Lookaside Buffer (SLB)" on page 31) applies to SLB entries created by ***mtsr*** and ***mtsrin***. This requirement is satisfied naturally if only ***mtsr*** and ***mtsrin*** are used to create SLB entries for a given ESID, because for these instructions the association between SLB entries and ESID values is fixed (SLB entry i is used for ESID i). However, care must be taken if ***slbmte*** is also used to create SLB entries for the ESID, because for ***slbmte*** the association between SLB entries and ESID values is specified by software.

Move To Segment Register X-form

mtsr SR,RS

31	RS	/	SR	///	210	/
0	6	11	12	16	21	31

The SLB entry specified by SR is loaded from register RS, as follows.

SLBE Bit(s)	*Set to*	*SLB Field(s)*
0:31	0x0000_0000	$ESID_{0:31}$
32:35	SR	$ESID_{32:35}$
36	0b1	V
37:61	0x00_0000||0b0	$VSID_{0:24}$
62:88	$(RS)_{37:63}$	$VSID_{25:51}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

†

Special Registers Altered:

None

Architecture Note

The requirement that the *Segment Register Manipulation* instructions be executed only in 32-bit mode permits normal EA computation (in which the high-order 32 bits of the result are treated as zeros in 32-bit mode but not in 64-bit mode) to be used for ***mtsrin*** and ***mfsrin***.

Move To Segment Register Indirect X-form

mtsrin RS,RB

[POWER mnemonic: mtsri]

31	RS	///	RB	242	/
0	6	11	16	21	31

The SLB entry specified by $(RB)_{32:35}$ is loaded from register RS, as follows.

SLBE Bit(s)	*Set to*	*SLB Field(s)*
0:31	0x0000_0000	$ESID_{0:31}$
32:35	$(RB)_{32:35}$	$ESID_{32:35}$
36	0b1	V
37:61	0x00_0000||0b0	$VSID_{0:24}$
62:88	$(RS)_{37:63}$	$VSID_{25:51}$
89:91	$(RS)_{33:35}$	K_sK_pN
92	$(RS)_{36}$	L ($(RS)_{36}$ must be 0b0)
93	0b0	C

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction is privileged.

†

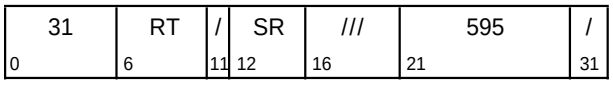
Special Registers Altered:

None

†

Move From Segment Register X-form

mfsr RT,SR



The contents of the low-order 27 bits of the VSID field, and the contents of the K_s, K_p, N, and L fields, of the SLB entry specified by SR are placed into register RT, as follows.

SLBE Bit(s)	*Copied to*	*SLB Field(s)*
62:88	$RT_{37:63}$	$VSID_{25:51}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID< 16, V=1, VSID< 2^{27}, L=0, and C=0). Otherwise the contents of register RT are undefined.

This instruction is privileged.

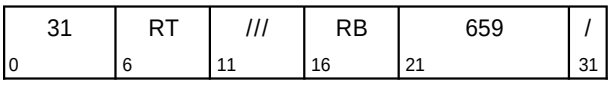
†

Special Registers Altered:

None

Move From Segment Register Indirect X-form

mfsrin RT,RB



The contents of the low-order 27 bits of the VSID field, and the contents of the K_s, K_p, N, and L fields, of the SLB entry specified by $(RB)_{32:35}$ are placed into register RT, as follows.

SLBE Bit(s)	*Copied to*	*SLB Field(s)*
62:88	$RT_{37:63}$	$VSID_{25:51}$
89:91	$RT_{33:35}$	K_sK_pN
92	RT_{36}	L ($SLBE_L$ must be 0b0)

RT_{32} is set to 0. The contents of $RT_{0:31}$ are undefined.

MSR_{SF} must be 0 when this instruction is executed; otherwise the results are boundedly undefined.

This instruction must be used only to read an SLB entry that was, or could have been, created by ***mtsr*** or ***mtsrin*** and has not subsequently been invalidated (i.e., an SLB entry in which ESID< 16, V=1, VSID< 2^{27}, L=0, and C=0). Otherwise the contents of register RT are undefined.

This instruction is privileged.

†

Special Registers Altered:

None

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended † mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and † symbols related to instructions defined in Book III.

† Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Move To/From Special Purpose Register Mnemonics

† This section defines extended mnemonics for the ***mtspr*** and ***mfspr*** instructions, including the Special Purpose Registers (SPRs) defined in Book I and certain privileged SPRs, and for the *Move From Time Base* instruction defined in Book II.

† The ***mtspr*** and ***mfspr*** instructions specify an SPR as a numeric operand; extended mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. Similar extended mnemonics are provided for the *Move From Time Base* instruction, which specifies the portion of the Time Base as a numeric operand.

Note: ***mftb*** serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the extended form. In the extended form the TBR operand is omitted and assumed to be 268 (the value that corresponds to TB).

Table 3 (Page 1 of 2). Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	Equivalent to	Extended	Equivalent to
Fixed Point Exception Register	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9
Data Storage Interrupt Status Register	mtdsisr Rx	mtspr 18,Rx	mfdsisr Rx	mfspr Rx,18
Data Address Register	mtdar Rx	mtspr 19,Rx	mfdar Rx	mfspr Rx,19
Decrementer	mtdec Rx	mtspr 22,Rx	mfdec Rx	mfspr Rx,22
Storage Description Register 1	mtsdr1 Rx	mtspr 25,Rx	mfsdr1 Rx	mfspr Rx,25
Save/Restore Register 0	mtsrr0 Rx	mtspr 26,Rx	mfsrr0 Rx	mfspr Rx,26
Save/Restore Register 1	mtsrr1 Rx	mtspr 27,Rx	mfsrr1 Rx	mfspr Rx,27
ACCR	mtaccr Rx	mtspr 29,Rx	mfaccr Rx	mfspr Rx,29
CTRL	mtctrl Rx	mtspr 152,Rx	mfctrl Rx	mfspr Rx,136
Special Purpose Registers G0 through G3	mtsprg *n*,Rx	mtspr 272+*n*,Rx	mfsprg Rx,*n*	mfspr Rx,272+*n*
Time Base [Lower]	mttbl Rx	mtspr 284,Rx	mftb Rx	mftb Rx,268
Time Base Upper	mttbu Rx	mtspr 285,Rx	mftbu Rx	mftb Rx,269
Processor Version Register	–	–	mfpvr Rx	mfspr Rx,287
MMCRA	mtmmcra Rx	mtspr 786,Rx	mfmmcra Rx	mfspr Rx,770
PMC1	mtpmc1 Rx	mtspr 787,Rx	mfpmc1 Rx	mfspr Rx,771
PMC2	mtpmc2 Rx	mtspr 788,Rx	mfpmc2 Rx	mfspr Rx,772
PMC3	mtpmc3 Rx	mtspr 789,Rx	mfpmc3 Rx	mfspr Rx,773
PMC4	mtpmc4 Rx	mtspr 790,Rx	mfpmc4 Rx	mfspr Rx,774
PMC5	mtpmc5 Rx	mtspr 791,Rx	mfpmc5 Rx	mfspr Rx,775
PMC6	mtpmc6 Rx	mtspr 792,Rx	mfpmc6 Rx	mfspr Rx,776
PMC7	mtpmc7 Rx	mtspr 793,Rx	mfpmc7 Rx	mfspr Rx,777

Table 3 (Page 2 of 2). Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR[1]	
	Extended	Equivalent to	Extended	Equivalent to
PMC8	mtpmc8 Rx	mtspr 794,Rx	mfpmc8 Rx	mfspr Rx,778
MMCR0	mtmmcr0 Rx	mtspr 795,Rx	mfmmcr0 Rx	mfspr Rx,779
MMCR1	mtmmcr1 Rx	mtspr 798,Rx	mfmmcr1 Rx	mfspr Rx,782
Processor Identification Register	–	–	mfpir Rx	mfspr Rx,1023

[1]Except for ***mftb*** and ***mftbu***.

Programming Note

The extended mnemonics in Table 3 for SPRs associated with the Performance Monitor facility are based on the definitions in Appendix E.

Other versions of Performance Monitor facilities used different sets of SPR numbers (all 32-bit PowerPC processors used a different set, and some early PowerPC AS processors used yet a different set).

Appendix B. Cross-Reference for Changed POWER Mnemonics

The following table lists the POWER instruction mnemonics that have been changed in the PowerPC AS Operating Environment Architecture, sorted by POWER mnemonic.

To determine the PowerPC AS mnemonic for one of these POWER mnemonics, find the POWER mnemonic in the second column of the table: the remainder of the line gives the PowerPC AS mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER mnemonics that have not changed are not listed. POWER instruction names that are the same in PowerPC AS are not repeated: i.e., for these, the last column of the table is blank.

Page	POWER		PowerPC AS	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
91	mtsri	Move To Segment Register Indirect	mtsrin	
12	rfsvc	Return From SVC	rfscv	Return From System Call Vectored
11	svca	Supervisor Call	sc	System Call
12	svcl	Supervisor Call	scv	System Call Vectored
55	tlbi	TLB Invalidate Entry	tlbie	

Appendix C. New Instructions

The following instructions in the PowerPC AS Operating Environment Architecture are new: they are not
† in the POWER Architecture.

| The following instructions are optional: ***tlbia, tlbsync,***
| ***mtmsr***. In addition the following instructions may optionally be provided as part of a "bridge" facility as
| described in Section 11.1, "Bridge to SLB
| Architecture" on page 89: ***mfsr, mfsrin, mtsr, mtsrin***.

|

mfsrin	Move From Segment Register Indirect
mtmsrd	Move To Machine State Register Doubleword

|

rfid	Return From Interrupt Doubleword
slbia	SLB Invalidate All
slbie	SLB Invalidate Entry
slbmfee	SLB Move From Entry ESID
slbmfev	SLB Move From Entry VSID
slbmte	SLB Move To Entry
tlbia	TLB Invalidate All
tlbsync	TLB Synchronize

Appendix D. Interpretation of the DSISR as Set by an Alignment Interrupt

For most causes of Alignment interrupt, the interrupt handler will emulate the interrupting instruction. To do this, it needs the following characteristics of the interrupting instruction:

Load or store
Length (halfword, word, doubleword, or quadword)
String, multiple, or elementary
Fixed-point or floating-point
Update or non-update
Byte reverse or not
Is it ***dcbz***?

The PowerPC AS Architecture optionally provides this information by setting bits in the DSISR that identify the interrupting instruction type. It is not necessary for the interrupt handler to load the interrupting instruction from storage. The mapping is unique except for a few exceptions that are discussed below. The near-uniqueness depends on the fact that many instructions, such as the fixed- and floating-point arithmetic instructions and the one-byte loads and stores, cannot cause an Alignment interrupt.

See Section 7.5.8, “Alignment Interrupt” on page 67 for a description of how the opcode and extended opcode are mapped to a DSISR value for an X-, D-, or DS-form instruction that causes an Alignment interrupt.

The table on the next page shows the inverse mapping: how the DSISR bits identify the interrupting instruction. The following notes are cited in the table.

(1) The instructions ***lwz*** and ***lwarx*** give the same DSISR bits (all zero). But if ***lwarx*** causes an Alignment interrupt, it should not be emulated. It is adequate for the Alignment interrupt handler simply to treat the instruction as if it were ***lwz***. The emulator must use the address in the DAR, rather than compute it from RA/RB/D, because ***lwz*** and ***lwarx*** have different instruction formats.

If opcode 0 (“Illegal or Reserved”) can cause an Alignment interrupt, it will be indistinguishable to the interrupt handler from ***lwarx*** and ***lwz***.

(2) These are distinguished by DSISR bits 12:13, which are not shown in the table.

The interrupt handler has no need to distinguish between an X-form instruction and the corresponding D- or DS-form instruction if one exists, and vice versa. Therefore two such instructions may yield the same DSISR value (all 32 bits). For example, ***stw*** and ***stwx*** may both yield either the DSISR value shown in the following table for ***stw***, or that shown for ***stwx***.

If DSISR 15:21 is:	then it is either X-form opcode:	or D/DS-form opcode:	so the instruction is:
00 0 0000	00000xxx00	x00000	lwarx, lwz, reserved (1)
00 0 0001	00010xxx00	x00010	ldarx
00 0 0010	00100xxx00	x00100	stw
00 0 0011	00110xxx00	x00110	-
00 0 0100	01000xxx00	x01000	lhz
00 0 0101	01010xxx00	x01010	lha
00 0 0110	01100xxx00	x01100	sth
00 0 0111	01110xxx00	x01110	lmw
00 0 1000	10000xxx00	x10000	lfs
00 0 1001	10010xxx00	x10010	lfd
00 0 1010	10100xxx00	x10100	stfs
00 0 1011	10110xxx00	x10110	stfd
00 0 1100	11000xxx00	x11000	lq
00 0 1101	11010xxx00	x11010	ld, ldu, lwa, lmd (2)
00 0 1110	11100xxx00	x11100	-
00 0 1111	11110xxx00	x11110	std, stdu, stmd, stq (2)
00 1 0000	00001xxx00	x00001	lwzu
00 1 0001	00011xxx00	x00011	-
00 1 0010	00101xxx00	x00101	stwu
00 1 0011	00111xxx00	x00111	-
00 1 0100	01001xxx00	x01001	lhzu
00 1 0101	01011xxx00	x01011	lhau
00 1 0110	01101xxx00	x01101	sthu
00 1 0111	01111xxx00	x01111	stmw
00 1 1000	10001xxx00	x10001	lfsu
00 1 1001	10011xxx00	x10011	lfdu
00 1 1010	10101xxx00	x10101	stfsu
00 1 1011	10111xxx00	x10111	stfdu
00 1 1100	11001xxx00	x11001	-
00 1 1101	11011xxx00	x11011	-
00 1 1110	11101xxx00	x11101	-
00 1 1111	11111xxx00	x11111	-
01 0 0000	00000xxx01		ldx
01 0 0001	00010xxx01		-
01 0 0010	00100xxx01		stdx
01 0 0011	00110xxx01		-
01 0 0100	01000xxx01		-
01 0 0101	01010xxx01		lwax
01 0 0110	01100xxx01		-
01 0 0111	01110xxx01		-
01 0 1000	10000xxx01		lswx
01 0 1001	10010xxx01		lswi
01 0 1010	10100xxx01		stswx
01 0 1011	10110xxx01		stswi
01 0 1100	11000xxx01		-
01 0 1101	11010xxx01		-
01 0 1110	11100xxx01		-
01 0 1111	11110xxx01		-
01 1 0000	00001xxx01		ldux
01 1 0001	00011xxx01		-
01 1 0010	00101xxx01		stdux
01 1 0011	00111xxx01		-
01 1 0100	01001xxx01		-
01 1 0101	01011xxx01		lwaux
01 1 0110	01101xxx01		-
01 1 0111	01111xxx01		-
01 1 1000	10001xxx01		lsdx
01 1 1001	10011xxx01		lsdi
01 1 1010	10101xxx01		stsdx
01 1 1011	10111xxx01		stsdi
01 1 1100	11001xxx01		-
01 1 1101	11011xxx01		-
01 1 1110	11101xxx01		-
01 1 1111	11111xxx01		-
10 0 0000	00000xxx10		-
10 0 0001	00010xxx10		-
10 0 0010	00100xxx10		stwcx.
10 0 0011	00110xxx10		stdcx.
10 0 0100	01000xxx10		-
10 0 0101	01010xxx10		-
10 0 0110	01100xxx10		-
10 0 0111	01110xxx10		-
10 0 1000	10000xxx10		lwbrx
10 0 1001	10010xxx10		-
10 0 1010	10100xxx10		stwbrx
10 0 1011	10110xxx10		-
10 0 1100	11000xxx10		lhbrx
10 0 1101	11010xxx10		-
10 0 1110	11100xxx10		sthbrx
10 0 1111	11110xxx10		-
10 1 0000	00001xxx10		-
10 1 0001	00011xxx10		-
10 1 0010	00101xxx10		-
10 1 0011	00111xxx10		-
10 1 0100	01001xxx10		eciwx
10 1 0101	01011xxx10		-
10 1 0110	01101xxx10		ecowx
10 1 0111	01111xxx10		-
10 1 1000	10001xxx10		-
10 1 1001	10011xxx10		-
10 1 1010	10101xxx10		-
10 1 1011	10111xxx10		-
10 1 1100	11001xxx10		-
10 1 1101	11011xxx10		-
10 1 1110	11101xxx10		-
10 1 1111	11111xxx10		dcbz
11 0 0000	00000xxx11		lwzx
11 0 0001	00010xxx11		-
11 0 0010	00100xxx11		stwx
11 0 0011	00110xxx11		-
11 0 0100	01000xxx11		lhzx
11 0 0101	01010xxx11		lhax
11 0 0110	01100xxx11		sthx
11 0 0111	01110xxx11		-
11 0 1000	10000xxx11		lfsx
11 0 1001	10010xxx11		lfdx
11 0 1010	10100xxx11		stfsx
11 0 1011	10110xxx11		stfdx
11 0 1100	11000xxx11		-
11 0 1101	11010xxx11		-
11 0 1110	11100xxx11		-
11 0 1111	11110xxx11		stfiwx
11 1 0000	00001xxx11		lwzux
11 1 0001	00011xxx11		-
11 1 0010	00101xxx11		stwux
11 1 0011	00111xxx11		-
11 1 0100	01001xxx11		lhzux
11 1 0101	01011xxx11		lhaux
11 1 0110	01101xxx11		sthux
11 1 0111	01111xxx11		-
11 1 1000	10001xxx11		lfsux
11 1 1001	10011xxx11		lfdux
11 1 1010	10101xxx11		stfsux
11 1 1011	10111xxx11		stfdux
11 1 1100	11001xxx11		-
11 1 1101	11011xxx11		-
11 1 1110	11101xxx11		-
11 1 1111	11111xxx11		-

Appendix E. Example Performance Monitor (Optional)

A Performance Monitor facility provides a means of collecting information about program and system performance.

The resources (e.g., SPR numbers) that a Performance Monitor facility may use are identified elsewhere in this Book. All other aspects of any Performance Monitor facility are implementation-dependent, and are described in the Book IV, *PowerPC AS Implementation Features* document for the implementation.

This appendix provides an example of a Performance Monitor facility. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail.

Programming Note

Because the features provided by a Performance Monitor facility are implementation-dependent, operating systems should provide services that support the useful performance monitoring functions in a generic fashion. Application programs should use these services, and should not depend on the features provided by a particular implementation.

The example Performance Monitor facility consists of the following features (described in detail in subsequent sections).

- one MSR bit
  - — PMM (Performance Monitor Mark), which can be used to select one or more programs for monitoring
- SPRs
  - — PMC1 – PMC8 (Performance Monitor Counter registers 1 – 8), which count events
  - — MMCR0 and MMCR1 (Monitor Mode Control Registers 0 and 1), which control the Performance Monitor facility
  - — SIAR and SDAR (Sampled Instruction Address Register and Sampled Data Address Register), which contain the address of the "sampled instruction" and of the "sampled data"
- the Performance Monitor interrupt, which can be caused by monitored conditions and events

The minimal subset of these features that makes the resulting Performance Monitor useful to applications consists of MSR_{PMM}, PMC1, PMC2, PMC3, PMC4, MMCR0, MMCR1, and MMCRA and certain bits of these three Monitor Mode Control Registers. These features support the counting of four selected events, and are identified as the "basic" features below. The remaining features (the remaining SPRs, the remaining bits in MMCR0, and the Performance Monitor interrupt) are considered "extensions".

The events that can be counted in the PMCs are implementation-dependent. The Book IV, *PowerPC AS Implementation Features* document for the implementation describes the events that are available for each PMC, and also the code that identifies each event. The events and codes may vary between PMCs, as well as between implementations. The event to be counted in a given PMC is selected by specifying the appropriate code in the MMCR "Selector" field for the PMC. As described in Book IV, some events may include operations that are performed out-of-order.

Many aspects of the operation of the Performance Monitor are summarized by the following hierarchy, which is described starting at the lowest level.

- A "counter negative condition" occurs when the value in a PMC is negative (i.e., when bit 0 of the PMC is 1). A "Time Base transition event" occurs when a selected bit of the Time Base changes from 0 to 1 (the bit is selected by an MMCR field). The term "condition or event" is used as an abbreviation for "counter negative condition or Time Base transition event". A condition or event can be caused implicitly by the processor (e.g., incrementing a PMC) or explicitly by software (***mtspr***).
- A condition or event is enabled if the corresponding "Enable" bit in an MMCR is 1. The occurrence of an enabled condition or event can have side effects within the Performance Monitor, such as causing the PMCs to cease counting.

- An enabled condition or event causes a Performance Monitor exception if Performance Monitor exceptions are enabled by the corresponding "Enable" bit in an MMCR. A single Performance Monitor exception may reflect multiple enabled conditions and events.
- A Performance Monitor exception causes a Performance Monitor interrupt when $MSR_{EE}=1$.

Programming Note

The Performance Monitor can be effectively disabled (i.e., put into a state in which Performance Monitor SPRs are not altered and Performance Monitor interrupts do not occur) by setting MMCR0 to 0x8000_0000.

E.1 PMM Bit of the Machine State Register

The Performance Monitor uses MSR bit PMM, which is defined as follows.

Bit	Description
61	***Performance Monitor Mark*** (PMM) This bit is a basic feature. This bit contains the Performance Monitor "mark" (0 or 1).

If an ***mtmsr*** or ***mtmsrd*** instruction is executed that changes the value of the PMM bit, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79).

Programming Note

Software can use this bit as a process-specific marker which, in conjunction with $MMCR0_{FCM0\ FCM1}$ (see Section E.2.2), permits events to be counted on a process-specific basis. (The bit is saved by interrupts and restored by ***rfid***.)

Common uses of the PMM bit include the following.

- Count events for a few selected processes. This use requires the following bit settings.
  - $MSR_{PMM}=1$ for the selected processes, $MSR_{PMM}=0$ for all other processes
  - $MMCR0_{FCM0}=1$
  - $MMCR0_{FCM1}=0$
- Count events for all but a few selected processes. This use requires the following bit settings.
  - $MSR_{PMM}=1$ for the selected processes, $MSR_{PMM}=0$ for all other processes
  - $MMCR0_{FCM0}=0$
  - $MMCR0_{FCM1}=1$

Notice that for both of these uses a mark value of 1 identifies the "few" processes and a mark value of 0 identifies the remaining "many" processes. Because the PMM bit is set to 0 when an interrupt occurs (see Figure 30 on page 62), interrupt handlers are treated as one of the "many". If it is desired to treat interrupt handlers as one of the "few", the mark value convention just described would be reversed.

Architecture Note

The two mark values (0 and 1) are equivalent except with respect to interrupts. That is, either mark value can be specified for a given process, and either mark value can control whether the PMCs are incremented, but interrupts always cause the mark value in the MSR to be set to 0 (see Figure 30).

Architecture Note

No MSR bit is provided to disable the Performance Monitor, because the Performance Monitor is considered a system-wide resource rather than a per-process resource. MMCR0 can be used to achieve the effect of disabling the Performance Monitor, as described in the introduction to Appendix E.

E.2 Special Purpose Registers

The Performance Monitor SPRs count events, control the operation of the Performance Monitor, and provide associated information.

The Performance Monitor SPRs can be read and written using the ***mfspr*** and ***mtspr*** instructions (see Section 3.4.1, "Move To/From System Register Instructions" on page 18). The Performance Monitor SPR numbers are shown in Figures 38. Writing any of the Performance Monitor SPRs is privileged. Reading any of the Performance Monitor SPRs is *not* privileged (however, the privileged SPR numbers used to write the SPRs can also be used to read them; see the figures).

The elapsed time between the execution of an instruction and the time at which events due to that instruction have been reflected in Performance Monitor SPRs is not defined. No means are provided by which software can ensure that all events due to preceding instructions have been reflected in Performance Monitor SPRs. Similarly, if the events being monitored may be caused by operations that are performed out-of-order, no means are provided by which software can prevent such events due to subsequent instructions from being reflected in Performance Monitor SPRs. Thus the value obtained by reading a Performance Monitor SPR may not be precise: it may fail to reflect some events due to instructions that precede the ***mfspr*** and may reflect some events due to instructions that follow the ***mfspr***. This lack of precision applies regardless of whether the state of the processor is such that the SPR is subject to change by the processor at the time the ***mfspr*** is executed.

If an ***mtspr*** instruction is executed that changes the value of a Performance Monitor SPR other than SIAR or SDAR, the change is not guaranteed to have taken effect until after a subsequent context synchronizing instruction has been executed (see Chapter 9, "Synchronization Requirements for Special Registers and for Lookaside Buffers" on page 79).

Programming Note

Depending on the events being monitored, the contents of Performance Monitor SPRs may be affected by aspects of the runtime environment (e.g., cache contents) that are not directly attributable to the programs being monitored.

SPR[1,2] decimal	$spr_{5:9}$ $spr_{0:4}$	Register Name	Privileged
770,786	11000 n0010	MMCRA	no,yes
771,787	11000 n0011	PMC1	no,yes
772,788	11000 n0100	PMC2	no,yes
773,789	11000 n0101	PMC3	no,yes
774,790	11000 n0110	PMC4	no,yes
775,791	11000 n0111	PMC5	no,yes
776,792	11000 n1000	PMC6	no,yes
777,793	11000 n1001	PMC7	no,yes
778,794	11000 n1010	PMC8	no,yes
779,795	11000 n1011	MMCR0	no,yes
780,796	11000 n1100	SIAR	no,yes
781,797	11000 n1101	SDAR	no,yes
782,798	11000 n1110	MMCR1	no,yes

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.
[2] For ***mtspr***, n must be 1. For ***mfspr***, reading the SPR is privileged if and only if n=1.

Figure 38. Performance Monitor SPR encodings for mtspr and mfspr

E.2.1 Performance Monitor Counter Registers

The eight Performance Monitor Counter registers, PMC1 through PMC8, are 32-bit registers that count events.

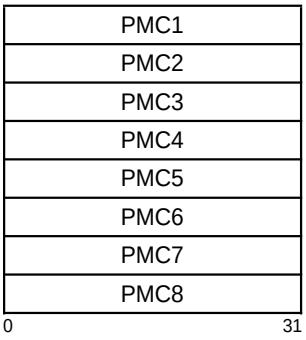


Figure 39. Performance Monitor Counter registers

PMC1 and PMC2 are basic features.

Normally each PMC is incremented each processor cycle by the number of times the corresponding event occurred in that cycle. Other modes of incrementing may also be provided (e.g., see the description of MMCR1 bits PMC1HIST and PMCjHIST).

"PMCj" is used as an abbreviation for "PMCi, i > 1".

Programming Note

Software can use a PMC to "pace" the collection of Performance Monitor data. For example, if it is desired to collect event counts every *n* cycles, software can specify that a particular PMC count cycles and set that PMC to 0x8000_0000 – *n*. The events of interest would be counted in other PMCs. The counter negative condition that will occur after *n* cycles can, with the appropriate setting of MMCR bits, cause counter values to become frozen, cause a Performance Monitor interrupt to occur, etc.

Architecture Note

Because they count events, the PMCs indirectly measure time and are therefore subject to the same requirements as the Time Base with respect to "covert channels" (see Section 8.2, "Time Base" on page 75). The requirements are satisfied by $MMCR0_{FC}$ (see Section E.2.2).

Architecture Note

The PMCs are numbered 1– 8, rather than 0– 7 which would be more consistent with the numbering in other register names, because early implementations of Performance Monitors numbered them thus.

E.2.2 Monitor Mode Control Register 0

Monitor Mode Control Register 0 (MMCR0) is a 32-bit register. This register, along with MMCR1, controls the operation of the Performance Monitor.

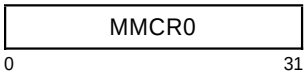


Figure 40. Monitor Mode Control Register 0

MMCR0 is a basic feature. Within MMCR0, some of the bits and fields are basic features and some are extensions. The basic bits and fields are identified as such, below.

Some bits of MMCR0 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR0 are as follows. MMCR0 bits that are not implemented are treated as reserved.

Bit(s) Description

0 ***Freeze Counters*** (FC)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented.

The processor sets this bit to 1 when an enabled condition or event occurs and $MMCR0_{FCECE}=1$.

1 ***Freeze Counters in Supervisor State*** (FCS)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{PR}=0$.

2 ***Freeze Counters in Problem State*** (FCP)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{PR}=1$.

3 ***Freeze Counters while Mark = 1*** (FCM1)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{PMM}=1$.

4 ***Freeze Counters while Mark = 0*** (FCM0)

This bit is a basic feature.

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $MSR_{PMM}=0$.

5 ***Performance Monitor Exception Enable*** (PMXE)

This bit is a basic feature.

0 Performance Monitor exceptions are disabled.
1 Performance Monitor exceptions are enabled until a Performance Monitor exception occurs, at which time:
- $MMCR0_{PMXE}$ is set to 0

Programming Note

Software can set this bit to 0 to prevent Performance Monitor interrupts.

Software can set this bit to 1 and then poll the bit to determine whether an enabled condition or event has occurred. This is especially useful on an implementation that does not provide the Performance Monitor interrupt.

6 ***Freeze Counters on Enabled Condition or Event*** (FCECE)

0 The PMCs are incremented (if permitted by other MMCR bits).

1 The PMCs are incremented (if permitted by other MMCR bits) until an enabled condition or event occurs when $MMCR0_{TRIGGER}=0$, at which time:
- $MMCR0_{FC}$ is set to 1

If the enabled condition or event occurs when $MMCR0_{TRIGGER}=1$, the FCECE bit is treated as if it were 0.

7:8 ***Time Base Selector*** (TBSEL)

This field selects the Time Base bit that can cause a Time Base transition event (the event occurs when the selected bit changes from 0 to 1).

00 Time Base bit 63 is selected.
01 Time Base bit 55 is selected.
10 Time Base bit 51 is selected.
11 Time Base bit 47 is selected.

Programming Note

Time Base transition events can be used to collect information about processor activity, as revealed by event counts in PMCs and by addresses in SIAR and SDAR, at periodic intervals.

In multiprocessor systems in which the Time Base registers are synchronized among the processors, Time Base transition events can be used to correlate the Performance Monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all the processors in the system.

Because the frequency of the Time Base is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.

9 ***Time Base Event Enable*** (TBEE)

0 Time Base transition events are disabled.
1 Time Base transition events are enabled.

10:15 ***Threshold*** (THRESHOLD)

This field contains a “threshold value”, which is a value such that only events that exceed the value are counted. The events to which a threshold value can apply are implementation-dependent, as are the dimension of the threshold (e.g., duration in cycles) and the granularity with which the threshold value is interpreted. See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

Programming Note

By varying the threshold value, software can obtain a profile of the characteristics of the events subject to the threshold. For example, if PMC1 counts the number of cache misses for which the duration exceeds the threshold value, then software can obtain the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

Engineering Note

A desirable use of THRESHOLD is to obtain a profile of the durations of cache misses.

It is recommended that one or two bits in a HID register be provided that permit software to control the granularity with which the THRESHOLD value is interpreted. For example, if one bit is provided the value 0 could specify a granularity of 1 and the value 1 could specify a granularity of 32.

16 ***PMC1 Condition Enable*** (PMC1CE)

This bit controls whether counter negative conditions due to a negative value in PMC1 are enabled.

0 Counter negative conditions for PMC1 are disabled.
1 Counter negative conditions for PMC1 are enabled.

17 ***PMCj Condition Enable*** (PMCjCE)

This bit controls whether counter negative conditions due to a negative value in any PMCj (i.e., in any PMC except PMC1) are enabled.

0 Counter negative conditions for all PMCjs are disabled.
1 Counter negative conditions for all PMCjs are enabled.

18 ***Trigger*** (TRIGGER)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 PMC1 is incremented (if permitted by other MMCR bits). The PMCjs are not incremented until PMC1 is negative or an enabled condition or event occurs, at which time:
- the PMCjs resume incrementing (if permitted by other MMCR bits)
- $MMCR0_{TRIGGER}$ is set to 0

See the description of the FCECE bit, above, regarding the interaction between TRIGGER and FCECE.

Programming Note

Uses of TRIGGER include the following.

- Resume counting in the PMCjs when PMC1 becomes negative, without causing a Performance Monitor interrupt. Then freeze all PMCs (and optionally cause a Performance Monitor interrupt) when a PMCj becomes negative. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time a PMCj becomes negative. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=0
  - PMCjCE=1
  - TBEE=0
  - FCECE=1
  - PMXE=1 (if a Performance Monitor interrupt is desired)
- Resume counting in the PMCjs when PMC1 becomes negative, and cause a Performance Monitor interrupt without freezing any PMCs. The PMCjs then reflect the events that occurred between the time PMC1 became negative and the time the interrupt handler reads them. This use requires the following MMCR0 bit settings.
  - TRIGGER=1
  - PMC1CE=1
  - TBEE=0
  - FCECE=0
  - PMXE=1

19:25 ***PMC1 Selector*** (PMC1SEL)

This field is a basic feature.

This field contains a code (one of at most 128 values) that identifies the event to be counted in PMC1; see the Book IV, *PowerPC AS Implementation Features* document for the implementation.

26:31 ***PMC2 Selector*** (PMC2SEL)

This field is a basic feature.

This field contains a code (one of at most 64 values) that identifies the event to be counted in PMC2; see Book IV.

E.2.3 Monitor Mode Control Register 1

Monitor Mode Control Register 1 (MMCR1) is a 32-bit register. This register, along with MMCR0, controls the operation of the Performance Monitor.

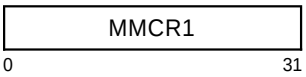


Figure 41. Monitor Mode Control Register 1

Some bits of MMCR1 are altered by the processor when various events occur, as described below.

The bit definitions of MMCR1 are as follows. MMCR1 bits that are not implemented are treated as reserved.

Bit(s) Description

0:4 ***PMC3 Selector*** (PMC3SEL)
5:9 ***PMC4 Selector*** (PMC4SEL)
10:14 ***PMC5 Selector*** (PMC5SEL)
15:19 ***PMC6 Selector*** (PMC6SEL)
20:24 ***PMC7 Selector*** (PMC7SEL)

Each of these fields contains a code (one of at most 32 values) that identifies the event to be counted in PMCs 3 through 7 respectively; see Book IV.

25:28 ***PMC8 Selector*** (PMC8SEL)

This field contains a code (one of at most 16 values) that identifies the event to be counted in PMC8; see Book IV.

29 ***Freeze Counters until IABR Match*** (FCUIABR)

0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented until a "monitored" IABR match occurs. An IABR match is said to be "monitored" if it occurs when PMC incrementing is permitted by $MMCR0_{0:4}$ and $MSR_{PR\ PMM}$. When a monitored IABR match occurs:
  - the PMCs resume incrementing (if permitted by other MMCR bits)
  - $MMCR1_{FCUIABR}$ is set to 0

The IABR (Instruction Address Breakpoint Register) is an implementation-specific SPR, and the definition of "IABR match" is implementation-dependent; see the Book IV, *PowerPC AS Implementation Features* document for the implementation.

30 ***PMC1 History Mode*** (PMC1HIST)

This bit controls whether PMC1 is incremented in the normal way, described in Section E.2.1, or in "history mode". In history mode a PMC is shifted left by one bit each processor cycle, and the vacated low-order bit is set to 1 if the associated event occurred

(one or more times) in that cycle and is set to 0 otherwise.

0 PMC1 is incremented normally (if incrementing is permitted by other MMCR bits).
1 PMC1 is incremented in history mode (if incrementing is permitted by other MMCR bits).

31 ***PMCj History Mode*** (PMCjHIST)

This bit controls whether all PMCjs are incremented in the normal way, described in Section E.2.1, or in "history mode", described under PMC1HIST above.

0 All PMCjs are incremented normally (if incrementing is permitted by other MMCR bits).
1 All PMCjs are incremented in history mode (if incrementing is permitted by other MMCR bits).

E.2.4 Monitor Mode Control Register A

Monitor Mode Control Register A (MMCRA) is a 32-bit register. This register, along with MMCR0 and MMCR1, controls the operation of the Performance Monitor.

Figure 42. Monitor Mode Control Register A

The bit definitions of MMCRA are as follows. MMCRA bits that are not implemented are treated as reserved.

Bit(s) Description

0 ***Multithread Count Mode*** (MODE)
0 Global Mode: All PMCs count all threads (no thread active gating)
Example: If MMCR0 is programmed to have PMC1 count instructions executed, PMC1 will count instructions executed by both thread 0 and 1.
1 Thread Mode: PMC1 - PMC4 count events tor thread 0. PMC5-8 count the same events for thread 1.
Example: If MMCR0 is programmed to have PMC1 count instructions executed, PMC1 will count instructions executed both thread 0, and and PMC5 will count instruction executed by thread 1.

When MODE = 1, the PMC SPR addressing changes.

- For thread 0, PMC1 - PMC4 (Performance Monitor Counter registers 1 - 4) are addressed using PMC1 - PMC4 SPR addresses from Figure 38 on page 107. The results of ***mfspr*** or ***mtspr*** instructions that use a PMC5 - PMC8 SPR address are implementation-dependent.
- For thread 1, PMC5 - PMC8 are addressed using PMC1 - PMC4 SPR addresses from Figure 38 on page 107. The results of ***mfspr*** or ***mtspr*** instructions that use a PMC5-8 SPR address are implementation-dependent.

1 ***Freeze Counters 1-4*** (FC1-4)
0 PMC1 - PMC4 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented

2 ***Freeze Counters 5-8*** (FC5-8)
0 PMC5 - PMC8 are incremented (if permitted by other MMCR bits).
1 PMC1 - PMC4 are not incremented

3-7 Reserved

8-14 Reserved for implementation-specific use

15 ***External Performance Monitor Exception*** (EPMX)

Set to 1 if an External Performance Monitor Exception is received. This bit can be set to 0 only by the ***mtspr*** instruction. Software should set this bit to 0 after handling the external event.

16 ***External Performance Monitor Exception Enable*** (EPMXE)
0 External Performance Monitor exceptions are disabled.
1 External Performance Monitor exceptions are enabled.

An external signal can be driven by other components in the system to signal an exception when one or more of their counters has its most significant bit set to 1.

17:23 Reserved

24:27 Reserved for implementation-specific use

28 ***Freeze Counters in Tags Inactive Mode*** (FCTI)
0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented in *tags inactive* mode.

This bit is a basic feature.

29 ***Freeze Counters in Tags Active Mode*** (FCTA)
0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented in *tags active* mode.

This bit is a basic feature.

30 ***Freeze Counters in Wait State*** (FCWAIT)
0 The PMCs are incremented (if permitted by other MMCR bits).
1 The PMCs are not incremented if $CTRL_{31}=0$. Software is expected to set $CTRL_{31}=0$ when it is in a *wait state*, i.e. there is no process ready to run.

This bit is a basic feature.

Only Branch Unit type of events do not increment if $CTRL_{31}=0$. Other units continue to count.

31 Reserved

E.2.5 Sampled Instruction Address Register

The Sampled Instruction Address Register (SIAR) is a 64-bit register. It contains the address of the "sampled instruction" when a Performance Monitor exception occurs.

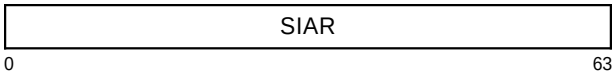


Figure 43. Sampled Instruction Address Register

When a Performance Monitor exception occurs, SIAR is set to the effective address of an instruction that was executing, possibly out-of-order, at or around the time that the Performance Monitor exception occurred. This instruction is called the "sampled instruction".

The contents of SIAR may be altered by the processor if and only if $MMCR0_{PMXE}=1$. Thus after the Performance Monitor exception occurs, the contents of SIAR are not altered by the processor until software sets $MMCR0_{PMXE}$ to 1. After software sets $MMCR0_{PMXE}$ to 1, the contents of SIAR are undefined until the next Performance Monitor exception occurs.

See Section E.4 regarding the effects of the optional Trace facility on SIAR.

Engineering Note

If the Performance Monitor exception is caused by an enabled counter negative condition that can be associated with the execution of a specific instruction, it is preferable to set SIAR to that instruction's address.

E.2.6 Sampled Data Address Register

The Sampled Data Address Register (SDAR) is a 64-bit register. It contains the address of the "sampled data" when a Performance Monitor exception occurs.

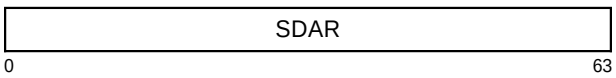


Figure 44. Sampled Data Address Register

When a Performance Monitor exception occurs, SDAR is set to the effective address of the storage operand of an instruction that was executing, possibly out-of-order, at or around the time that the Performance Monitor exception occurred. This storage operand is called the "sampled data". The sampled data may be, but need not be, the storage operand (if any) of the "sampled instruction" (see Section E.2.5). If the Performance Monitor exception causes a Performance Monitor interrupt, SRR1 indicates whether the sampled data is in fact the storage operand of the sampled instruction (see Section E.3).

The contents of SDAR may be altered by the processor if and only if $MMCR0_{PMXE}=1$. Thus after the Performance Monitor exception occurs, the contents of SDAR are not altered by the processor until software sets $MMCR0_{PMXE}$ to 1. After software sets $MMCR0_{PMXE}$ to 1, the contents of SDAR are undefined until the next Performance Monitor exception occurs.

See Section E.4 regarding the effects of the optional Trace facility on SDAR.

Engineering Note

If the sampled instruction has a storage operand, it is preferable to set SDAR to that storage operand's address.

E.3 Performance Monitor Interrupt

The Performance Monitor interrupt is a system-caused interrupt (see Section 7.3, "Interrupt Classes" on page 60). It is masked by MSR_{EE} in the same manner that External and Decrementer interrupts are.

A Performance Monitor interrupt occurs when no higher priority exception exists, a Performance Monitor exception exists, and $MSR_{EE}=1$. The occurrence of the interrupt cancels the exception (i.e., causes the exception to cease to exist).

If multiple Performance Monitor exceptions occur before the first causes a Performance Monitor interrupt, the interrupt reflects the most recent Performance Monitor exception and the preceding Performance Monitor exceptions are lost.

The following registers are set:

SRR0 Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.

SRR1

33 Set to 1 if the contents of SIAR and SDAR are associated with the same instruction (i.e., if SDAR contains the effective address of the storage operand of the "sampled instruction"); otherwise set to 0 (including the case in which the "sampled instruction" has no storage operand).

34:36 and 42:47 See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

Others Loaded from the MSR.

Engineering Note

SRR1 bits 34:36 and 42:47 can be used to provide information about the state of the processor at the time the "sampled instruction" was being executed or at the time the Performance Monitor exception is generated.

MSR See Figure 30 on page 62.

SIAR Set to the effective address of the "sampled instruction" (see Section E.2.5).

SDAR Set to the effective address of the "sampled data" (see Section E.2.6).

Execution resumes at effective address 0x0000_0000_0000_0F00.

In general, statements about External and Decrementer interrupts elsewhere in this Book apply also to the Performance Monitor interrupt; for example, if a Performance Monitor exception is pending when an ***mtmsr*** or ***mtmsrd*** instruction is executed that changes MSR_{EE} from 0 to 1, the Performance Monitor interrupt will occur before the next instruction is executed (if no higher priority exception exists).

The priority of the Performance Monitor interrupt is between that of the External interrupt and that of the Decrementer interrupt (see Section 7.7.2, "Ordered Exceptions" on page 73 and Section 7.8, "Interrupt Priorities" on page 73).

E.4 Interaction with the Trace Facility

If the Trace facility includes setting SIAR and SDAR (see Appendix F, "Example Trace Extensions (Optional)" on page 115), and tracing is active ($MSR_{SE}=1$ or $MSR_{BE}=1$), the contents of SIAR and SDAR as used by the Performance Monitor facility are undefined and may change even when $MMCR0_{PMXE}=0$, and the contents of $SRR1_{33}$ when a Performance Monitor interrupt occurs are also undefined.

Programming Note

A potential combined use of the Trace and Performance Monitor facilities is to trace the control flow of a program and simultaneously count events for that program.

E.5 Synchronization Requirements for Registers

Any requirements for synchronizing the effect of loading performance monitor registers is implementation-dependent.

Appendix F. Example Trace Extensions (Optional)

This appendix provides an example of extensions that may be added to the optional Trace facility described in Section 7.5.13, “Trace Interrupt” on page 71. It is only an example; implementations may provide all, some, or none of the features described here, or may provide features that are similar to those described here but differ in detail. See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

The extensions consist of the following features (described in detail below).

- use of $MSR_{SE\ BE}$=0b11 to specify new causes of Trace interrupts
- specification of how certain SRR1 bits are set when a Trace interrupt occurs
- setting of SIAR and SDAR (see Appendix E) when a Trace interrupt occurs

$MSR_{SE\ BE}$ = 0b11

If $MSR_{SE\ BE}$=0b11, the processor generates a Trace exception under the conditions described in Section 7.5.13 for $MSR_{SE\ BE}$=0b01, and also after successfully completing the execution of any instruction that would cause at least one of SRR1 bits 33:36, 42, and 44:46 to be set to 1 (see below) if the instruction were executed when $MSR_{SE\ BE}$=0b10.

This overrides the implicit statement in Section 7.5.13 that the effects of $MSR_{SE\ BE}$=0b11 are the same as those of $MSR_{SE\ BE}$=0b10.

SRR1

When a Trace interrupt occurs, the SRR1 bits that are not loaded from the MSR are set as follows instead of as described in Section 7.5.13.

33 Set to 1 if the traced instruction is ***icbi***; otherwise set to 0.

34 Set to 1 if the traced instruction is ***dcbt, dcbtst, dcbz, dcbst,*** or ***dcbf***; otherwise set to 0.

35 Set to 1 if the traced instruction is a *Load* instruction or ***eciwx***; may be set to 1 if the traced instruction is ***icbi, dcbt, dcbtst, dcbst,*** or ***dcbf***; otherwise set to 0.

36 Set to 1 if the traced instruction is a *Store* instruction, ***dcbz,*** or ***ecowx***; otherwise set to 0.

42 Set to 1 if the traced instruction is ***lswx*** or ***stswx***; otherwise set to 0.

43 See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

44 Set to 1 if the traced instruction is a *Branch* instruction and the branch is taken; otherwise set to 0.

45 Set to 1 if the traced instruction is ***eciwx*** or ***ecowx***; otherwise set to 0.

46 Set to 1 if the traced instruction is ***lwarx, ldarx, stwcx.,*** or ***stdcx.***; otherwise set to 0.

47 See the Book IV, *PowerPC AS Implementation Features* document for the implementation.

Engineering Note

The setting of bit 44 as specified above is not expected to be provided on implementations that fold branches.

Bits 43 and 47 can be used to provide information about the state of the processor at the time the traced instruction was being executed.

SIAR and SDAR

If the optional Performance Monitor facility is implemented and includes SIAR and SDAR (see Appendix E, “Example Performance Monitor (Optional)” on page 105), the following additional registers are set when a Trace interrupt occurs:

SIAR Set to the effective address of the traced instruction.

SDAR Set to the effective address of the storage operand (if any) of the traced instruction; otherwise undefined.

If the state of the Performance Monitor is such that the Performance Monitor may be altering these registers (i.e., if $MMCR0_{PMXE}$=1), the contents of SIAR and SDAR as used by the Trace facility are undefined and may change even when no Trace interrupt occurs.

Engineering Note

On an implementation for which the Performance Monitor permits the number of instructions completed between successive Trace interrupts to be counted exactly, the setting of SIAR as described above is not needed.

It is acceptable for SDAR not to be set as specified above under certain conditions (e.g., for a *Storage Access* instruction that causes a Data Storage interrupt).

Appendix G. PowerPC AS Operating Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Priv.[2]	Page	Mnemonic	Instruction
	Primary	Extend					
X	31	83		P	21	mfmsr	Move From Machine State Register
XFX	31	339		O	20	mfspr	Move From Special Purpose Register
X	31	595	32	P	92	mfsr	Move From Segment Register
X	31	659	32	P	92	mfsrin	Move From Segment Register Indirect
X	31	146		P	87	mtmsr	Move To Machine State Register
X	31	178		P	21	mtmsrd	Move To Machine State Register Doubleword
XFX	31	467		O	19	mtspr	Move To Special Purpose Register
X	31	210	32	P	91	mtsr	Move To Segment Register
X	31	242	32	P	91	mtsrin	Move To Segment Register Indirect
XL	19	18		P	13	rfid	Return From Interrupt Doubleword
XL	19	82	TA	P	12	rfscv	Return From System Call Vectored
SC	17	1			11	sc	System Call
SC	17	0	TA		12	scv	System Call Vectored
X	31	498		P	52	slbia	SLB Invalidate All
X	31	434		P	51	slbie	SLB Invalidate Entry
X	31	915		P	54	slbmfee	SLB Move From Entry ESID
X	31	851		P	54	slbmfev	SLB Move From Entry VSID
X	31	402		P	53	slbmte	SLB Move To Entry
X	31	370		P	56	tlbia	TLB Invalidate All
X	31	306	64	H	55	tlbie	TLB Invalidate Entry
X	31	566		H	56	tlbsync	TLB Synchronize

[1]Key to Mode Dependency Column

† Except as described below and in the section entitled † "Effective Address Calculation" in Book I, all † instructions in the PowerPC AS Operating Environ- † ment Architecture are independent of whether the † processor is in 32-bit or 64-bit mode and of whether † the processor is in *tags active* or *tags inactive* mode.

† TA The instruction can be executed only in *tags active* mode. In *tags inactive* mode the instruction is an illegal instruction.

32 The instruction must be executed only in 32-bit mode.

64 The instruction must be executed only in 64-bit mode.

[2]Key to Privilege Column

P denotes a privileged instruction.

O denotes an instruction that may be treated as privileged or nonprivileged (or hypervisor, for ***mtspr***), depending on the SPR number.

H denotes an instruction that can be executed only in hypervisor state.

Index

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Numerics

Last Page - End of Document