PowerPC AS User Instruction Set Architecture

Book I

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 PowerPC AS User Instruction Set Architecture. It covers the base instruction set and related facilities available to the application programmer.

Other related documents define the PowerPC AS Virtual Environment Architecture, the PowerPC AS Operating Environment Architecture, and PowerPC AS Implementation Features. Book II, *PowerPC AS Virtual Environment Architecture* defines the storage model and related instructions and facilities available to the application programmer, and the time-keeping facilities available to the application programmer. Book III, *PowerPC AS Operating Environment Architecture* defines the system (privileged) instructions and related facilities. 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.

Engineering Note

The PowerPC AS Architecture permits implementation-specific extensions to the architecture to be defined in Book IV. This Note provides guidelines and limitations on the features that are permitted to be defined in that book. Any exceptions to the guidelines and limitations must be approved in advance by the PowerPC AS Architecture process.

To understand the terminology used in this Note it may be necessary to refer to Book II and Book III. In particular, the term “privileged state” means a processor state in which nearly all resources of the architecture are accessible (typically the state in which operating systems run) and the term “problem state” means a processor state in which “privileged” resources are not accessible (typically the state in which application programs run). (A few resources are accessible only in “hypervisor state”; see the section entitled “Logical Partitioning (LPAR)” in Book III.)

- The only architecture resources (e.g., opcodes, SPR numbers, interrupt vector locations, bits in defined registers and in defined storage tables) that may be used for implementation-specific differences or extensions are those explicitly identified in Book I, II, or III as reserved for implementation-specific use.
- It is imperative that fragmentation of the software base be avoided. Application software must be able to run without change on all implementations. Operating system software that obeys the programming model described in Book III must run without change on implementations that claim to conform to Book III. Any difference or extension that is likely to fragment the software base is prohibited. Examples include but are not limited to the following.
  — Features, including instructions and registers, that are accessible in problem state.
  — Mechanisms that control whether a feature is accessible in problem state.
  — Privileged features, including instructions and registers, that provide functions useful primarily to application software.
- It is permissible to provide a privileged control mechanism that can be used to alter the behavior of a defined feature for use in performing infrequent operations associated with system initialization and the like. An example is a control mechanism that causes a TLB invalidation instruction to interpret an operand as specifying the physical TLB entry to be invalidated, enabling software to invalidate all TLB entries during system initialization.
- Any implementation-specific 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; see the Book III section cited above.

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

Incomplete as of 1999/02/24

topic	reason	page
Additional programming examples should be added to Section C.2, Floating-Point Conversions.		180

Changes as of 1999/02/24 Version 2.00

change	reason	page
Make the following changes. ■ Add the "TH" field for use in the ***dcbt*** instruction. ■ In Section 3.3.15 make a minor change in wording dealing with extended mnemonics for consistency with other sections. ■ In Section 5.2 remove the statement that the instruction is optional in each instruction description.	RFC02000.	10, 13, 95, 140-141
Make the following changes. ■ Clarify that software is permitted to write any value to reserved bits in System Registers unless otherwise stated. ■ Remove E=R and E=DS addressing and Direct-Store Errors from the architecture.	RFC02001.	3, 6, 29, 189
Make the following changes. ■ Move the *Storage Synchronization* instructions descriptions to Book II. ■ Add the "L" field for use in the ***sync*** instruction. ■ Remove the ***vsync*** instruction from the architecture. ■ Remove a paragraph in the Programming Note in Section 4.4 concerning the use of an execution synchronizing instruction. ■ Make various minor changes for consistency in wording in Section B.5. ■ Move the "Synchronization" programming examples to Book II. ■ Show instructions deleted from the architecture in parentheses in Appendix I. ■ Clarify the "Key to Mode Dependency Column" in Appendix K.	RFC02002.	4, 10, 12, 22, 57, 109, 145, 161, 166, 177, 183, 185, 189, 195, 197, 208+1, 213, 216ff, 221+1ff

change	reason	page
Make the following changes. ■ Remove mention of FP Unavailable exceptions from Book I. ■ Remove FP Assist exception from the architecture. ■ Clarify which bytes of a storage operand have an EAO associated with them.	RFC02004.	3, 6, 16, 17+1
Make the following changes. ■ Make minor changes in Section 1.5.3. ■ Replace the Segment Table with a software-managed SLB. ■ Add new instructions to move to/from the SLB (***slbmte***, ***slbmfev***, and ***slbmfee***). ■ Remove the ***mtsrd***, ***mtsrdin***, and ***rfi*** instructions from the architecture ■ Redefine the ***tlbie*** "S" field to be an "L" field. ■ Show instructions deleted from the architecture in parentheses in Appendix I. ■ In Appendix K define a 32-bit and 64-bit mode dependency.	RFC02005, except that in Appendix I parentheses were placed around the opcode, instead of around the mnemonic, to reduce the likelihood that wide mnemonics overlay mnemonics in adjacent columns. In addition the following changes were made. ■ In the definitions of ***mulhw***[***u***] and ***divw***[***u***], for consistency with usage elsewhere "$(RT)_{0:31}$ are undefined" was changed to "The contents of $RT_{0:31}$ are undefined". The definitions of ***fctiw***[***z***] and ***mffs*** were changed similarly. ■ The title and body of Section 5.3.6 were modified to omit reference to lookaside buffers and storage tables, because the changes made by this RFC to the *Lookaside Buffer Management* instructions are such that these instructions do not specify an EA. ■ The ***rfi*** entry added in Sections E.29 and E.30 was flagged with a "(*)" to show the instruction is privileged. ■ To avoid confusion with parentheses around an opcode to indicate that the instruction is no longer defined, the parentheses (used as a separator) around ***lq*** in Appendix I were changed to a "/".	4-5, 10, 12-13, 15, 65-67, 131, 134, 146, 148, 186, 189-191, 201ff, 215ff, 223ff

change	reason	page
Make the following changes. ■ Add a Branch Hint (BH) field to the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions. ■ Replace the branch "y" bit hint and some of the "z" bits with "at" bit hints. ■ Eliminate ambiguous use of the term "condition" in Sections 2.4.1 and B.2.	RFC02006 and Correspondence of 19 March '99, except that the RFC's proposed change to add ",0" to the extended mnemonics in the description of the *Branch Conditional* instruction in Section 2.4.1 was not made because the "at" value is specified in the BO field (not as a separate operand) in the basic mnemonic. In addition the following changes were made. ■ The second sentence of the definition of the Link Register in Section 2.3.2 was reworded slightly to eliminate the term "*Branch and Link*", which is nowhere defined. ■ ***bclr***[***l***] and ***bcctr***[***l***] instruction descriptions: — A sentence was added to the first paragraph mentioning the BH field. — As for other such cases, a Programming Note was added stating that ***bclr, bclrl, bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic, and an example of this was added under "Extended Mnemonics" for each of the two instructions. A similar Note was added to Section B.2. ■ In the Compatibility Notes in Sections 2.4.2, 3.2.2, 3.3.5, and 3.3.15, and in the Programming Note in Section 5.3.3.1, "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. ■ In the Compatibility Note in Section 3.2.2, the bit range was corrected. ■ The new wording of Section B.2.1 was clarified in several respects. ■ In Section B.2.3 the second sentence of the second paragraph was corrected with respect to the BH field and was moved to the end of the paragraph, and the BH value was added for the basic mnemonic in the last example. ■ In Section B.2.4 the statement that a suffix can be added to *any* mnemonic was corrected (not all BO encodings have "at" bits).	10, 11, 23-28, 34+1, 162-164, 185, 186

change	reason	page
Make the following changes for LPAR. ■ Add a new privilege state, hypervisor state. ■ Redefine the ***sc*** instruction to include the LEV field, which can be used to call the hypervisor. Remove statements in Book I concerning 32-bit *tags active* mode being undefined. Also, correct a statement in Section E.5 concerning POWER's MSR_{24}, which corresponds to the US bit in PowerPC AS.	RFC02007. In addition the following changes were made. ■ For consistency with the change made to Section 1.12.2, a related sentence in the definitions of CIA and NIA in Section 1.5.3 was deleted. ■ For consistency with the change made to the SC-form, the reference to the *System Call Vectored* instruction was removed from the LK description in Section 1.7.18. ■ For consistency with a change for the Compatibility Note in Section 2.4.2, “this architecture” was changed to “the PowerPC AS Architecture” in the Compatibility Notes in Sections 3.2.2 and 3.3.5. ■ In Appendix K the definition of “TA” was reworded slightly for consistency with changes made in the “Key” definitions in Book III by this RFC and RFC02005.	iii, 5, 9, 12, 17, 18, 29, 34+1, 51, 185, 186+1, 188, 225
Remove firm consistency from the architecture.	RFC02009.	192
Remove the ***dcba*** and ***dcbi*** instructions from the architecture.	RFC02010, except that the proposed bullet was not added to Section E.21 because permitting ***dcbz*** to fetch the block from main storage does not affect the migratibility of software from POWER to PowerPC AS.	5, 208ff, 217ff, 221
Make the following changes. ■ Allow the system data storage error handler to be invoked for a ***lq*** or ***stq*** that accesses Write Through Required or Caching Inhibited storage. ■ Change “system error handler” to “system data storage error handler” in Section 3.3.1.2. ■ Clarify aspects in which PowerPC AS is incompatible with POWER for *Load Multiple*, *Store Multiple*, and *Move Assist* instructions. ■ Remove the requirement to load DAR and DSISR for Alignment interrupts.	RFC02011. In addition the following changes were made. ■ Mention of the fact that ***lq*** and ***stq*** can cause the system alignment error handler to be invoked if they access Write Through Required or Caching Inhibited storage was added to Section 3.3.1.2. ■ In Section 4.6.1.1 “system error handler” was changed to “system data storage error handler”, for consistency with this RFC's change to Section 3.3.1.2.	36, 117, 187, 189

change	reason	page
Make the following changes. ■ Define optional new versions of the ***mtcrf*** and ***mfcr*** instructions to operate on one and only one CR field. ■ Specify a preferred form of the *Condition Register Logical* instructions. ■ For the ***mcrf*** instruction, clarify that CR field BF is altered, not the entire CR.	RFC02012 and Correspondence of 21 Dec. '98, except that in the ***mcrf*** instruction description under the subheading “Special Registers Altered” “CR” was replaced by “CR field BF”. In addition the following changes were made. ■ The definition of the FXM field in Section 1.7.18 was reworded slightly, to avoid implying that ***mfcr*** is optional. ■ The last sentence of the last Engineering Note with the ***mtspr*** instruction description was deleted (referred to “different Book III”). ■ The preferred form for ***mtcrf*** was handled in a manner consistent with other preferred forms: — An entry for it was added in Section 1.9.1. — Description of the preferred form was added to the introduction to Section 3.3.15. — The Programming Note in the ***mtcrf*** instruction description was shortened and corrected. ■ In the description of the extended mnemonic for ***mtcrf*** in the introduction to Section 3.3.15, “old software” was clarified. ■ An Engineering Note was added to the description of ***mtcrf*** and ***mfcr*** in Section 3.3.15, pointing the designers to the description of the optional version of these instructions. ■ In the ***mfcr*** instruction format in Section 3.3.15, bits 12:15 and 16:20 were merged into a single reserved field. ■ Under “Special Registers Altered” for ***mcrxr*** and ***mcrxrt***, “CR” was changed to “CR field BF”. ■ Section 5.1: — A 3-level section header was added for consistency with Section 5.2, and an introductory sentence was added to clarify that the instructions are versions of non-optional instructions. — The first two sentences of the first paragraph of the instruction descriptions were modified to cover the case of an all-zero FXM field. — To reduce repetition, the Programming Notes for the two instructions were combined into a single Programming Note, and similarly for the Engineering Notes and Architecture Notes, and the Notes were clarified in various respects. In particular, in the Architecture Note the statement about configurability of Northstar processors was weakened to be an expectation. — An Assembler Note was added to explain how the Assembler should determine whether to generate the old forms of the instructions or the new. ■ The ***mfcr*** entry in Appendix J and Appendix K was modified to show the “Form” as “XFX”.	10, 12, 14, 30-32, 95, 97, 137-139, 185, 216, 223

change	**reason**	**page**
Clarify restrictions related to changing Endian mode for ***scv***.	RFC02013.	148

change	reason	page
Make the following changes. ■ Make Little-Endian optional. ■ Make $FPSCR_{NI}$ a reserved bit. ■ Clarify the programming model to avoid EAO exceptions. ■ Delete tagged pointer support for the ***lmd*** instruction. ■ Clarify tag bit support for the data cache. ■ Clarify that XER_{OC} is set to an undefined value by the *Subtract From Immediate Carrying* and *Subtract From Carrying* instructions. ■ Remove deviations for pre-Version 2.00 processors.	RFC02014, with the following exceptions. ■ The SC-form changes to Section 1.7.3 proposed by RFC02007 were used in lieu of those proposed by RFC02014. ■ Material that was moved from the old Little-Endian appendix (Appendix C) to Section 5.3 has change bars only where where it is modified. In addition the following changes were made. ■ In Section 1.7.18 the definitions of the following fields were reworded slightly, for consistency with other definitions in the section: BD, D, DS, DQ, LI, IB, IS, and PT. For all but the last three the change is too minor to warrant a change bar. ■ For clarity, in the first sentence of Section 1.12.2 “storage” was changed to “main storage”. ■ Overflow Carry (OC) bit: — In Section 3.2.2 the setting of the OC bit for *Subtract From Carrying* type of instructions was changed to “set to an undefined value”, for consistency with the change in this RFC for the ***subfic*** and ***subfc*** instruction descriptions. — Under “Special Registers Altered” in the ***subfic*** and ***subfc*** instruction descriptions “(set to undefined value)” was abbreviated to “(undefined)”, for consistency with the treatment of FPRF in Chapter 4. ■ EAO exceptions: — In Sections 3.3.1 and 4.6.1, the paragraph citing the Programming Note on page 6 was made a separate Programming Note because it has nothing to do with the ***la*** extended mnemonic. — The second sentence of the first Programming Note in Section 3.3.1 was abbreviated, and its details were moved to be the first paragraph of the new Programming Note for ***la*** in Section B.11. The new second sentence was also added to the ***la*** Programming Note in Section 4.6.1. — The (now) second paragraph of the Programming Note in Section B.11 was clarified. ■ Section 5.3: — For consistency with references to the section elsewhere, “Byte Ordering” was omitted from the section title rather than being changed to “Storage Addressing”. — The new first paragraph of the section was made more consistent with the introductions to other “optional” sections. ■ A reference to MXU in Appendix I was deleted. ■ “Amazon” was changed to “PowerPC AS” throughout the Book, without change bars.	6, 16-18, 34, 36, 44, 48, 49, 51, 53, 61, 99, 103, 117, 120, 137, 142, 175, 193, 208

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
Delete the statement in Section 3.3.6 that *Load/Store String Indexed* instructions follow the rule for preferred forms for *Load/Store Indexed* instructions since there are no longer such preferred forms.	out of date statement	53
Add new instruction field for ***tlbie***, and new incompatibilities between POWER and PowerPC AS for ***tlbie***.	RFC00248 as rewritten by Correspondence of 9 Dec. '97. In addition the following changes were made. ■ 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 in Book III to represent the $\log_2$ of the page size.) ■ The last sentence of the third paragraph of Section 1.7 was deleted because it is obsolete; the section of Book III that described the Book-III-only fields was deleted in Version 1.09, and Book II never had such a section.	10, 13, 189

Changes as of 1998/03/27 Version 1.06

change	reason	page
State that, in general, optional facilities and instructions are described in chapters, appendices, and sections for which the title contains the word “Optional”.	RFC00246. The order of “facilities” and “instructions” in the new paragraph was reversed from that in the RFC, for consistency with the rest of the Architecture Note. In addition, for correctness, “if necessary” was added near the end of the last paragraph under “Category 1”.	15
State that facilities and instructions in optionality category 2 *generally* appear in a separate chapter.	RFC00245 as amended at June PAWG meeting.	15
Removed statement that the Trace facility is optional in PowerPC AS.	Amazon RFC 365	192

change	reason	page
Add Process Local Storage (PLS) architecture: ■ Modify the definition of XER_{OC} to account for PLS ■ Modify the definitions of ***cmpla***, ***td*** and ***tdi*** to detect PLS boundary cross. ■ In *tags active* mode ***tw*** and ***twi*** with TO=11100 are invalid forms. ■ Remove *tags active* mode option for 24-bit effective address addition for both instructions and data.	Amazon RFC 347	5-6, 18ff, 34, 70-73
Listed Northstar deviations.	Amazon RFC 346	old “Deviations” app.
Add missing “At” to *Select* instructions in Table 15 on page 204.	Amazon RFC 344	204
Add missing parentheses in ***lq*** RTL.	Amazon RFC 338	43
Clarify that ***scv*** and ***rfscv*** still have implementation-dependent requirements when switching Endian modes.	Amazon RFC 337	148
Remove EAO X-form implementation-dependent option for 24-bit add.	Amazon RFC 334	5
Make instruction address EA calculations boundedly undefined in *tags active* mode when the 64-bit result and CIA are different address types: PLS, SLS, E=R, or E=DS. If also in Privileged mode, the result is undefined.	Amazon RFC 330	6
Remove TG term for XER_{TAG}.	Amazon RFC 327	44
Modify PowerPC RFC00220 to E.23 to qualify a statement about optional direct-store segments with *tags inactive* mode.	Amazon RFC 323	189
Remove statement about extended opcodes for opcode 0 being assigned to various companies.	Amazon RFC 319	199
Correct ***stq*** RTL to state that all tag bits in the QW are set.	Amazon RFC 315	48
Define ***sync*** to not be context synchronizing with respect to Direct-Store Errors	Amazon RFC 308	old ***sync*** def.
Add miscellaneous changes including: ■ Add list of *tags active* instructions to Appendix G. ■ Correct bit numbering for MSR_{FC} in E.31.3. ■ Remove “?” symbol's meaning of implementation-dependent value. ■ State that for n=0, ***stsdx*** stores no bytes. ■ Eliminate redundant statements about RB containing the smaller operand for multiplies. ■ Correct statement that if any tag bits in a QW are 0, the QW is “untagged”.	Amazon RFC 306	4, 17, 43, 57, 64ff, 192, 197
State COBRA 4 problem with ***settag*** or ***mtxer*** immediately following ***stq***.	Amazon RFC 305	48

change	**reason**	**page**
Delete Matrix Unit from the architecture.	Amazon RFC 302	old Matrix Unit sect.
Make PowerPC optional instructions ***fsqrt***, ***fsqrts***, ***fres***, ***frsqrte***, and ***fsel*** optional in PowerPC AS.	Amazon RFC 299	126, 133, 137ff, old “Deviations” app.
Revise COBRA 4 deviation list for hardware problems: undefined SRR0 and SRR1 if Direct-Store Error interrupt collides with other interrupts, MSR_{US} and MSR_{FC} set incorrectly when switching from *tags active* to *tags inactive* mode, invalid forms of load string operations, and ***icbi***	Amazon RFC 298	old “Deviations” app.
Revise MUSKIE deviation list for hardware problems: requirement for interrupt code to do Load or Store before floating-point instruction, ***dcbi***, MSR_{MM}, W bit aliasing, E=DS instruction fetch, data cache parity error, precise mode restriction, minimizing time with $MSR_{EE}=0$, and ***dcbtst***.	Amazon RFC 297	old “Deviations” app.
Add APACHE deviation list.	Amazon RFC 296	old “Deviations” app.
Update COBRA-Lite deviation list for new bugs, $MMCR0_{10:11}$, W bit aliasing, and Little-Endian.	Amazon RFC 295	old “Deviations” app.
Remove deviation lists for early processor passes and remove ***tlbiex*** and ***slbiex*** from the architecture.	Amazon RFC 294	old “Deviations” app.
Add ***mtmsrd*** and ***rfid*** to COBRA-Lite deviation list.	Amazon RFC 236	old “Deviations” app.
Make minor corrections related to instruction fields, DS-form, optional instructions, illegal instructions, and POWER vs. POWER2	RFC00231 as amended at Oct. PAWG meeting.	2, 13, 15, 36, 193, 197
Reduce use of “instruction storage” and “data storage”.	RFC00242 Correspondence of 14 Nov. '96.	3, 16, 145-146
Make minor corrections related to FPSCR exception summary bits.	RFC00230.	16, 101, 108-109, 134
Redefine ***sync*** to make it a memory barrier.	RFC00233 and Correspondence of 7 Nov. '96. In addition, for consistency with similar wording added elsewhere by the RFC, “that processor” was used instead of “that other processor” near the end of the first paragraph of the Programming Note in the ***sync*** instruction description.	old “Stg. Synch. Instrs.” sect., old ***sync*** def.-58, 134, old synch. prog. examples sect., 189, 192
Reserve SPRs for implementation-specific uses.	RFC00167 as rewritten by Correspondence of 30 May '96, as amended at Oct. PAWG meeting.	95, 188

change	reason	page
Correct an error in the lock acquisition programming example.	Error Notice of 5 Dec. '96.	old synch. prog. examples sect.
Amplify differences from POWER2 regarding Trace.	RFC00223 as rewritten by Correspondence of 19 Nov. '96.	192
Reserve opcodes for implementation-specific uses.	RFC00225 as amended at Oct. PAWG meeting.	201ff
Specify what can be defined in Book IV and in non-AIM Books II and III.	RFC00224 as amended at March PAWG meeting. In addition, “for implementation-specific differences or extensions” was inserted into the first bullet, instead of “for implementation-specific extensions” as agreed at the meeting, for reasons given in mail from Cathy May 19 April '96, and the Engineering Note was placed after the boldface “Note”, instead of before it as proposed in the RFC, for readability and page layout.	iii
Correct several minor errors.	Error Notices of 3 May (two Notices) and 6 May '96, except that the correction proposed for Section E.31.2 was not made, because the current terminology was deemed acceptable and the affected sentence appears also in Sections E.21 and E.25.	2, 11+1, 34, 58, 71, 97, 101, 109, 113, 114, 117, 120, 121, 133, 134, 177ff, 181, 186, 188
Add one Engineering Note about reserved bits, and revise another.	RFC00213 and Correspondence of 23 March '96, as amended at March PAWG meeting.	4, 187
Describe why various facilities and instructions are optional.	RFC00218 and Correspondence of 10 April '96. The “Optional Instructions” appendix was made a chapter, as agreed at the March PAWG meeting; this necessitated changing “appendix” to “chapter” in several places.	13, 15, 102, 137ff, 180, 182
Eliminate reference to Book III sentence that RFC deletes.	RFC00226.	old ***sync*** def.
Start to phase direct-store out of the architecture.	RFC00220.	143, 189
Delete Programming Note about unaligned Little-Endian storage accesses.	RFC00177 and Correspondence of 23 March '96.	145
Add new *Cache Management* instruction *Data Cache Block Allocate* (***dcba***).	RFC00228 and Correspondence of 10 May '96.	201+8, 215+3, 221
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
Relax rules for hardware's handling of reserved bits in registers.	RFC00195.	3, 185

change	**reason**	**page**
Correct several minor errors.	Error Notice of 27 Oct. '94, Book I items 1-11. (Item 12 is done as part of RFC 173.)	11, 13, 22, 80, 84, 106, 110, 117, 120, 141, 183+1, 186
Make 64-bit MMU functions an extension of 32-bit MMU functions.	RFC00178 as rewritten by Correspondence of 24 Oct. '94.	15, 189, 201ff
In the first two sentences of the ***sync*** description, omit the word “given” (three occurrences).	RFC00199 and Correspondence of 25 Oct. '94.	old ***sync*** def.
Use “performed” vs. “executed” consistently for loads and stores.	RFC00205.	old ***sync*** def.
Clarify that $CR0_{0:2}$ is undefined for certain instructions in 64-bit mode.	RFC00194.	65-67
Clarify meaning of floating-point “intermediate result”.	RFC00185 as amended at Nov. PAWG meeting. The RFC says to number bits G, R, and X in Figure 36 on page 113, but this looked too crowded so G and R are not numbered (there is no ambiguity about the lengths of these fields).	106-107, 112-114
Correct the definition of rounding.	RFC00198 as amended at Nov. PAWG meeting.	101-102, 106-107, 113-114, 129, 132, 181
Clarify descriptions of Underflow and Inexact Exceptions.	RFC00186.	110, 112-112
Say that ***rfi*** and interrupts change Endian mode reliably for I-fetch.	RFC00181. The wording of the Engineering Note for p. 148 has been revised slightly to include ***rfid*** and ***mtmsrd***, which were added by RFC00178. The last sentence has been reworded slightly to match similar wording used in RFC00203.	148
Delete text that suggests using all 64 bits of GPRs when in 32-bit mode.	RFC00173. In addition, the second sentence of the second paragraph of the section introduction (p. 177) has been deleted because it refers to material deleted by the RFC, and it is not in the Morgan Kaufmann book.	177, 177+1
Note additional POWER incompatibility for *Store Floating-Point Single*.	RFC00196.	188
Delete “17” from the list of primary opcodes that have unused extended opcodes.	RFC00173.	197
Define “AIM” and use “-AIM” suffix on citations as needed.	RFC00203.	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

This chapter describes computation modes, compatibility with the POWER Architecture, document conventions, a processor overview, instruction formats, storage addressing, and instruction fetching.

1.2 Computation Modes

The PowerPC AS Architecture requires a 64-bit implementation, in which all registers except some Special Purpose Registers are 64 bits long and effective addresses are 64 bits long. All 64-bit implementations have two modes of operation: 64-bit mode and 32-bit mode. The mode controls how the effective address is interpreted, how status bits are set, and how the Count Register is tested by *Branch Conditional* instructions. All instructions are available in both modes. In both 64-bit mode and 32-bit mode, instructions that set a 64-bit register affect all 64 bits, and the value placed into the register is independent of mode. In both modes, effective address computations use all 64 bits of the relevant registers (General

Purpose Registers, Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored when accessing data and are set to 0 when fetching instructions.

The PowerPC AS Architecture does not permit an implementation that provides *only* the equivalent of 32-bit mode (i.e., an implementation in which all registers except Floating-Point Registers are 32 bits long).

1.3 Instruction Mnemonics and Operands

The description of each instruction includes the mnemonic and a formatted list of operands. Some examples are the following.

stw	RS,D(RA)
addis	RT,RA,SI

PowerPC AS-compliant Assemblers will support the mnemonics and operand lists exactly as shown. They should also provide certain extended mnemonics, as described in Appendix B, “Assembler Extended Mnemonics” on page 161.

1.4 Compatibility with the POWER Architecture

The PowerPC AS Architecture provides binary compatibility for POWER application programs, except as described in Appendix E, “Incompatibilities with the POWER Architecture” on page 185.

Many of the PowerPC AS instructions are identical to POWER instructions. For some of these the PowerPC AS instruction name and/or mnemonic differs from that in POWER. To assist readers familiar with the POWER Architecture, POWER mnemonics are shown with the individual instruction descriptions when they differ from the PowerPC AS mnemonics. Also, Appendix D, “Cross-Reference for Changed POWER Mnemonics” on page 183 provides a cross-reference from POWER mnemonics to PowerPC AS mnemonics for the instructions in Books I, II, and III.

References to the POWER Architecture include POWER2 implementations of the POWER Architecture unless otherwise stated.

1.5 Document Conventions

1.5.1 Definitions and Notation

The following definitions and notation are used throughout the PowerPC AS Architecture documents.

- A program is a sequence of related instructions.
- Octwords are 256 bits, quadwords are 128 bits, doublewords are 64 bits, words are 32 bits, halfwords are 16 bits, and bytes are 8 bits.
- All numbers are decimal unless specified in some special way.
  - 0bnnnn means a number expressed in binary format.
  - 0xnnnn means a number expressed in hexadecimal format.

  Underscores may be used between digits.
- RT, RA, R1, ... refer to General Purpose Registers.
- FRT, FRA, FR1, ... refer to Floating-Point Registers.
- (x) means the contents of register x, where x is the name of an instruction field. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields. Names such as LR and CTR denote registers, not fields, so parentheses are not used with them. Parentheses are also omitted when register x is the register into which the result of an operation is placed.
- (RA|0) means the contents of register RA if the RA field has the value 1-31, or the value 0 if the RA field is 0.
- Bits in registers, instructions, and fields are specified as follows.
  - Bits are numbered left to right, starting with bit 0.
  - Ranges of bits are specified by two numbers separated by a colon (:). The range p:q consists of bits p through q.
- X_p means bit p of register/field X.
- $X_{p:q}$ means bits p through q of register/field X.
- $X_{p\ q\ \ldots}$ means bits p, q, ... of register/field X.
- $\neg$(RA) means the one's complement of the contents of register RA.
- Field i refers to bits $4\times i$ through $4\times i+3$ of a register.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of certain Special Purpose Registers as a side effect of exe-

cution, as described in Chapter 2 through Chapter 4.

- The symbol || is used to describe the concatenation of two values. For example, 010 || 111 is the same as 010111.
- x^n means x raised to the nth power.
- ${}^n x$ means the replication of x, n times (i.e., x concatenated to itself n−1 times). ${}^n 0$ and ${}^n 1$ are special cases:
  - — ${}^n 0$ means a field of n bits with each bit equal to 0. Thus ${}^5 0$ is equivalent to 0b00000.
  - — ${}^n 1$ means a field of n bits with each bit equal to 1. Thus ${}^5 1$ is equivalent to 0b11111.
- Positive means greater than zero.
- Negative means less than zero.
- A system library program is a component of the system software that can be called by an application program using a *Branch* instruction.
- A system service program is a component of the system software that can be called by an application program using a *System Call* instruction.
- The system trap handler is a component of the system software that receives control when the conditions specified in a *Trap* instruction are satisfied.
- The system error handler is a component of the system software that receives control when an error occurs. The system error handler includes a component for each of the various kinds of error. These error-specific components are referred to as the system alignment error handler, the system data storage error handler, etc.
- Each bit and field in instructions, and in status and control registers (XER and FPSCR) and Special Purpose Registers, is either defined or reserved.
- /, //, ///, ... denotes a reserved field in an instruction.
- Latency refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction.
- Unavailable refers to a resource that cannot be
† used by the program. For example, storage is
† unavailable if access to it is denied. See Book III, *PowerPC AS Operating Environment Architecture*.
- The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary between implementations, and between different executions on the same implementation, and are not further defined in this document.
- The sequential execution model is the model of program execution described in Section 2.2, “Instruction Fetching” on page 21.

1.5.2 Reserved Fields

All reserved fields in instructions should be zero. If they are not, the instruction form is invalid: see Section 1.9.2, “Invalid Instruction Forms” on page 14.

The handling of reserved bits in System Registers (e.g., XER, FPSCR) is implementation-dependent.
† Unless otherwise stated, software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

Programming Note

It is the responsibility of software to preserve bits that are now reserved in System Registers, as they may be assigned a meaning in some future version of the architecture.

In order to accomplish this preservation in implementation-independent fashion, software should do the following.

- Initialize each such register supplying zeros for all reserved bits.
- Alter (defined) bit(s) in the register by reading the register, altering only the desired bit(s), and then writing the new value back to the register.

The XER and FPSCR are partial exceptions to this recommendation. Software can alter the status bits in these registers, preserving the reserved bits, by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined bit in the FPSCR by executing a *Floating-Point Status and Control Register* instruction. Using such instructions is likely to yield better performance than using the method described in the second item above.

When a currently reserved bit is subsequently assigned a meaning, every effort will be made to have the value to which the system initializes the bit correspond to the “old behavior”.

Engineering Note

Reserved bits in System Registers need not be implemented.

1.5.3 Description of Instruction Operation

A formal description is given of the operation of each instruction. In addition, the operation of most instructions is described by a semiformal language at the register transfer level (RTL). This RTL uses the notation given below, in addition to the definitions and notation described in Section 1.5.1, “Definitions and Notation” on page 2. Some of this notation is also used in the formal descriptions of instructions. RTL notation not summarized here should be self-explanatory.

The RTL descriptions cover the normal execution of the instruction, except that “standard” setting of the Condition Register, Fixed-Point Exception Register, and Floating-Point Status and Control Register are not shown. (“Non-standard” setting of these registers, † such as the setting of the Condition Register by the † *Compare* instructions, is shown.) The RTL descriptions do not cover cases in which the system error handler is invoked, or for which the results are boundedly undefined.

The RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

The RTL is written for implementations that have a tag bit per doubleword or a tag bit per quadword (i.e. a doubleword or quadword tag block), although other tag block sizes are permitted.

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_{iea}$	† Assignment of an instruction effective address. In 32-bit mode the high-order 32 bits of the 64-bit target address are set to 0. †
¬	NOT logical operator
+	Two's complement addition
$+_{tia}$	In *tags active* mode, special rules apply to instruction address addition (see 1.5.3.2, “Tags Active Mode $+_{tia}$” on page 6). In *tags inactive* mode, this notation means two's complement 64-bit addition.
$+_{tea}$	In *tags active* mode, special rules apply to effective address addition (see below). In *tags inactive* mode, this notation means two's complement 64-bit addition.
–	Two's complement subtraction, unary minus
×	Multiplication
÷	Division (yielding quotient)
√	Square root
=, ≠	Equals, Not Equals relations
<, ≤, >, ≥	Signed comparison relations
$\overset{u}{<}$, $\overset{u}{>}$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format, using the model shown on page 117
EXTS(x)	Result of extending x on the left with sign bits
FLOOR(x)	Greatest integer ≤ x
GPR(x)	General Purpose Register x
MASK(x, y)	Mask having 1s in positions x through y (wrapping if x > y) and 0s elsewhere

MEM(x, y)	Contents of y bytes of storage starting at address x. In 32-bit mode of a 64-bit implementation, the high-order 32 bits of the 64-bit value x are ignored.
$MEM_{tag}(x)$	Tag bit of the tag block (see “tag block” below) in storage that contains address x. In 32-bit mode of a 64-bit implementation the high-order 32 bits of the 64-bit value x are ignored.
$MEM_{tag}(x, y)$	Tag bits of all the tag blocks (see “tag block” below) that contain any of the y bytes of storage starting at address x. In 32-bit mode of a 64-bit implementation the high-order 32 bits of the 64-bit value x are ignored.
$ROTL_{64}(x, y)$	Result of rotating the 64-bit value x left y positions
$ROTL_{32}(x, y)$	Result of rotating the 64-bit value x||x left y positions, where x is 32 bits long
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format, using the model shown on page 120
SPREG(x)	Special Purpose Register x
tag block	An implementation-dependent quantity of storage containing 1, 2, 4, 8 or 16 bytes. The block is integrally aligned. For each tag block there is one tag bit in storage.
TRAP	Invoke the system trap handler
characterization	Reference to the setting of status bits, in a standard way that is explained in the text
tags active mode	See Book III, *PowerPC AS Operating Environment Architecture*.
tags inactive mode	See Book III, *PowerPC AS Operating Environment Architecture*.
undefined	An undefined value. The value may vary between implementations, and between different executions on the same implementation.
CIA	Current Instruction Address, which is the 64-bit address of the instruction being described by a sequence of RTL. Used by relative branches to set the Next Instruction Address (NIA), and by *Branch* instructions with LK=1 to set the Link Register. † In 32-bit mode, the high-order 32 bits of CIA are always set to 0. Does not correspond to any architected register.
NIA	Next Instruction Address, which is the 64-bit address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching (see Book III, *PowerPC AS Operating Environment Architecture*), the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA $+_{tia}$ 4. In 32-bit mode, the high-order 32 bits of NIA are † always set to 0. Does not correspond to any architected register.
if ... then ... else ...	Conditional execution, indenting shows range; else is optional
do	Do loop, indenting shows range. “To” and/or “by” clauses specify incrementing an iteration variable, and a “while” clause gives termination conditions.
leave	Leave innermost do loop, or do loop described in leave statement
for	For loop, indenting shows range. Clause after “for” specifies the entities for which to execute the body of the loop.

1.5.3.1 Tags Active Mode $+_{tea}$

In *tags active* mode, the effective address addition operator, $+_{tea}$, has two operands which are 64-bit numbers. The $+_{tea}$ operation involves a 64-bit add. With the exception of the ***dcbt*** and ***dcbtst*** instructions (see Book II, *PowerPC AS Virtual Environment Architecture* for details on these instructions), $+_{tea}$ can produce an Effective Address Overflow (EAO) exception. An EAO exception results in invocation of the system data storage error handler.

The following describes when an EAO exception occurs. For this description C_{40} is defined to be the carry out of bit position 40 assuming two 64-bit operands are added. C_{16} is defined to be the carry out of bit position 16. D_0 is the most significant bit of the displacement in a D-form, DS-form, or DQ-form instruction

- **D-form, DS-form, and DQ-form instructions**: There are several implementation-dependent options for detecting EAO exceptions. One of these must be implemented.

  — If $EA_{0:15} = 0$ & RA $\neq 0$ & ($C_{16} \oplus D_0$), then an EAO exception occurs. If $EA_{0:15} \neq 0$ & RA $\neq 0$ & ($C_{40} \oplus D_0$), then an EAO exception occurs.

  — If $(RA)_{0:15} = 0$ & RA $\neq 0$ & ($C_{16} \oplus D_0$), then an EAO exception occurs. If $(RA)_{0:15} \neq 0$ & RA $\neq 0$ & ($C_{40} \oplus D_0$), then an EAO exception occurs.

  — If $EA_{0:15} = 0$ & RA $\neq 0$ & $(RA)_{0:15} \neq EA_{0:15}$, then an EAO exception occurs. If $EA_{0:15} \neq 0$ & RA $\neq 0$ & $(RA)_{0:39} \neq EA_{0:39}$, then an EAO exception occurs.

- If $(RA)_{0:15} = 0$ & $RA \neq 0$ & $(RA)_{0:15} \neq EA_{0:15}$, then an EAO exception occurs. If $(RA)_{0:15} \neq 0$ & $RA \neq 0$ & $(RA)_{0:39} \neq EA_{0:39}$, then an EAO exception occurs.

- ■ **X-form instruction**: There are several implementation-dependent options for detecting EAO exceptions. One of these must be implemented.

  - If $EA_{0:15} = 0$ & $RA \neq 0$ & $(RA)_{0:15} \neq EA_{0:15}$, then an EAO exception occurs. If $EA_{0:15} \neq 0$ & $RA \neq 0$ & $(RA)_{0:39} \neq EA_{0:39}$, then an EAO exception occurs.
  - If $(RA)_{0:15} = 0$ & $RA \neq 0$ & $(RA)_{0:15} \neq EA_{0:15}$, then an EAO exception occurs. If $(RA)_{0:15} \neq 0$ & $RA \neq 0$ & $(RA)_{0:39} \neq EA_{0:39}$, then an EAO exception occurs.
  - If $EA_{0:15} = 0$ & $RA \neq 0$ & ($(RB)_{0:14} \neq {}^{15}(RB)_{15} \mid C_{16} \oplus (RB)_{15}$), then an EAO exception occurs. If $EA_{0:15} \neq 0$ & $RA \neq 0$ & ($(RB)_{0:38} \neq {}^{39}(RB)_{39} \mid C_{40} \oplus (RB)_{39}$), then an EAO exception occurs.
  - If $(RA)_{0:15} = 0$ & $RA \neq 0$ & ($(RB)_{0:14} \neq {}^{15}(RB)_{15} \mid C_{16} \oplus (RB)_{15}$), then an EAO exception occurs. If $(RA)_{0:15} \neq 0$ & $RA \neq 0$ & ($(RB)_{0:38} \neq {}^{39}(RB)_{39} \mid C_{40} \oplus (RB)_{39}$), then an EAO exception occurs.

† **Programming Note**

† In order that all implementations detect EAO
† exceptions correctly and do not cause unnec-
† essary EAO exceptions, if the base address is
† in RB then RA must be 0.

† ■ **Operand length**: An EAO exception occurs for
† byte j of a storage operand, $0 \leq j < n$ where n is
† the length of the operand, if any of the following
† conditions is true. EA is the effective address of
† the operand, and EAj is the effective address of
† byte j.

† — $RA \neq 0$ and either of the following is true.

† — $(RA)_{0:15} = 0$ and $(RA)_{0:15} \neq EAj_{0:15}$.

† — $(RA)_{0:15} \neq 0$ and $(RA)_{0:39} \neq EAj_{0:39}$.

† — $RA = 0$, the instruction has an RB field, and
† either of the following is true.

† — $(RB)_{0:15} = 0$ and $(RB)_{0:15} \neq EAj_{0:15}$.

† — $(RB)_{0:15} \neq 0$ and $(RB)_{0:39} \neq EAj_{0:39}$.

† — $RA = 0$, the instruction does not have an RB
† field, and $EA_0 \neq EAj_0$.

The effective address calculations for branches and sequential instruction fetching do not cause EAO exceptions.

1.5.3.2 Tags Active Mode $+_{tia}$

In *tags active* mode, the effective address calculations for branches and sequential instruction fetching is called $+_{tia}$. $+_{tia}$ has a right and left operand. Both operands are treated as 64-bit numbers. In the following situations, the result is boundedly undefined in *tags active* mode:

1. $CIA_{0:15}$=0x0000 and a 64-bit effective address calculation would have produced a resulting $NIA_{0:15} \neq$ 0x0000.
2. $CIA_{0:15} \neq$ 0x0000 and a 64-bit effective address calculation would have produced a resulting $NIA_{0:15}$=0x0000.

|

$+_{tia}$ does not cause EAO exceptions.

1.5.3.3 Precedence Rules

The precedence rules for RTL operators are summarized in Table 1. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, – associates from left to right, so a– b– c = (a– b)– c.) Parentheses are used to override the evaluation order implied by the table or to increase clarity: parenthesized expressions are evaluated before serving as operands.

Table 1. Operator precedence

Operators	Associativity
subscript, function evaluation	left to right
pre-superscript (replication), post-superscript (exponentiation)	*right to left*
unary – , ¬	*right to left*
×, ÷	left to right
$+$, $-$, $+_{tea}$	left to right
$\Vert$	left to right
$=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $\overset{u}{<}$, $\overset{u}{>}$, ?	left to right
&, ⊕ , ≡	left to right
$\vert$	left to right
: (range)	none
$\leftarrow$	none

1.6 Processor Overview

The processor implements the instruction set, the storage model, and other facilities defined in this document. Instructions that the processor can execute fall into three classes:

- branch instructions,
- fixed-point instructions, and
- floating-point instructions.

Branch instructions are described in Section 2.4, “Branch Processor Instructions” on page 24. Fixed-point instructions are described in Section 3.3, “Fixed-Point Processor Instructions” on page 36. Floating-point instructions are described in Section 4.6, “Floating-Point Processor Instructions” on page 116.

Fixed-point instructions operate on byte, halfword, word, and doubleword operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC AS Architecture uses instructions that are four bytes long and word-aligned. It provides for byte, halfword, word, doubleword, and quadword operand fetches and stores between storage and a set of 32 General Purpose Registers (GPRs). It also provides for word, doubleword, and quadword operand fetches and stores between storage and a set of 32 Floating-Point Registers (FPRs).

Signed integers are represented in two's complement form.

There are no computational instructions that modify storage. To use a storage operand in a computation and then modify the same or another storage location, the contents of the storage operand must be loaded into a register, modified, and then stored back to the target location. Figure 1 is a logical representation of instruction processing. Figure 2 on page 8 shows the registers of the PowerPC AS User Instruction Set Architecture.

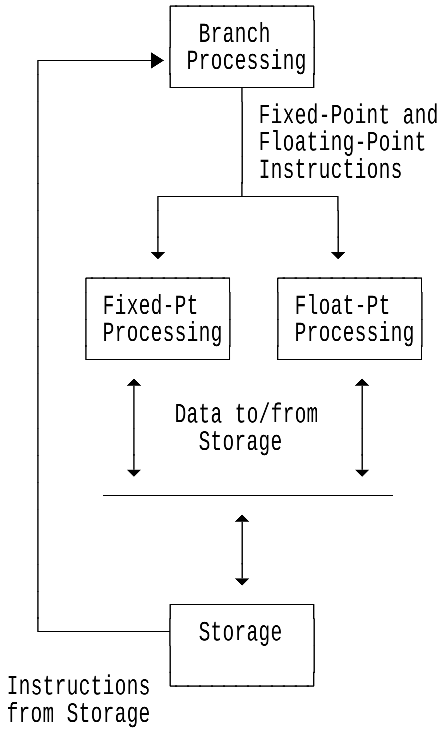

Figure 1. Logical processing model

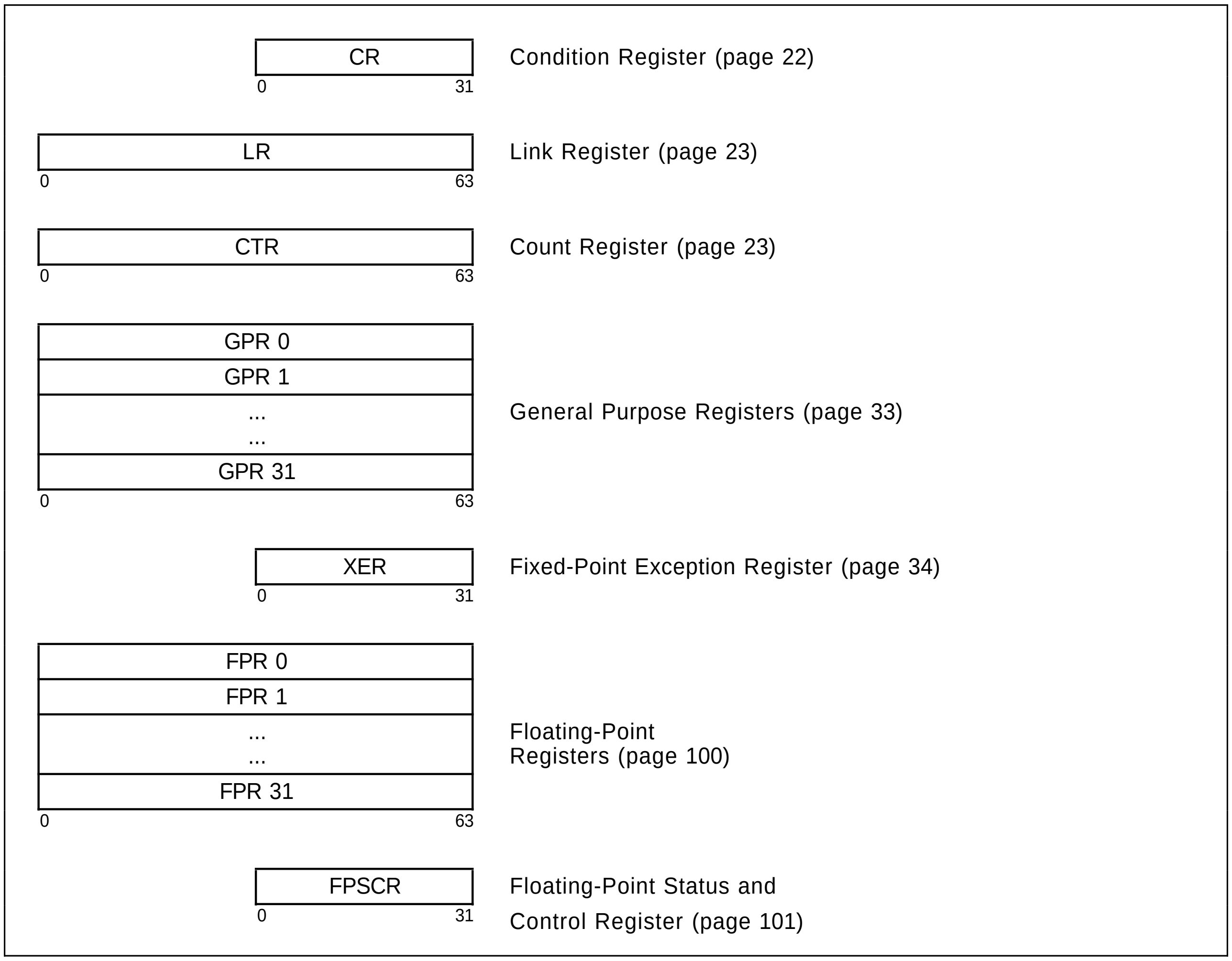
CR
0
31
Condition Register (page 22)
LR
0
63
Link Register (page 23)
CTR
0
63
Count Register (page 23)
GPR 0
GPR 1
...
...
GPR 31
0
63
General Purpose Registers (page 33)
XER
0
31
Fixed-Point Exception Register (page 34)
FPR 0
FPR 1
...
...
FPR 31
0
63
Floating-Point Registers (page 100)
FPSCR
0
31
Floating-Point Status and Control Register (page 101)

Figure 2. PowerPC AS user register set

1.7 Instruction Formats

All instructions are four bytes long and word-aligned. Thus, whenever instruction addresses are presented to the processor (as in *Branch* instructions) the low-order two bits are ignored. Similarly, whenever the processor develops an instruction address the low-order two bits are zero.

Bits 0:5 always specify the opcode (OPCD, below). Many instructions also have an extended opcode (XO, below). The remaining bits of the instruction contain one or more fields as shown below for the different instruction formats.

The format diagrams given below show horizontally all valid combinations of instruction fields. The diagrams include instruction fields that are used only by instructions defined in Book II, *PowerPC AS Virtual Environment Architecture*, or in Book III, *PowerPC AS Operating Environment Architecture*.

In some cases an instruction field is reserved, or must contain a particular value. If a reserved field does not have all bits set to 0, or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described in Section 1.9.2, “Invalid Instruction Forms” on page 14.

Split Field Notation

In some cases an instruction field occupies more than one contiguous sequence of bits, or occupies one contiguous sequence of bits that are used in permuted order. Such a field is called a *split field*. In the format diagrams given below and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the RTL description of an instruction having a split field, and in certain other places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. In all other places, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

1.7.1 I-Form

0	6	30	31
OPCD	LI	AA	LK

Figure 3. I instruction format

1.7.2 B-Form

0	6	11	16	30	31
OPCD	BO	BI	BD	AA	LK

Figure 4. B instruction format

1.7.3 SC-Form

0	6	11	16	20	27	30	31
OPCD	///	///	//	LEV	//	XO	1
OPCD	///	///	//	LEV	//	XO	/

Figure 5. SC instruction format

1.7.4 D-Form

0	6	11	16 31
OPCD	RT	RA	D
OPCD	RT	RA	SI
OPCD	RS	RA	D
OPCD	RS	RA	UI
OPCD	BF / L	RA	SI
OPCD	BF / L	RA	UI
OPCD	TO	RA	SI
OPCD	FRT	RA	D
OPCD	FRS	RA	D

Figure 6. D instruction format

1.7.5 DS-Form

0	6	11	16	30 31
OPCD	RT	RA	DS	XO
OPCD	RS	RA	DS	XO

Figure 7. DS instruction format

1.7.6 DQ-Form

0	6	11	16	28 31
OPCD	RT	RA	DQ	PT

Figure 8. DQ instruction format

1.7.7 X-Form

0	6	11	16	21	31
OPCD	RT	RA	RB	XO	/
OPCD	RT	RA	NB	XO	/
OPCD	RT	/ SR	///	XO	/
OPCD	RT	///	RB	XO	/
OPCD	RT	///	///	XO	/
OPCD	RS	RA	RB	XO	Rc
OPCD	RS	RA	RB	XO	1
OPCD	RS	RA	RB	XO	/
OPCD	RS	RA	NB	XO	/
OPCD	RS	RA	SH	XO	Rc
OPCD	RS	RA	///	XO	Rc
OPCD	RS	RA	///	XO	1
OPCD	RS	/ SR	///	XO	/
OPCD	RS	///	RB	XO	/
OPCD	RS	///	///	XO	/
OPCD	BF / L	RA	RB	XO	/
OPCD	BF / 1	RA	RB	XO	/
OPCD	BF //	FRA	FRB	XO	/
OPCD	BF //	BFA //	///	XO	/
OPCD	BF //	///	U /	XO	Rc
OPCD	BF //	///	///	XO	/
OPCD	/// TH	RA	RB	XO	/
OPCD	/// L	///	RB	XO	/
OPCD	/// L	///	///	XO	/
OPCD	TO	RA	RB	XO	/
OPCD	FRT	RA	RB	XO	/
OPCD	FRT	///	FRB	XO	Rc
OPCD	FRT	///	///	XO	Rc
OPCD	FRS	RA	RB	XO	/
OPCD	BT	///	///	XO	Rc
OPCD	///	RA	RB	XO	/
OPCD	///	///	RB	XO	/
OPCD	///	RA	///	XO	/
OPCD	///	///	///	XO	/

Figure 9. X instruction format

1.7.8 XL-Form

0	6	11	16	21	31
OPCD	BT	BA	BB	XO	/
OPCD	BO	BI	/// BH	XO	LK
OPCD	BF //	BFA //	///	XO	/
OPCD	///	///	///	XO	/

Figure 10. XL instruction format

1.7.9 XFX-Form

0	6	11	21	31
OPCD	RT	spr	XO	/
OPCD	RT	tbr	XO	/
OPCD	RT	0 ///	XO	/
OPCD	RT	1 FXM /	XO	/
OPCD	RS	0 FXM /	XO	/
OPCD	RS	1 FXM /	XO	/
OPCD	RS	spr	XO	/
OPCD	///	XO2	XO	/

Figure 11. XFX instruction format

1.7.10 XFL-Form

0	6	7	15	16	21	31
OPCD	/	FLM	/	FRB	XO	Rc

Figure 12. XFL instruction format

1.7.11 XS-Form

0	6	11	16	21	30	31
OPCD	RS	RA	sh	XO	sh	Rc

Figure 13. XS instruction format

1.7.12 XO-Form

0	6	11	16	21	22	31
OPCD	RT	RA	RB	OE	XO	Rc
OPCD	RT	RA	RB	/	XO	Rc
OPCD	RT	RA	///	OE	XO	Rc

Figure 14. XO instruction format

1.7.13 A-Form

0	6	11	16	21	26	31
OPCD	FRT	FRA	FRB	FRC	XO	Rc
OPCD	FRT	FRA	FRB	///	XO	Rc
OPCD	FRT	FRA	///	FRC	XO	Rc
OPCD	FRT	///	FRB	///	XO	Rc

Figure 15. A instruction format

1.7.14 M-Form

0	6	11	16	21	26	31
OPCD	RS	RA	RB	MB	ME	Rc
OPCD	RS	RA	SH	MB	ME	Rc

Figure 16. M instruction format

1.7.15 MD-Form

0	6	11	16	21	27	30	31
OPCD	RS	RA	sh	mb	XO	sh	Rc
OPCD	RS	RA	sh	me	XO	sh	Rc

Figure 17. MD instruction format

1.7.16 MDS-Form

0	6	11	16	21		27	31
OPCD	RS	RA	RB	mb		XO	Rc
OPCD	RS	RA	RB	me		XO	Rc
OPCD	IS	RA	IB	XBI	//	XO	Rc
OPCD	IS	RA	RB	XBI	//	XO	Rc
OPCD	RS	RA	IB	XBI	//	XO	Rc
OPCD	RS	RA	RB	XBI	//	XO	Rc

Figure 18. MDS instruction format

1.7.17 TX-Form

0	6	11	21	25	31
OPCD	TO	UI	XBI	XO	Rc

Figure 19. TX instruction format

1.7.18 Instruction Fields

AA (30)
Absolute Address bit.

0 The immediate field represents an address relative to the current instruction address. For I-form branches the effective address of the branch target is the sum of the LI field sign-extended to 64 bits and the address of the branch instruction. For B-form branches the effective address of the branch target is the sum of the BD field sign-extended to 64 bits and the address of the branch instruction.

1 The immediate field represents an absolute address. For I-form branches the effective address of the branch target is the LI field sign-extended to 64 bits. For B-form branches the effective address of the branch target is the BD field sign-extended to 64 bits.

BA (11:15)
Field used to specify a bit in the CR to be used as a source.

BB (16:20)
Field used to specify a bit in the CR to be used as a source.

BD (16:29)
Immediate field used to specify a 14-bit signed two's complement branch displacement which is concatenated on the right with 0b00 and sign-extended to 64 bits.

BF (6:8)
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a target.

BFA (11:13)
Field used to specify one of the CR fields or one of the FPSCR fields to be used as a source.

BH (19:20)
Field used to specify a hint in the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions. The encoding is described in Section 2.4.1, “Branch Instructions” on page 24.

BI (11:15)
† Field used to specify a bit in the CR to be tested
† by a *Branch Conditional* instruction.

BO (6:10)
Field used to specify options for the *Branch Conditional* instructions. The encoding is described in Section 2.4.1, “Branch Instructions” on page 24.

BT (6:10)
Field used to specify a bit in the CR or in the FPSCR to be used as a target.

D (16:31)
Immediate field used to specify a 16-bit signed two's complement integer which is sign-extended to 64 bits.

DS (16:29)
Immediate field used to specify a 14-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

DQ (16:27)
Immediate field used to specify a 12-bit signed two's complement integer which is concatenated on the right with 0b0000 and sign-extended to 64 bits.

FLM (7:14)
Field mask used to identify the FPSCR fields that are to be updated by the ***mtfsf*** instruction.

FRA (11:15)
Field used to specify an FPR to be used as a source.

FRB (16:20)
Field used to specify an FPR to be used as a source.

FRC (21:25)
Field used to specify an FPR to be used as a source.

FRS (6:10)
Field used to specify an FPR to be used as a source.

FRT (6:10)
Field used to specify an FPR to be used as a target.

FXM (12:19)
Field mask used to identify the CR fields that are
| to be updated by the ***mtcrf*** instruction or moved
| by the optional version of the ***mfcr*** instruction.

IB (16:20)
† Immediate field used to specify a 5-bit signed
† integer.

IS (6:10)
† Immediate field used to specify a 5-bit signed
† integer.

L (10)
Field used to specify whether a fixed-point *Compare* instruction is to compare 64-bit numbers or 32-bit numbers.

| Field used by the *Synchronize* instruction (see
| Book II, *PowerPC AS Virtual Environment Archi-*
| *tecture*).

| Field used by the *TLB Invalidate Entry* instruction
| (see Book III, *PowerPC AS Operating Environ-*
| *ment Architecture*).

LEV (20:26)
| Field used by the *System Call* instructions.

LI (6:29)
Immediate field used to specify a 24-bit signed two's complement integer which is concatenated on the right with 0b00 and sign-extended to 64 bits.

LK (31)
LINK bit.

0 Do not set the Link Register.

† 1 Set the Link Register. The address of the instruction following the *Branch* instruction is
† placed into the Link Register.

MB (21:25) and ME (26:30)
Fields used in M-form instructions to specify a 64-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive and 0-bits elsewhere, as described in Section 3.3.13, “Fixed-Point Rotate and Shift Instructions” on page 84.

MB (21:26)
Field used in MD-form and MDS-form instructions to specify the first 1-bit of a 64-bit mask, as described in Section 3.3.13, “Fixed-Point Rotate and Shift Instructions” on page 84.

ME (21:26)
Field used in MD-form and MDS-form instructions to specify the last 1-bit of a 64-bit mask, as described in Section 3.3.13, “Fixed-Point Rotate and Shift Instructions” on page 84.

NB (16:20)
Field used to specify the number of bytes to move in an immediate *Move Assist* instruction.

OPCD (0:5)
Primary opcode field.

OE (21)
Field used by XO-form instructions to enable setting OV and SO in the XER.

PT (28:31)
† Immediate field used to specify a 4-bit unsigned
† value.

RA (11:15)
Field used to specify a GPR to be used as a source or as a target.

RB (16:20)
Field used to specify a GPR to be used as a source.

Rc (31)
RECORD bit.

0 Do not alter the Condition Register.

1 Set Condition Register Field 0 or Field 1 as described in Section 2.3.1, “Condition Register” on page 22.

RS (6:10)
Field used to specify a GPR to be used as a source.

RT (6:10)
Field used to specify a GPR to be used as a target.

|

SH (16:20, or 16:20 and 30)
Field used to specify a shift amount.

SI (16:31)
Immediate field used to specify a 16-bit signed integer.

SPR (11:20)
Field used to specify a Special Purpose Register for the ***mtspr*** and ***mfspr*** instructions.

SR (12:15)
Field used by the *Segment Register Manipulation* instructions (see Book III, *PowerPC AS Operating Environment Architecture*).

TBR (11:20)
Field used by the *Move From Time Base* instruction (see Book II, *PowerPC AS Virtual Environment Architecture*).

| **TH (9:10)**
| Field used by the optional data stream variant of
| the ***dcbt*** instruction (see Book II, *PowerPC AS*
| *Virtual Environment Architecture*).

TO (6:10)
Field used to specify the conditions on which to trap. The encoding is described in Section 3.3.10, “Fixed-Point Trap Instructions” on page 71.

U (16:19)
Immediate field used as the data to be placed into a field in the FPSCR.

UI (11:20 or 16:31)
Immediate field used to specify a 16-bit unsigned integer.

XBI (21:24)
Field used to specify a bit in the XER.

† **XO (21:29, 21:30, 22:30, 25:30, 26:30, 27:29, 27:30, 30, or 30:31)**
Extended opcode field.

XO2 (11:20)
Second extended opcode field.

1.8 Classes of Instructions

An instruction falls into exactly one of the following three classes:

Defined
Illegal
Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

A given instruction is in the same class for all implementations of the PowerPC AS Architecture. In future versions of this architecture, instructions that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes described in Appendix H, “Reserved Instructions” on page 199). Similarly, instructions that are now reserved may become defined.

1.8.1 Defined Instruction Class

This class of instructions contains all the instructions defined in the PowerPC AS User Instruction Set Architecture, PowerPC AS Virtual Environment Architecture, and PowerPC AS Operating Environment Architecture.

In general, defined instructions are guaranteed to be provided in all implementations. The only exceptions are instructions that are optional instructions. These exceptions are identified in the instruction descriptions.

A defined instruction can have preferred and/or invalid forms, as described in Section 1.9.1, “Preferred Instruction Forms” on page 14 and Section 1.9.2, “Invalid Instruction Forms” on page 14.

1.8.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in Appendix G, “Illegal Instructions” on page 197. Illegal instructions are available for future extensions of the PowerPC AS Architecture: that is, some future version of the PowerPC AS Architecture may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction will cause the system illegal instruction error handler to be invoked and will have no other effect.

An instruction consisting entirely of binary 0's is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

Editors' Note

Instructions in this class were formerly called “invalid instructions”. The term was changed to “illegal instructions” to reduce confusion between these instructions and invalid *forms* of *defined* instructions.

1.8.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in Appendix H, “Reserved Instructions” on page 199.

Reserved instructions are allocated to specific purposes that are outside the scope of the PowerPC AS Architecture.

Any attempt to execute a reserved instruction will:

- perform the actions described in Book IV, *PowerPC AS Implementation Features* for the implementation if the instruction is implemented; or
- cause the system illegal instruction error handler to be invoked if the instruction is not implemented.

1.9 Forms of Defined Instructions

1.9.1 Preferred Instruction Forms

Some of the defined instructions have preferred forms. For such an instruction, the preferred form will execute in an efficient manner, but any other form may take significantly longer to execute than the preferred form.

Instructions having preferred forms are:

- the *Condition Register Logical* instructions
- the *Load/Store Multiple* instructions
- the *Load/Store String* instructions
- the *Or Immediate* instruction (preferred form of no-op)
- the *Move To Condition Register Fields* instruction

1.9.2 Invalid Instruction Forms

Some of the defined instructions have invalid forms. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

Any attempt to execute an invalid form of an instruction will either cause the system illegal instruction error handler to be invoked or yield boundedly undefined results. Exceptions to this rule are stated in the instruction descriptions.

Some kinds of invalid form can be deduced from the instruction layout. These are listed below.

- Field shown as “/”(s) but coded as nonzero.
- Field shown as containing a particular value but coded as some other value.

These invalid forms are not discussed further.

Instructions having invalid forms that cannot be so deduced are listed below. These kinds of invalid form are identified in the instruction descriptions.

- the *Branch Conditional* instructions
- the *Load/Store with Update* instructions
- the *Load Multiple* instructions
- the *Load String* instructions
- *Trap on XER* (***txer***)
- the *Load/Store Floating-Point with Update* instructions
- the *Load Quadword* (***lq***) and *Store Quadword* (***stq***) instructions
- the *Trap Word* (***tw***) and *Trap Word Immediate* (***twi***) instructions
- the *Set XER TAG* (***settag***) instruction

Assembler Note

To the extent possible, the Assembler should report uses of invalid instruction forms as errors.

Engineering Note

Causing the system illegal instruction error handler to be invoked if attempt is made to execute an invalid form of an instruction facilitates the debugging of software.

1.10 Optionality

Some of the defined instructions are optional. The optional instructions are defined in Chapter 5, "Optional Facilities and Instructions" on page 137. Additional optional instructions may be defined in Books II and III (e.g., see the section entitled "Look- † aside Buffer Management" in Book III, and the chapters entitled "Optional Facilities and Instructions" in Book II and Book III).

Any attempt to execute an optional instruction that is not provided by the implementation will cause the system illegal instruction error handler to be invoked.

In addition to instructions, other kinds of optional facilities, such as registers, may be defined in Books II and III. The effects of attempting to use an optional facility that is not provided by the implementation are described in Books II and III as appropriate.

Architecture Note

In general, optional facilities and instructions are described in chapters, appendices, and sections for which the title contains the word "Optional".

A facility or instruction is optional for any one of the following reasons.

1. It is being phased into the architecture. At some future date it will be required and no longer optional.
2. It is being phased out of the architecture. System developers should develop a migration plan to eliminate use of it in new systems.
3. It is useful primarily for certain kinds of applications and systems. It is likely to remain in the architecture, as optional.

Categories 1 and 2 permit the architecture to evolve gradually, by providing an intermediate status for facilities and instructions that are being added to or removed from the architecture. Category 3 is intended for facilities and instructions that are typically used primarily in library routines.

The category that a given optional facility or instruction is in can be identified as follows. The prototypical Notes and text shown below are altered as needed for each specific case.

Category 1

The description of each facility or instruction in this category contains an Engineering Note, the wording of which depends on how new the facility or instruction is. When the facility or instruction is first added to the architecture, the wording is similar to the following.

> **Engineering Note:**
>
> This instruction is being phased into the architecture, and will become required in a future version of the architecture.

Subsequently, when a version number "n.mm" of the architecture has been determined such that processors being designed to comply with other aspects of that version will implement the facility or instruction, the wording is changed to be similar to the following.

> **Engineering Note:**
>
> This instruction is being phased into the architecture, and must be implemented in processors that comply with Version n.mm of the architecture specification or with any subsequent version.

When the facility or instruction later becomes required, its description will be moved to the body of the Book if necessary, and the Engineering Note will be removed.

Category 2

The facilities and instructions in this category generally appear in a separate chapter. A prominent warning such as the following appears in the chapter introduction.

> **Warning:** The facilities and instructions described in this chapter are being phased out of the architecture.

Also, the description of each such facility or instruction contains a Programming Note and an Engineering Note similar to the following.

> **Programming Note:**
>
> **Warning:** This instruction is being phased out of the architecture. It is likely to perform poorly on future implementations. New programs should not use it.
>
> **Engineering Note:**
>
> Decisions regarding whether to implement this instruction in a given implementation, and how well to make it perform there, must include consideration of migration plans for existing software that uses it.

Category 3

The facilities and instructions in this category are identified by the absence of the distinguishing marks of the other two categories.

1.11 Exceptions

There are two kinds of exception, those caused directly by the execution of an instruction and those caused by an asynchronous event. In either case, the exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- an attempt to execute an illegal instruction, or an attempt by an application program to execute a "privileged" instruction (see Book III, *PowerPC AS Operating Environment Architecture*) (system illegal instruction error handler or system privileged instruction error handler)
- the execution of a defined instruction using an invalid form (system illegal instruction error handler or system privileged instruction error handler)
- the execution of an optional instruction that is not provided by the implementation (system illegal instruction error handler)
- an attempt to access a storage location that is unavailable (system instruction storage error handler or system data storage error handler)
- an attempt to access storage in a manner that causes Effective Address Overflow as described by the $+_{tea}$ operator on page 5 (system data storage error handler)
- an attempt to access storage with an effective address alignment that is invalid for the instruction (system alignment error handler)
- the execution of a *System Call* instruction (system service program)
- the execution of a *Trap* instruction that traps (system trap handler)

†

- the execution of a floating-point instruction that causes a floating-point enabled exception to exist (system floating-point enabled exception error handler)

|

The exceptions that can be caused by an asynchronous event are described in Book III, *PowerPC AS Operating Environment Architecture.*

The invocation of the system error handler is precise, except that if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see page 109) then the invocation of the system floating-point enabled exception error handler may be imprecise. When the system error handler is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the system error handler, has not yet occurred).

Additional information about exception handling can be found in Book III, *PowerPC AS Operating Environment Architecture*.

1.12 Storage Addressing

A program references storage using the effective address computed by the processor when it executes a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*), or when it fetches the next sequential instruction.

1.12.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Storage operands may be bytes, halfwords, words, doublewords, or quadwords, or, for the *Load/Store Multiple* and *Move Assist* instructions, a sequence of bytes, words, or doublewords. The address of a storage operand is the address of its first byte (i.e., of its lowest-numbered byte). Byte ordering is Big-Endian. However, if the optional Little-Endian facility is implemented the system can be operated in a mode in which byte ordering is Little-Endian; see Section 5.3.

Operand length is implicit for each instruction.

The operand of a single-register *Storage Access* instruction, or of a quadword *Load* or *Store* instruction, has a "natural" alignment boundary equal to the operand length. In other words, the "natural" address of an operand is an integral multiple of the operand length. A storage operand is said to be *aligned* if it is aligned at its natural boundary: otherwise it is said to be *unaligned*.

Storage operands for single-register *Storage Access* instructions have the following characteristics. (Although not permitted as storage operands, octwords are shown because octword alignment is desirable for certain storage operands.)

Operand	Length	$\text{Addr}_{59:63}$ if aligned
Byte	8 bits	xxxxx
Halfword	2 bytes	xxxx0
Word	4 bytes	xxx00
Doubleword	8 bytes	xx000
Quadword	16 bytes	x0000
Octword	32 bytes	00000
Note: An "x" in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.		

The concept of alignment is also applied more generally, to any datum in storage. For example, a 12-byte datum in storage is said to be word-aligned if its address is an integral multiple of 4.

Some instructions require their storage operands to have certain alignments. In addition, alignment may affect performance. For single-register *Storage Access* instructions, and for quadword *Load* and *Store* instructions, the best performance is obtained when storage operands are aligned. Additional effects of data placement on performance are described in Book II, *PowerPC AS Virtual Environment Architecture*.

Instructions are always four bytes long and word-aligned.

1.12.2 Tag Bits

† A *tag bit* is associated with each tag block in main storage and in the data cache (see Book II, *PowerPC AS Virtual Environment Architecture*). If all tag bits in a quadword are 1, the quadword is said to be "tagged", and if any tag bits in a quadword are 0 the
† quadword is said to be "untagged". Main storage and
† the data cache implement at least one tag bit per
† quadword. Main storage supplies tag bits to and
† accepts them from the data cache.

To simplify discussion, it is sometimes convenient to describe tag bits operation as if there were a single tag bit per quadword. Thus, sometimes the term "quadword tag bit" is used to describe the logical AND of all tag bits in the quadword or to describe setting all the tag bits in the quadword to a single value.

† Tag bits are intended for use in *tags active* mode only.
† When stores are performed in *tags inactive* mode, the
† tag of all affected tag blocks in storage is set to 0.

If storage is modified by mechanisms that do not maintain tag bits, the tag of all affected tag blocks in storage is set to 0. An example of such a mechanism is an I/O device that stores directly into the processor's storage.

Programming Note

Tag bits are intended to indicate whether an address has been constructed or validated by the operating system. A value of 1 for a tag bit is intended to mean that the associated value is an address that has been so constructed or validated, while a value of 0 for a tag bit is intended to mean that the associated value has not been so constructed or validated.

1.12.3 Effective Address Calculation

The 64-bit or 32-bit address computed by the processor when executing a *Storage Access* or *Branch* instruction (or certain other instructions described in Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*), or when fetching the next sequential instruction, is called the *effective address* and specifies a byte in storage.

1.12.3.1 Tags Inactive Mode Effective Address Calculation

In general, effective address computations, for both data and instruction accesses, use 64-bit effective address addition. Thus all 64 bits participate, regardless of mode (32-bit or 64-bit). The 64-bit current instruction address and next instruction address are not affected by a change from 32-bit mode to 64-bit mode, but they are affected by a change from 64-bit mode to 32-bit mode (the high-order 32 bits are set to 0).

In 64-bit mode, the entire 64-bit result comprises the 64-bit effective address. The effective address arithmetic wraps around from the maximum address, $2^{64}-1$, to address 0.

In 32-bit mode, the low-order 32 bits of the 64-bit result comprise the effective address for the purpose of addressing storage. The high-order 32 bits of the 64-bit effective address are ignored for the purpose of accessing data, but are included whenever a 64-bit effective address is placed into a GPR by *Load with Update* and *Store with Update* instructions. The high-order 32 bits of the 64-bit effective address are set to 0 for the purpose of fetching instructions, and whenever a 64-bit effective address is placed into the Link Register by *Branch* instructions having LK=1. The high-order 32 bits of the 64-bit effective address are set to 0 in Special Purpose Registers when the system error handler is invoked. As used to address storage, the effective address arithmetic appears to wrap around from the maximum address, $2^{32}-1$, to address 0 in *tags inactive* mode.

A zero in the RA field indicates the absence of the corresponding address component. For the absent component, a value of zero is used for the address. This is shown in the instruction descriptions as (RA|0).

†

Effective addresses are computed as follows. In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bit contents, independent of mode, but that in 32-bit mode only bits 32:63 of the 64-bit result of the computation are used to address storage.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ***lswi*** and ***stswi***) are added to the contents of the GPR designated by RA or to zero if RA=0.
- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added to the contents of the GPR designated by RA or to zero if RA=0.
- With I-form *Branch* instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the *Branch* instruction to form the effective address of the next instruction. If AA=1, this address component is the effective address of the next instruction.
- With B-form *Branch* instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the *Branch* instruction to form the effective address of the next instruction. If AA=1, this address component is the effective address of the next instruction.
- With XL-form *Branch* instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the next instruction.
- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction.

†

1.12.3.2 Tags Active Mode Effective Address Calculation

In general, effective address computations, for data accesses, use 64-bit *tags active* mode effective address addition as defined by the $+_{tea}$ operator (see page 5). The entire 64-bit result comprises the 64-bit effective address. Effective address addition for instructions also uses a 64-bit result, which is sometimes produced from addition.

†

A zero in the RA field indicates the absence of the corresponding address component. For the absent component, a value of zero is used for the address. This is shown in the instruction descriptions as (RA|0).

†

Effective addresses are computed as follows. Additional implementation options for *tags active* mode are given for load/store operands in the $+_{tea}$ definition (see page 5). In the descriptions below, it should be understood that “the contents of a GPR” refers to the entire 64-bit contents.

- With X-form instructions, in computing the effective address of a data element, the contents of the GPR designated by RB (or the value zero for ***lswi***, ***lsdi***, ***stswi***, and ***stsdi***) is added according to the rules of $+_{tea}$ to the contents of the GPR designated by RA or to zero if RA=0.
- With D-form instructions, the 16-bit D field is sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added according to the rules of $+_{tea}$ to the contents of the GPR designated by RA or to zero if RA=0.
- With DQ-form instructions, the 12-bit DQ field is concatenated on the right with 0b0000 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added according to the rules of $+_{tea}$ to the contents of the GPR designated by RA or to zero if RA=0.
- With DS-form instructions, the 14-bit DS field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. In computing the effective address of a data element, this address component is added according to the rules of $+_{tea}$ to the contents of the GPR designated by RA or to zero if RA=0.
- With I-form *Branch* instructions, the 24-bit LI field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the *Branch* instruction to form the effective address of the next instruction. If

AA=1, this address component is the effective address of the next instruction.

- With B-form *Branch* instructions, the 14-bit BD field is concatenated on the right with 0b00 and sign-extended to form a 64-bit address component. If AA=0, this address component is added to the address of the *Branch* instruction to form the effective address of the next instruction. If AA=1, this address component is the effective address of the next instruction.
- With XL-form *Branch* instructions, bits 0:61 of the Link Register or the Count Register are concatenated on the right with 0b00 to form the effective address of the next instruction.
- With sequential instruction fetching, the value 4 is added to the address of the current instruction to form the effective address of the next instruction.

Effective address calculations for branches and sequential instruction fetching do not cause EAO exceptions.

For instructions that refer to more than one byte of storage, the effective address for each byte after the first is computed by adding 1 to the effective address of the preceding byte. This addition follows the rules of $+_{tea}$.

Chapter 2. Branch Processor

2.1 Branch Processor Overview

This chapter describes the registers and instructions that make up the Branch Processor facility. Section 2.3, “Branch Processor Registers” on page 22 describes the registers associated with the Branch Processor. Section 2.4, “Branch Processor Instructions” on page 24 describes the instructions associated with the Branch Processor.

2.2 Instruction Fetching

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- *Branch* instructions for which the branch is taken cause execution to continue at the target address specified by the *Branch* instruction.
- *Trap* instructions for which the trap conditions are satisfied, and *System Call* instructions, cause the appropriate system handler to be invoked.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.11, “Exceptions” on page 16.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.
- For sequential instruction fetching in *tags active* mode, if $CIA_{40:63}$ = 0xFFFFFC then the next instruction address is implementation-dependent and can be either CIA + 4 or $CIA_{0:39}$ || 0x000000. Typically, the operating system will prevent this situation from arising.

The model of program execution in which each instruction appears to complete before the next instruction starts is called the “sequential execution model”. In general, from the view of the processor executing the instructions, the sequential execution model is obeyed. For the instructions and facilities defined in this Book, the only exceptions to this rule are the following.

- A floating-point exception occurs when the processor is running in one of the Imprecise floating-point exception modes (see Section 4.4, “Floating-Point Exceptions” on page 108). The instruction that causes the exception does not complete before the next instruction starts, with respect to setting exception bits and (if the exception is enabled) invoking the system error handler.
- A *Store* instruction modifies a storage location that contains an instruction. Software synchronization is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction: see Book II, *PowerPC AS Virtual Environment Architecture*.

Programming Note

If a program modifies the instructions it intends to execute, it should call the appropriate system library program before attempting to execute the modified instructions, to ensure that the modifications have taken effect with respect to instruction fetching.

2.3 Branch Processor Registers

2.3.1 Condition Register

The Condition Register (CR) is a 32-bit register which reflects the result of certain operations, and provides a mechanism for testing (and branching).

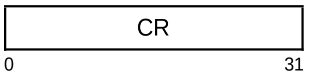

Figure 20. Condition Register

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (***mtcrf***).
- A specified field of the CR can be set by a move to the CR from another CR field (***mcrf***), from $XER_{32:35}$ (***mcrxr***), 0b0 || $XER_{41:43}$ (***mcrxrt***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- A specified CR field can be set as the result of either a fixed-point or a floating-point *Compare* instruction.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

For all fixed-point instructions in which Rc=1, and for ***addic., andi.***, and ***andis.***, the first three bits of CR Field 0 (bits 0:2 of the Condition Register) are set by signed comparison of the result to zero, and the fourth bit of CR Field 0 (bit 3 of the Condition Register) is copied from the SO field of the XER. "Result" here refers to the entire 64-bit value placed into the target register in 64-bit mode, and to bits 32:63 of the 64-bit value placed into the target register in 32-bit mode.

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if      (target_register)$_{M:63}$ < 0 then c ← 0b100
else if (target_register)$_{M:63}$ > 0 then c ← 0b010
else                                    c ← 0b001
CR0 ← c || XER$_{SO}$
```

If any portion of the result is undefined, then the value placed into the first three bits of CR Field 0 is undefined.

The bits of CR Field 0 are interpreted as follows.

Bit	Description
0	***Negative*** (LT) The result is negative.
1	***Positive*** (GT) The result is positive.
2	***Zero*** (EQ) The result is zero.
3	***Summary Overflow*** (SO) This is a copy of the final state of XER_{SO} at the completion of the instruction.

Programming Note

CR Field 0 may not reflect the "true" (infinitely precise) result if overflow occurs: see Section 3.3.8, "Fixed-Point Arithmetic Instructions" on page 59.

† The ***stwcx.*** and ***stdcx.*** instructions (see Book II,
† *PowerPC AS Virtual Environment Architecture*) also set CR Field 0.

For all floating-point instructions in which Rc=1, CR Field 1 (bits 4:7 of the Condition Register) is set to the Floating-Point exception status, copied from bits 0:3 of the Floating-Point Status and Control Register. These bits are interpreted as follows.

Bit	Description
4	***Floating-Point Exception Summary*** (FX) This is a copy of the final state of $FPSCR_{FX}$ at the completion of the instruction.
5	***Floating-Point Enabled Exception Summary*** (FEX) This is a copy of the final state of $FPSCR_{FEX}$ at the completion of the instruction.
6	***Floating-Point Invalid Operation Exception Summary*** (VX) This is a copy of the final state of $FPSCR_{VX}$ at the completion of the instruction.
7	***Floating-Point Overflow Exception*** (OX) This is a copy of the final state of $FPSCR_{OX}$ at the completion of the instruction.

For *Compare* instructions, a specified CR field is set to reflect the result of the comparison. The bits of the specified CR field are interpreted as follows. A complete description of how the bits are set is given in the instruction descriptions in Section 3.3.9, "Fixed-Point Compare Instructions" on page 68 and Section 4.6.7, "Floating-Point Compare Instructions" on page 133.

Bit	Description
0	***Less Than, Floating-Point Less Than*** (LT, FL) For fixed-point *Compare* instructions, (RA) < SI or (RB) (signed comparison) or (RA) $\overset{u}{<}$ UI or (RB) (unsigned comparison). For floating-point *Compare* instructions, (FRA) < (FRB).

1 ***Greater Than, Floating-Point Greater Than*** (GT, FG)
For fixed-point *Compare* instructions, (RA) $>$ SI or (RB) (signed comparison) or (RA) $\overset{u}{>}$ UI or (RB) (unsigned comparison). For floating-point *Compare* instructions, (FRA) $>$ (FRB).

2 ***Equal, Floating-Point Equal*** (EQ, FE)
For fixed-point *Compare* instructions, (RA) = SI, UI, or (RB). For floating-point *Compare* instructions, (FRA) = (FRB).

3 ***Summary Overflow, Incomparable, Floating-Point Unordered (SO, IC, FU)***
For fixed-point *Compare* instructions except ***cmpla***, this is a copy of the final state of XER_{SO} at the completion of the instruction. For the ***cmpla*** instruction, the operands are not comparable due to quantities with bits 0:39 unequal. For floating-point *Compare* instructions, one or both of (FRA) and (FRB) is a NaN.

2.3.2 Link Register

The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the *Branch Conditional to Link Register* instruction, and it
† holds the return address after *Branch* instructions for
† which LK=1 and after *System Call Vectored*
instructions.

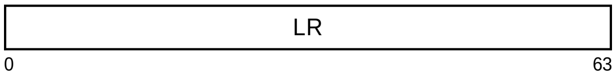

Figure 21. Link Register

2.3.3 Count Register

The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of *Branch* instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the *Branch Conditional to Count Register* instruction. The Count Register is modified by the *System Call Vectored* instruction.

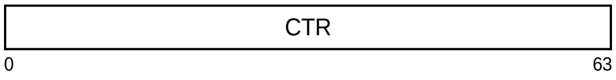

Figure 22. Count Register

2.4 Branch Processor Instructions

2.4.1 Branch Instructions

The sequence of instruction execution can be changed by the *Branch* instructions. Because all instructions are on word boundaries, bits 62 and 63 of the generated branch target address are ignored by the processor in performing the branch.

The *Branch* instructions compute the effective address (EA) of the target in one of the following four ways, as described in Section 1.12.3, “Effective Address Calculation” on page 17.

1. Adding a displacement to the address of the *Branch* instruction (*Branch* or *Branch Conditional* with AA=0).
2. Specifying an absolute address (*Branch* or *Branch Conditional* with AA=1).
3. Using the address contained in the Link Register (*Branch Conditional to Link Register*).
4. Using the address contained in the Count Register (*Branch Conditional to Count Register*).

In all four cases, in 32-bit mode the final step in the address computation is setting the high-order 32 bits of the target address to 0.

For the first two methods, the target addresses can be computed sufficiently ahead of the *Branch* instruction that instructions can be prefetched along the target path. For the third and fourth methods, prefetching instructions along the target path is also possible provided the Link Register or the Count Register is loaded sufficiently ahead of the *Branch* instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided (LK=1), the effective address of the instruction following the *Branch* instruction is placed into the Link Register after the branch target address has been computed; this is done regardless of whether the branch is taken.

For *Branch Conditional* instructions, the BO field specifies the conditions under which the branch is † taken, as shown in Figure 23. In the figure, M=0 in 64-bit mode and M=32 in 32-bit mode. If the BO field specifies that the CTR is to be decremented, the entire 64-bit CTR is decremented regardless of the mode.

BO	Description
0000z	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and $CR_{BI} = 0$
0001z	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$ and $CR_{BI} = 0$
001at	Branch if $CR_{BI} = 0$
0100z	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$ and $CR_{BI} = 1$
0101z	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$ and $CR_{BI} = 1$
011at	Branch if $CR_{BI} = 1$
1a00t	Decrement the CTR, then branch if the decremented $CTR_{M:63} \neq 0$
1a01t	Decrement the CTR, then branch if the decremented $CTR_{M:63} = 0$
1z1zz	Branch always

Notes:
1. “z” denotes a bit that is ignored.
2. The “a” and “t” bits are used as described below.

Figure 23. BO field encodings

The “a” and “t” bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken, as shown in Figure 24.

at	Hint
00	No hint is given
01	Reserved
10	The branch is very likely not to be taken
11	The branch is very likely to be taken

Figure 24. "at" bit encodings

Programming Note

Many implementations have dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate, and is likely to be overridden by any hint provided by the “at” bits, the “at” bits should be set to 0b00 unless the static prediction implied by at=0b10 or at=0b11 is highly likely to be correct.

For *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions, the BH field provides a hint about the use of the instruction, as shown in Figure 25.

BH	Hint
00	***bclr***[***l***]: The instruction is a subroutine return ***bcctr***[***l***]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken
01	***bclr***[***l***]: The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken ***bcctr***[***l***]: Reserved
10	Reserved
11	***bclr***[***l***] and ***bcctr***[***l***]: The target address is not predictable

Figure 25. BH field encodings

Programming Note

The hint provided by the BH field is independent of the hint provided by the "at" bits (e.g., the BH field provides no indication of whether the branch is likely to be taken).

Extended mnemonics for branches

Many extended mnemonics are provided so that *Branch Conditional* instructions can be coded with portions of the BO and BI fields as part of the mnemonic rather than as part of a numeric operand. Some of these are shown as examples with the *Branch* instructions. See Appendix B, "Assembler Extended Mnemonics" on page 161 for additional extended mnemonics.

Programming Note

The hints provided by the "at" bits and by the BH field do not affect the results of executing the instruction.

The "z" bits should be set to 0, as they may be assigned a meaning in some future version of the architecture.

Programming Note

Many implementations have dynamic mechanisms for predicting the target addresses of ***bclr***[***l***] and ***bcctr***[***l***] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by *Branch* instructions for which LK=1 and for which the branch was taken) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

- Use *Branch* instructions for which LK=1 only as subroutine calls (including function calls, etc.).
- Pair each subroutine call (i.e., each *Branch* instruction for which LK=1 and the branch is taken) with a ***bclr*** instruction that returns from the subroutine and has BH=0b00.
- Do not use ***bclrl*** as a subroutine call. (Some implementations access the return address cache at most once per instruction; such implementations are likely to treat ***bclrl*** as a subroutine return, and not as a subroutine call.)
- For ***bclr***[***l***] and ***bcctr***[***l***], use the appropriate value in the BH field.

The following are examples of programming conventions that obey these rules. In the examples, BH is assumed to contain 0b00 unless otherwise stated. In addition, the "at" bits are assumed to be coded appropriately.

† Let A, B, and Glue be specific programs.

- Loop counts:
  Keep them in the Count Register, and use a ***bc*** instruction (LK=0) to decrement the count and to branch back to the beginning of the loop if the decremented count is nonzero.
- Computed goto's, case statements, etc.:
  Use the Count Register to hold the address to branch to, and use a ***bcctr*** instruction (LK=0, and BH=0b11 if appropriate) to branch to the selected address.
- Direct subroutine linkage:
  Here A calls B and B returns to A. The two branches should be as follows.
  - A calls B: use a ***bl*** or ***bcl*** instruction (LK=1).
  - B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).
- Indirect subroutine linkage:
  Here A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller; the Binder inserts "glue" code to mediate the branch.) The three branches should be as follows.
  - A calls Glue: use a ***bl*** or ***bcl*** instruction (LK=1).

(Programming Note continues in next column....)

Programming Note (continued)

- Glue calls B: place the address of B into the Count Register, and use a ***bcctr*** instruction (LK=0).
- B returns to A: use a ***bclr*** instruction (LK=0) (the return address is in, or can be restored to, the Link Register).

■ Function call:

Here A calls a function, the identity of which may vary from one instance of the call to another, instead of calling a specific program B. This case should be handled using the conventions of the preceding two bullets, depending on whether the call is direct or indirect, with the following differences.

- If the call is direct, place the address of the function into the Count Register, and use a ***bcctrl*** instruction (LK=1) instead of a ***bl*** or ***bcl*** instruction.
- For the ***bcctr***[***l***] instruction that branches to the function, use BH=0b11 if appropriate.

Compatibility Note

The bits corresponding to the current "a" and "t" bits, and to the current "z" bits except in the "branch always" BO encoding, had different meanings in versions of the architecture that precede Version 2.00.

■ The bit corresponding to the "t" bit was called the "y" bit. The "y" bit indicated whether to use the architected default prediction (y=0) or to use the complement of the default prediction (y=1). The default prediction was defined as follows.

- If the instruction is ***bc***[***l***][***a***] with a negative value in the displacement field, the branch is taken. (This is the only case in which the prediction corresponding to the "y" bit differs from the prediction corresponding to the "t" bit.)
- In all other cases (***bc***[***l***][***a***] with a nonnegative value in the displacement field, ***bclr***[***l***], or ***bcctr***[***l***]), the branch is not taken.

■ The BO encodings that test both the Count Register and the Condition Register had a "y" bit in place of the current "z" bit. The meaning of the "y" bit was as described in the preceding item.

■ The "a" bit was a "z" bit.

Because these bits have always been defined either to be ignored or to be treated as hints, a given program will produce the same result on any implementation regardless of the values of the bits. Also, because even the "y" bit is ignored, in practice, by most processors that implement versions of the architecture that precede Version 2.00, the performance of a given program on those processors will not be affected by the values of the bits.

Architecture Note

In some future version of the architecture, the value at=0b01 may be used to indicate that the branch path (taken or not taken) is unpredictable (i.e., that neither static nor dynamic prediction is likely to predict the path accurately). It is expected that any new meaning will be such that future *Branch Conditional* instructions that use at=0b01 would use at=0b00 in the current architecture.

Decisions regarding assignment of a meaning for at=0b01 must include consideration of the extent to which software still uses the earlier meaning (see the preceding Compatibility Note), and of the effect that the new meaning would have on the performance of such software.

Decisions regarding assignment of a meaning for bit 16 of ***bclr***[***l***] and ***bcctr***[***l***] instructions in some future version of the architecture (e.g., to extend the BH field) must include consideration of the fact that processors that implement versions of the architecture that precede Version 2.00 may use the bit in computing the prediction associated with the "y" bit. Specifically, for all three *Branch Conditional* instructions, such processors may predict that the branch will be taken if the value of the following expression is 1, and will not be taken if the value is 0. "s" represents bit 16 of the instruction.

$$(BO_0 \ \& \ BO_2) \ | \ (s \oplus BO_4)$$

The expression assumes that instruction bit 16, which is the sign bit of the displacement field for ***bc***[***l***][***a***], contains 0 for ***bclr***[***l***] and ***bcctr***[***l***].

Branch I-form

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)

18	LI	AA	LK
0	6	30	31

```
if AA then NIA $\leftarrow_{iea}$ EXTS(LI || 0b00)
else       NIA $\leftarrow_{iea}$ CIA $+_{tia}$ EXTS(LI || 0b00)
if LK then LR $\leftarrow_{iea}$ CIA $+_{tia}$ 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

LR	(if LK=1)

Branch Conditional B-form

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)

16	BO	BI	BD	AA	LK
0	6	11	16	30	31

```
if (64-bit mode)
  then M $\leftarrow$ 0
  else M $\leftarrow$ 32
if $\neg BO_2$ then CTR $\leftarrow$ CTR $-$ 1
ctr_ok $\leftarrow$ $BO_2$ | (($CTR_{M:63} \neq 0$) $\oplus$ $BO_3$)
cond_ok $\leftarrow$ $BO_0$ | ($CR_{BI} \equiv BO_1$)
if ctr_ok & cond_ok then
  if AA then NIA $\leftarrow_{iea}$ EXTS(BD || 0b00)
  else       NIA $\leftarrow_{iea}$ CIA $+_{tia}$ EXTS(BD || 0b00)
if LK then LR $\leftarrow_{iea}$ CIA $+_{tia}$ 4
```

† The BI field specifies the Condition Register bit to be
† tested. The BO field is used to resolve the branch as
† described in Figure 23. *target_addr* specifies the
branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional*:

Extended:		*Equivalent to:*	
blt	target	bc	12,0,target
bne	cr2,target	bc	4,10,target
bdnz	target	bc	16,0,target

Branch Conditional to Link Register XL-form

bclr	BO,BI,BH	(LK=0)
bclrl	BO,BI,BH	(LK=1)

[POWER mnemonics: bcr, bcrl]

19	BO	BI	///	BH	16	LK
0	6	11	16	19	21	31

```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬$BO_2$ then CTR ← CTR − 1
ctr_ok ← $BO_2$ | (($CTR_{M:63}$ ≠ 0) ⊕ $BO_3$)
cond_ok ← $BO_0$ | ($CR_{BI}$ ≡ $BO_1$)
if ctr_ok & cond_ok then NIA $\leftarrow_{iea}$ $LR_{0:61}$ || 0b00
if LK then LR $\leftarrow_{iea}$ CIA $+_{tia}$ 4
```

The BI field specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 23. The BH field is used as described in Figure 25. The branch target address is $LR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Special Registers Altered:

CTR	(if BO_2=0)
LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Link Register*:

Extended:		*Equivalent to:*	
bclr	4,6	bclr	4,6,0
bltlr		bclr	12,0,0
bnelr	cr2	bclr	4,10,0
bdnzlr		bclr	16,0,0

Programming Note

bclr, bclrl, bcctr, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr, bclrl, bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr, bclrl, bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00.

Branch Conditional to Count Register XL-form

bcctr	BO,BI,BH	(LK=0)
bcctrl	BO,BI,BH	(LK=1)

[POWER mnemonics: bcc, bccl]

19	BO	BI	///	BH	528	LK
0	6	11	16	19	21	31

```
cond_ok ← $BO_0$ | ($CR_{BI}$ ≡ $BO_1$)
if cond_ok then NIA $\leftarrow_{iea}$ $CTR_{0:61}$ || 0b00
if LK then LR $\leftarrow_{iea}$ CIA $+_{tia}$ 4
```

The BI field specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 23. The BH field is used as described in Figure 25. The branch target address is $CTR_{0:61}$ || 0b00, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

If the “decrement and test CTR” option is specified (BO_2=0), the instruction form is invalid.

Special Registers Altered:

LR	(if LK=1)

Extended Mnemonics:

Examples of extended mnemonics for *Branch Conditional to Count Register*:

Extended:		*Equivalent to:*	
bcctr	4,6	bcctr	4,6,0
bltctr		bcctr	12,0,0
bnectr	cr2	bcctr	4,10,0

2.4.2 System Call Instructions

These instructions provide the means by which a program can call upon the system to perform a service.

System Call SC-form

sc LEV

[POWER mnemonic: svca]

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

System Call Vectored SC-form

scv LEV

[POWER mnemonic: svcl]

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

These instructions call the system to perform a service. A complete description of these instructions can be found in Book III, *PowerPC AS Operating Environment Architecture*.

The first form of the instruction (***sc***) provides a single system call. The second form of the instruction (***scv***) provides the capability for 128 unique system calls.

The use of the LEV field is described in Book III. In the first form of the instruction the contents of the LEV field must be 0 or 1; otherwise the results are boundedly undefined.

When control is returned to the program that executed the *System Call* or *System Call Vectored* instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

These instructions are context synchronizing (see Book III, *PowerPC AS Operating Environment Architecture*).

In *tags inactive* mode, ***scv*** is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
Dependent on the system service

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.

In application programs the value of the LEV operand for ***sc*** should be 0.

Compatibility Note

For a discussion of POWER compatibility with respect to instruction bits 16:19 and 27:29, see Appendix E, "Incompatibilities with the POWER Architecture" on page 185. For compatibility with future versions of the PowerPC AS Architecture, these bits should be coded as zeros.

2.4.3 Condition Register Logical Instructions

The *Condition Register Logical* instructions have preferred forms: see Section 1.9.1, “Preferred Instruction Forms” on page 14. In the preferred forms, the BT and BB fields satisfy the following rule.

- The bit specified by BT is in the same Condition Register field as the bit specified by BB.

Extended mnemonics for Condition Register logical operations

A set of extended mnemonics is provided that allow additional Condition Register logical operations, beyond those provided by the basic *Condition Register Logical* instructions, to be coded easily. Some of these are shown as examples with the *Condition Register Logical* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Condition Register AND XL-form

crand BT,BA,BB

19	BT	BA	BB	257	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \,\&\, CR_{BB}$

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB, and the result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Condition Register OR XL-form

cror BT,BA,BB

19	BT	BA	BB	449	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \mid CR_{BB}$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB, and the result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register OR*:

Extended:	*Equivalent to:*
crmove Bx,By	cror Bx,By,By

Condition Register XOR XL-form

crxor BT,BA,BB

19	BT	BA	BB	193	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \oplus CR_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB, and the result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register XOR*:

Extended:	*Equivalent to:*
crclr Bx	crxor Bx,Bx,Bx

Condition Register NAND XL-form

crnand BT,BA,BB

19	BT	BA	BB	225	/
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA} \,\&\, CR_{BB})$

The bit in the Condition Register specified by BA is ANDed with the bit in the Condition Register specified by BB, and the complemented result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Condition Register NOR XL-form

crnor BT,BA,BB

19	BT	BA	BB	33	/
0	6	11	16	21	31

$CR_{BT} \leftarrow \neg(CR_{BA} \mid CR_{BB})$

The bit in the Condition Register specified by BA is ORed with the bit in the Condition Register specified by BB, and the complemented result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register NOR*:

Extended:	*Equivalent to:*
crnot Bx,By	crnor Bx,By,By

Condition Register Equivalent XL-form

creqv BT,BA,BB

19	BT	BA	BB	289	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \equiv CR_{BB}$

The bit in the Condition Register specified by BA is XORed with the bit in the Condition Register specified by BB, and the complemented result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Extended Mnemonics:

Example of extended mnemonics for *Condition Register Equivalent*:

Extended:	*Equivalent to:*
crset Bx	creqv Bx,Bx,Bx

Condition Register AND with Complement XL-form

crandc BT,BA,BB

19	BT	BA	BB	129	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \ \& \ \neg CR_{BB}$

The bit in the Condition Register specified by BA is ANDed with the complement of the bit in the Condition Register specified by BB, and the result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

Condition Register OR with Complement XL-form

crorc BT,BA,BB

19	BT	BA	BB	417	/
0	6	11	16	21	31

$CR_{BT} \leftarrow CR_{BA} \mid \neg CR_{BB}$

The bit in the Condition Register specified by BA is ORed with the complement of the bit in the Condition Register specified by BB, and the result is placed into the bit in the Condition Register specified by BT.

† **Special Registers Altered:**
† CR_{BT}

2.4.4 Condition Register Field Instruction

Move Condition Register Field XL-form

mcrf BF,BFA

19	BF	//	BFA	//	///	0	/
0	6	9	11	14	16	21	31

$CR_{4\times BF:4\times BF+3} \leftarrow CR_{4\times BFA:4\times BFA+3}$

The contents of Condition Register field BFA are copied to Condition Register field BF.

† **Special Registers Altered:**
† CR field BF

Chapter 3. Fixed-Point Processor

3.1 Fixed-Point Processor Overview

This chapter describes the registers and instructions that make up the Fixed-Point Processor facility. Section 3.2, “Fixed-Point Processor Registers” describes the registers associated with the Fixed-Point Processor. Section 3.3, “Fixed-Point Processor Instructions” on page 36 describes the instructions associated with the Fixed-Point Processor.

3.2 Fixed-Point Processor Registers

3.2.1 General Purpose Registers

All manipulation of information is done in registers internal to the Fixed-Point Processor. The principal storage internal to the Fixed-Point Processor is a set of 32 General Purpose Registers (GPRs). See Figure 26.

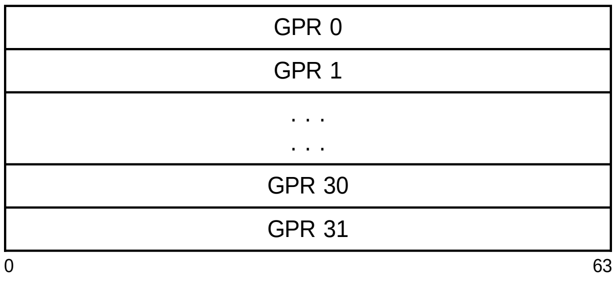

Figure 26. General Purpose Registers

Each GPR is a 64-bit register.

3.2.2 Fixed-Point Exception Register

The Fixed-Point Exception Register (XER) is a 64-bit register.

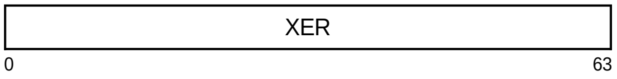

Figure 27. Fixed-Point Exception Register

The bit definitions for the Fixed-Point Exception Register are shown below. Here $M=0$ in 64-bit mode and $M=32$ in 32-bit mode.

The bits are set based on the operation of an instruction considered as a whole, not on intermediate results (e.g., the *Subtract From Carrying* instruction, the result of which is specified as the sum of three values, sets bits in the Fixed-Point Exception Register based on the entire operation, not on an intermediate sum).

Bit(s) Description

0:15 In *tags active* mode (see Book III, *PowerPC AS Operating Environment Architecture*), these bits are reserved. In *tags inactive* mode, a ***mfspr*** of the XER returns zeros for these bit positions.

16:31 ***Decimal Carries*** (DC)
In *tags active* mode, bit *n* of this field is set equal to the carry out of decimal digit position *n* (bit position $4 \times n$) when an *Add Carrying* or *Subtract From Carrying* instruction is executed. The name “Decimal Carries” conveys the intended use; however, these carries are binary carries from binary bit positions. In *tags inactive* mode, a ***mfspr*** of the XER returns zeros for this field.

32 ***Summary Overflow*** (SO)
The Summary Overflow bit is set to 1 whenever an instruction (except ***mtspr***) sets the Overflow bit. Once set, the SO bit remains set until it is cleared by an ***mtspr*** instruction (specifying the XER) or an ***mcrxr*** instruction. It is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow. Executing an ***mtspr*** instruction to the XER, supplying the values 0 for SO and 1 for OV, causes SO to be set to 0 and OV to be set to 1.

33 ***Overflow*** (OV)
The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction. XO-form *Add, Subtract From*, and *Negate* instructions having OE=1 set it to 1 if the carry out of bit *M* is not equal to the carry out of bit $M+1$, and set it to 0 otherwise. XO-form *Multiply Low* and *Divide* instructions having OE=1 set it to 1 if the result cannot be represented in 64 bits (***mulld, divd, divdu***) or in 32 bits (***mullw, divw, divwu***), and set it to 0 otherwise. The OV bit is not altered by *Compare* instructions, nor by other instructions (except ***mtspr*** to the XER, and ***mcrxr***) that cannot overflow.

34 ***Carry*** (CA)
The Carry bit is set as follows, during execution of certain instructions. *Add Carrying, Subtract From Carrying, Add Extended,* and *Subtract From Extended* instructions set it to 1 if there is a carry out of bit *M*, and set it to 0 otherwise. *Shift Right Algebraic* instructions set it to 1 if any 1-bits have been shifted out of a negative operand, and set it to 0 otherwise. The CA bit is not altered by *Compare* instructions, nor by other instructions (except *Shift Right Algebraic*, ***dtcs***, ***mtspr*** to the XER, and ***mcrxr***) that cannot carry.

35 ***Offset Carry*** (OC)
In *tags active* mode, during execution of *Add Carrying* type of instructions (***addic***[**.**], ***addic***[***o***][**.**]), the Offset Carry bit is set to one if one of the following conditions is met; otherwise, it is set to zero.

- $(RA)_{0:15}=0$ and the carry out of bit 16 is one.
- $(RA)_{0:15}\neq 0$ and the carry out of bit 40 is one.

In *tags active* mode, *Subtract From Carrying* type of instructions (***subfc***[***o***][**.**] and ***subfic***) set the Offset Carry bit to an undefined value.

36:39 ***FXCC***
In *tags active* mode, this field is set whenever CR Field 0 is set by an instruction that records, and whenever any CR field is set by the fixed-point *Compare* instructions. In *tags active* mode, the first three bits (LT, GT, EQ) are set in the same way as the corresponding bits of the affected CR field (see Section 2.3.1, “Condition Register” on page 22): specifically, they are set by algebraic or logical comparison of the two operands (*Compare* instructions), or by algebraic comparison of the result with zero (other instructions). The fourth bit (IC) is set as described below.

36 ***Negative, Less Than*** (LT)

37 ***Positive, Greater Than*** (GT)

38 ***Zero, Equal*** (EQ)

39 ***Incomparable*** (IC)
In *tags active* mode, the ***cmpla*** instruction sets the Incomparable bit to one if the two operands do not have the same value in bits 0:39, and sets it to zero otherwise. In *tags active* mode, all other instructions that set the FXCC set the Incomparable bit to zero.

40 ***Decimal Summary*** (DS)
In *tags active* mode, the Decimal Summary bit is set to the logical OR of all the DC bits when an *Add Carrying* or *Subtract From Carrying* instruction is executed.

41:43 ***Tag Condition Code*** (TGCC)

41 ***T02***
When ***lq*** is executed, this bit is set based on a decode of bits 0:2 of the data and tag bit fetched from storage and an immediate field in the instruction.

42 ***T07***
When ***lq*** is executed, this bit is set based on a decode of bits 0:7 of the data and tag bit fetched from storage and an immediate field in the instruction.

43 ***TAG***
When ***lq*** is executed, this bit is set to the value of the quadword tag bit fetched from storage. This bit is set to 1 by ***settag***.

44:56 Reserved

57:63 This field specifies the number of bytes to be transferred by a *Load String Indexed* or *Store String Indexed* instruction.

Compatibility Note

For a discussion of POWER compatibility with | respect to XER bits 48:55, see Appendix E, “Incompatibilities with the POWER Architecture” on page 185. For compatibility with future ver- † sions of the PowerPC AS Architecture, these bits should be set to zero.

3.3 Fixed-Point Processor Instructions

3.3.1 Fixed-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.12.3, “Effective Address Calculation” on page 17.

†

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at that address. Unlike a *Load* or *Store* instruction, ***la*** cannot cause an Effective Address Overflow exception. This extended mnemonic is described in Section B.11, “Miscellaneous Mnemonics” on page 174.

Programming Note

The DS field in DS-form *Storage Access* instructions is a word offset, not a byte offset like the D field in D-form *Storage Access* instructions. However, for programming convenience, Assemblers should support the specification of byte offsets for both forms of instruction.

Programming Note

See the Programming Note on page 6 regarding base register usage for X-form *Load* and *Store* instructions in *tags active* mode.

3.3.1.1 Tagged Values

Certain fixed-point *Load* instructions copy tag bits from storage to special purpose registers. Only one type of *Store* instruction, ***stq***, can cause a storage tag bit to be set to 1 while all others cause tag bits to be set to 0.

Additional details on tag preservation are given in the individual instruction descriptions.

†

3.3.1.2 Storage Access Exceptions

Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

If the storage location specified by a *Load Quadword* or *Store Quadword* instruction is in storage that is Write Through Required or Caching Inhibited (see Book II, *PowerPC AS Virtual Environment Architecture*), the system data storage error handler or the system alignment error handler may be invoked.

3.3.2 Fixed-Point Load Instructions

The byte, halfword, word, or doubleword in storage addressed by EA is loaded into register RT. If the instruction is ***lq***, the quadword in storage addressed by EA is loaded into registers RT and RT+1, in increasing order of storage address and register number.

†

Many of the *Load* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA≠0 and RA≠RT, the effective address is placed into register RA and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into RT.

Programming Note

In some implementations, the *Load Algebraic* and *Load with Update* instructions may have greater latency than other types of *Load* instructions. Moreover, *Load with Update* instructions may take longer to execute in some implementations than the corresponding pair of a non-update *Load* instruction and an *Add* instruction.

Load Byte and Zero D-form

lbz RT,D(RA)

34	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(D)
RT ← 560 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:
None

Load Byte and Zero Indexed X-form

lbzx RT,RA,RB

31	RT	RA	RB	87	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RT ← 560 || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

Special Registers Altered:
None

Load Byte and Zero with Update D-form

lbzu RT,D(RA)

35	RT	RA	D
0	6	11	16 31

```
EA ← (RA) +tea EXTS(D)
RT ← 560 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Byte and Zero with Update Indexed X-form

lbzux RT,RA,RB

31	RT	RA	RB	119	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
RT ← 560 || MEM(EA, 1)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). The byte in storage addressed by EA is loaded into $RT_{56:63}$. $RT_{0:55}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword and Zero D-form

lhz RT,D(RA)

40	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
RT ← $^{48}0$ || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Halfword and Zero Indexed X-form

lhzx RT,RA,RB

31	RT	RA	RB	279	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
RT ← $^{48}0$ || MEM(EA, 2)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Halfword and Zero with Update D-form

lhzu RT,D(RA)

41	RT	RA	D
0	6	11	16 31

```
EA ← (RA) $+_{tea}$ EXTS(D)
RT ← $^{48}0$ || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword and Zero with Update Indexed X-form

lhzux RT,RA,RB

31	RT	RA	RB	311	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
RT ← $^{48}0$ || MEM(EA, 2)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword Algebraic D-form

lha RT,D(RA)

42	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
RT ← EXTS(MEM(EA, 2))
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic Indexed X-form

lhax RT,RA,RB

31	RT	RA	RB	343	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
RT ← EXTS(MEM(EA, 2))
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

Special Registers Altered:
None

Load Halfword Algebraic with Update D-form

lhau RT,D(RA)

43	RT	RA	D
0	6	11	16 31

```
EA ← (RA) $+_{tea}$ EXTS(D)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Halfword Algebraic with Update Indexed X-form

lhaux RT,RA,RB

31	RT	RA	RB	375	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
RT ← EXTS(MEM(EA, 2))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). The halfword in storage addressed by EA is loaded into $RT_{48:63}$. $RT_{0:47}$ are filled with a copy of bit 0 of the loaded halfword.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Word and Zero D-form

lwz RT,D(RA)

[POWER mnemonic: l]

32	RT	RA	D
0	6	11	16 … 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
RT ← $^{32}0$ || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update D-form

lwzu RT,D(RA)

[POWER mnemonic: lu]

33	RT	RA	D
0	6	11	16 … 31

```
EA ← (RA) $+_{tea}$ EXTS(D)
RT ← $^{32}0$ || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word and Zero Indexed X-form

lwzx RT,RA,RB

[POWER mnemonic: lx]

31	RT	RA	RB	23	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
RT ← $^{32}0$ || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

Load Word and Zero with Update Indexed X-form

lwzux RT,RA,RB

[POWER mnemonic: lux]

31	RT	RA	RB	55	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
RT ← $^{32}0$ || MEM(EA, 4)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:

None

Load Word Algebraic DS-form

lwa RT,DS(RA)

58	RT	RA	DS	2
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(DS || 0b00)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(DS||0b00). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:
None

Load Word Algebraic Indexed X-form

lwax RT,RA,RB

31	RT	RA	RB	341	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RT ← EXTS(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

Special Registers Altered:
None

Load Word Algebraic with Update Indexed X-form

lwaux RT,RA,RB

31	RT	RA	RB	373	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
RT ← EXTS(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are filled with a copy of bit 0 of the loaded word.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Doubleword DS-form

ld RT,DS(RA)

58	RT	RA	DS	0
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(DS || 0b00)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ *tea*(DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:
None

Load Doubleword with Update DS-form

ldu RT,DS(RA)

58	RT	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) $+_{tea}$ EXTS(DS || 0b00)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ *tea*(DS||0b00). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Doubleword Indexed X-form

ldx RT,RA,RB

31	RT	RA	RB	21	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ *tea*(RB). The doubleword in storage addressed by EA is loaded into RT.

Special Registers Altered:
None

Load Doubleword with Update Indexed X-form

ldux RT,RA,RB

31	RT	RA	RB	53	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
RT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ *tea*(RB). The doubleword in storage addressed by EA is loaded into RT.

EA is placed into register RA.

If RA=0 or RA=RT, the instruction form is invalid.

Special Registers Altered:
None

Load Quadword DQ-form

lq RT,DQ(RA),PT

56	RT	RA	DQ	PT
0	6	11	16	28 31

```
This instruction uses a DECODE function
   y = DECODE(x) defined by:
           x   y
          00  1000
          01  0100
          10  0010
          11  0001

if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(DQ || 0b0000)
if EA_60:63 = 0b0000 then
  RT ←  MEM(EA, 8)
  GPR(RT+1) ← MEM(EA+8, 8)
  if ((DECODE(MEM_0:1(EA, 1)) & PT) ≠ 0b0000) &
     (MEM_2(EA, 1) = 0) & (MEM_tag(EA) = 1) &
     (MEM_tag(EA+8) = 1) then XER_41 ← 0b1
  else XER_41 ← 0b0
  if (MEM(EA, 1) = (0b1010 || PT)) & (MEM_tag(EA) = 1)
     & (MEM_tag(EA+8) = 1) then XER_42 ← 0b1
  else XER_42 ← 0b0
  XER_43 ← MEM_tag(EA) & MEM_tag(EA+8)
else
  RT ← undefined
  GPR(RT+1) ← undefined
  u ← undefined 1-bit value
  if u then
    XER_41:43 ← 0b000
  else
    XER_41:43 ← undefined
    system alignment error handler
```

If $EA_{60:63}$=0b0000, the quadword in storage addressed by EA is loaded into registers RT and RT+1, in increasing order of storage address and register number; otherwise, the contents of registers RT and RT+1 are undefined.

XER_{41} is set to 1 if four conditions are met:

- all tags within the quadword are 1,
- bit 2 of the data loaded into RT is 0, and
- a DECODE of the two high-order bits loaded into RT ANDed with the PT field is not equal to 0b0000
- $EA_{60:63}$=0b0000

otherwise, it is set to 0. The DECODE function y = DECODE(x) is defined by:

x	y
00	1000
01	0100
10	0010
11	0001

If $EA_{60:63} \neq$ 0b0000, it is implementation-dependent whether the system alignment error handler is invoked. If the system alignment error handler is invoked, XER is undefined.

If $EA_{60:63}$=0b0000 and if all tags within the quadword are 1 and the high-order byte loaded into RT is equal to the byte formed by concatenating 0b1010 with the PT field then XER_{42} is set to 1; otherwise it is set to 0.

XER_{43} is set to 1 if $EA_{60:63}$=0b0000 and all tags within the quadword are 1; otherwise it is set to 0.

If RT is odd, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
TGCC

3.3.3 Fixed-Point Store Instructions

The contents of register RS are stored into the byte, halfword, word, or doubleword in storage addressed by EA. For ***stq***, the contents of registers RT and RT+1 are stored into the quadword in storage addressed by EA, in increasing order of storage address and register number. If an aligned quadword is stored, the tag(s) of the quadword in storage is set to the value of the XER TAG bit. For all other fixed-point *Store* instructions, the tag of every tag block affected is set to zero.

†

Many of the *Store* instructions have an "update" form, in which register RA is updated with the effective address. For these forms, the following rules apply.

- If RA≠0, the effective address is placed into register RA.
- If RS=RA, the contents of register RS are copied to the target storage element and then EA is placed into RA (RS).

Store Byte D-form

stb RS,D(RA)

38	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(D)
MEM(EA, 1) ← (RS)56:63
MEMtag(EA) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:
None

Store Byte Indexed X-form

stbx RS,RA,RB

31	RS	RA	RB	215	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 1) ← (RS)56:63
MEMtag(EA) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

Special Registers Altered:
None

Store Byte with Update D-form

stbu RS,D(RA)

39	RS	RA	D
0	6	11	16 31

```
EA ← (RA) +tea EXTS(D)
MEM(EA, 1) ← (RS)56:63
MEMtag(EA) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Byte with Update Indexed X-form

stbux RS,RA,RB

31	RS	RA	RB	247	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
MEM(EA, 1) ← (RS)56:63
MEMtag(EA) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). $(RS)_{56:63}$ are stored into the byte in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Halfword D-form

sth RS,D(RA)

44	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(D)
MEM(EA, 2) ← (RS)48:63
MEMtag(EA, 2) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword Indexed X-form

sthx RS,RA,RB

31	RS	RA	RB	407	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 2) ← (RS)48:63
MEMtag(EA, 2) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Halfword with Update D-form

sthu RS,D(RA)

45	RS	RA	D
0	6	11	16 31

```
EA ← (RA) +tea EXTS(D)
MEM(EA, 2) ← (RS)48:63
MEMtag(EA, 2) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Halfword with Update Indexed X-form

sthux RS,RA,RB

31	RS	RA	RB	439	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
MEM(EA, 2) ← (RS)48:63
MEMtag(EA, 2) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). $(RS)_{48:63}$ are stored into the halfword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Word D-form

stw RS,D(RA)

[POWER mnemonic: st]

36	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +_tea EXTS(D)
MEM(EA, 4) ← (RS)_32:63
MEM_tag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D. (RS)$_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Word Indexed X-form

stwx RS,RA,RB

[POWER mnemonic: stx]

31	RS	RA	RB	151	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +_tea (RB)
MEM(EA, 4) ← (RS)_32:63
MEM_tag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). (RS)$_{32:63}$ are stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Word with Update D-form

stwu RS,D(RA)

[POWER mnemonic: stu]

37	RS	RA	D
0	6	11	16 31

```
EA ← (RA) +_tea EXTS(D)
MEM(EA, 4) ← (RS)_32:63
MEM_tag(EA, 4) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D. (RS)$_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Word with Update Indexed X-form

stwux RS,RA,RB

[POWER mnemonic: stux]

31	RS	RA	RB	183	/
0	6	11	16	21	31

```
EA ← (RA) +_tea (RB)
MEM(EA, 4) ← (RS)_32:63
MEM_tag(EA, 4) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). (RS)$_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Doubleword DS-form

std RS,DS(RA)

62	RS	RA	DS	0
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
$MEM_{tag}$(EA, 8) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Doubleword Indexed X-form

stdx RS,RA,RB

31	RS	RA	RB	149	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
MEM(EA, 8) ← (RS)
$MEM_{tag}$(EA, 8) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). (RS) is stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Doubleword with Update DS-form

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) $+_{tea}$ EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
$MEM_{tag}$(EA, 8) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Doubleword with Update Indexed X-form

stdux RS,RA,RB

31	RS	RA	RB	181	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
MEM(EA, 8) ← (RS)
$MEM_{tag}$(EA, 8) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Quadword DS-form

stq RS,DS(RA)

62	RS	RA	DS	2
0	6	11	16	30 31

```
If RA = 0 then b ← 0
else           b ← (RA)
EA ← b +$_{tea}$ EXTS(DS || 0b00)
If EA$_{60:63}$ = 0b0000 then
  MEM(EA, 8) ← RS
  MEM(EA+8, 8) ← GPR(RS+1)
  MEM$_{tag}$(EA, 16) ← XER$_{43}$
else
  invoke system alignment error handler
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(DS||0b00). (RS) and (RS+1) are stored into the quadword in storage addressed by EA, in increasing order of storage address and register number.

If RS is odd, the instruction form is invalid.

All tags within the quadword in storage are set to the value of XER$_{43}$.

If the effective address is not quadword aligned, the system alignment error handler is invoked, and the store is not performed.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

†

3.3.4 Fixed-Point Load and Store with Byte Reversal Instructions

†

Programming Note

† These instructions have the effect of loading and
† storing data in Little-Endian byte order.

In some implementations, the *Load Byte-Reverse* instructions may have greater latency than other *Load* instructions.

Load Halfword Byte-Reverse Indexed X-form

lhbrx RT,RA,RB

31	RT	RA	RB	790	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RT ← 480 || MEM(EA+tea1, 1) || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ *tea*(RB). Bits 0:7 of the halfword in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the halfword in storage addressed by EA are loaded into $RT_{48:55}$. $RT_{0:47}$ are set to 0.

Special Registers Altered:
None

Load Word Byte-Reverse Indexed X-form

lwbrx RT,RA,RB

[POWER mnemonic: lbrx]

31	RT	RA	RB	534	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RT ← 320 || MEM(EA+tea3, 1) || MEM(EA+tea2, 1)
         || MEM(EA+tea1, 1) || MEM(EA, 1)
```

Let the effective address (EA) be the sum (RA|0)+ *tea*(RB). Bits 0:7 of the word in storage addressed by EA are loaded into $RT_{56:63}$. Bits 8:15 of the word in storage addressed by EA are loaded into $RT_{48:55}$. Bits 16:23 of the word in storage addressed by EA are loaded into $RT_{40:47}$. Bits 24:31 of the word in storage addressed by EA are loaded into $RT_{32:39}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:
None

Store Halfword Byte-Reverse Indexed X-form

sthbrx RS,RA,RB

31	RS	RA	RB	918	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 2) ← (RS)56:63 || (RS)48:55
MEMtag(EA, 2) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). $(RS)_{56:63}$ are stored into bits 0:7 of the halfword in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the halfword in storage addressed by EA.

Special Registers Altered:
None

Store Word Byte-Reverse Indexed X-form

stwbrx RS,RA,RB

[POWER mnemonic: stbrx]

31	RS	RA	RB	662	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 4) ← (RS)56:63 || (RS)48:55 || (RS)40:47 || (RS)32:39
MEMtag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). $(RS)_{56:63}$ are stored into bits 0:7 of the word in storage addressed by EA. $(RS)_{48:55}$ are stored into bits 8:15 of the word in storage addressed by EA. $(RS)_{40:47}$ are stored into bits 16:23 of the word in storage addressed by EA. $(RS)_{32:39}$ are stored into bits 24:31 of the word in storage addressed by EA.

Special Registers Altered:
None

3.3.5 Fixed-Point Load and Store Multiple Instructions

The *Load/Store Multiple* instructions have preferred forms: see Section 1.9.1, “Preferred Instruction Forms” on page 14. In the preferred forms, storage alignment satisfies the following rule.

- The combination of the EA and RT (RS) is such that the low-order byte of GPR 31 is loaded (stored) from (into) the last byte of an aligned octword in storage.

Compatibility Note

For a discussion of POWER compatibility with respect to the alignment of the EA for the *Load Multiple Word* and *Store Multiple Word* instructions, see Appendix E, “Incompatibilities with the POWER Architecture” on page 185. For compatibility with future versions of the PowerPC AS Architecture, these EAs should be word-aligned.

Engineering Note

Causing the system alignment error handler to be invoked if attempt is made to execute a *Load Multiple* or *Store Multiple* instruction having an incorrectly aligned effective address facilitates the debugging of software.

Load Multiple Word D-form

lmw RT,D(RA)

[POWER mnemonic: lm]

46	RT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← $b +_{tea}$ EXTS(D)
r ← RT
do while r ≤ 31
  GPR(r) ← $^{32}0$ || MEM(EA, 4)
  r ← r + 1
  EA ← $EA +_{tea} 4$
```

Let n = (32− RT). Let the effective address (EA) be the sum $(RA|0) +_{tea} D$.

n consecutive words starting at EA are loaded into the low-order 32 bits of GPRs RT through 31. The high-order 32 bits of these GPRs are set to zero.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Multiple Doubleword DS-form

lmd RT,DS(RA)

58	RT	RA	DS	3
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← $b +_{tea}$ EXTS(DS || 0b00)
r ← RT
do while r ≤ 31
  GPR(r) ← MEM(EA, 8)
  r ← r + 1
  EA ← $EA +_{tea} 8$
```

Let n = (32− RT). Let the effective address (EA) be the sum $(RA|0) +_{tea} (DS||0b00)$.

n consecutive doublewords starting at EA are loaded into GPRs from RT through 31.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Store Multiple Word D-form

stmw RS,D(RA)

[POWER mnemonic: stm]

47	RS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
r ← RS
do while r ≤ 31
  MEM(EA, 4) ← $GPR(r)_{32:63}$
  $MEM_{tag}$(EA, 4) ← 0
  r ← r + 1
  EA ← EA $+_{tea}$ 4
```

Let n = (32– RS). Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D.

n consecutive words starting at EA are stored from the low-order 32 bits of GPRs RS through 31.

EA must be a multiple of 4. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

Special Registers Altered:
None

Store Multiple Doubleword DS-form

stmd RS,DS(RA)

62	RS	RA	DS	3
0	6	11	16	30 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(DS || 0b00)
r ← RS
do while r ≤ 31
  MEM(EA, 8) ← GPR(r)
  $MEM_{tag}$(EA, 8) ← 0
  r ← r + 1
  EA ← EA $+_{tea}$ 8
```

Let n = (32– RS). Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(DS||0b00).

n consecutive doublewords starting at EA are stored from GPRs RS through 31.

EA must be a multiple of 8. If it is not, either the system alignment error handler is invoked or the results are boundedly undefined.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

3.3.6 Fixed-Point Move Assist Instructions

The *Move Assist* instructions allow movement of data from storage to registers or from registers to storage without concern for alignment. These instructions can be used for a short move between arbitrary storage locations or to initiate a long move between unaligned storage fields.

The *Load/Store String* instructions have preferred forms: see Section 1.9.1, “Preferred Instruction Forms” on page 14. In the preferred forms, register usage satisfies the following rules.

- RS = 4 or 5
- RT = 4 or 5
- last register loaded/stored ≤ 12

†

For some implementations, using GPR 4 for RS and RT may result in slightly faster execution than using GPR 5; see Book IV, *PowerPC AS Implementation Features*.

Architecture Note

The preferred register for RS and RT in PowerPC is GPR 5.

Load String Word Immediate X-form

lswi RT,RA,NB

[POWER mnemonic: lsi]

31	RT	RA	NB	597	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RT − 1
i ← 32
do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)_{i:i+7} ← MEM(EA, 1)
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA +_{tea} 1
  n ← n − 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to load. Let nr = CEIL(n÷4); nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr−1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load String Doubleword Immediate X-form

lsdi RT,RA,NB

31	RT	RA	NB	629	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RT − 1
i ← 0
do while n > 0
  if i = 0 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)_{i:i+7} ← MEM(EA, 1)
  i ← i + 8 (mod 64)
  EA ← EA +_{tea} 1
  n ← n − 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0: n is the number of bytes to load. Let nr = CEIL(n÷8): nr is the number of registers to receive data.

n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data are loaded into all eight bytes of each GPR.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If register RT+nr−1 is only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If RA is in the range of registers to be loaded, including the case in which RA=0, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Load String Word Indexed X-form

lswx RT,RA,RB

[POWER mnemonic: lsx]

31	RT	RA	RB	533	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
n ← XER57:63
r ← RT − 1
i ← 32
RT ← undefined
do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)i:i+7 ← MEM(EA, 1)
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA +tea 1
  n ← n − 1
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). Let n = XER$_{57:63}$; n is the number of bytes to load. Let nr = CEIL(n÷4); nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data are loaded into the low-order four bytes of each GPR; the high-order four bytes are set to 0.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If the low-order four bytes of register RT+nr−1 are only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, either the system illegal instruction error handler is invoked or the results are boundedly undefined. If RT=RA or RT=RB, the instruction form is invalid.

Special Registers Altered:
None

Load String Doubleword Indexed X-form

lsdx RT,RA,RB

31	RT	RA	RB	565	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
n ← XER57:63
r ← RT − 1
i ← 0
RT ← undefined
do while n > 0
  if i = 0 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)i:i+7 ← MEM(EA, 1)
  i ← i + 8 (mod 64)
  EA ← EA +tea 1
  n ← n − 1
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). Let n = XER$_{57:63}$: n is the number of bytes to load. Let nr = CEIL(n÷8): nr is the number of registers to receive data.

If n>0, n consecutive bytes starting at EA are loaded into GPRs RT through RT+nr−1. Data are loaded into all eight bytes of each GPR.

Bytes are loaded left to right in each register. The sequence of registers wraps around to GPR 0 if required. If register RT+nr−1 is only partially filled, the unfilled low-order byte(s) of that register are set to 0.

If n=0, the contents of register RT are undefined.

If RA or RB is in the range of registers to be loaded, including the case in which RA=0, either the system illegal instruction error handler is invoked or the results are boundedly undefined. If RT=RA or RT=RB, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Store String Word Immediate X-form

stswi RS,RA,NB

[POWER mnemonic: stsi]

31	RS	RA	NB	725	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RS − 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)$_{i:i+7}$
  MEM$_{tag}$(EA) ← 0
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA +$_{tea}$ 1
  n ← n − 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0; n is the number of bytes to store. Let nr = CEIL(n÷4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr−1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

Special Registers Altered:
None

Store String Doubleword Immediate X-form

stsdi RS,RA,NB

31	RS	RA	NB	757	/
0	6	11	16	21	31

```
if RA = 0 then EA ← 0
else           EA ← (RA)
if NB = 0 then n ← 32
else           n ← NB
r ← RS − 1
i ← 0
do while n > 0
  if i = 0 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)$_{i:i+7}$
  MEM$_{tag}$(EA) ← 0
  i ← i + 8 (mod 64)
  EA ← EA +$_{tea}$ 1
  n ← n − 1
```

Let the effective address (EA) be (RA|0). Let n = NB if NB≠0, n = 32 if NB=0: n is the number of bytes to store. Let nr = CEIL(n÷8): nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr−1. Data are stored from all eight bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Store String Word Indexed X-form

stswx RS,RA,RB

[POWER mnemonic: stsx]

31	RS	RA	RB	661	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +$_{tea}$ (RB)
n ← XER$_{57:63}$
r ← RS – 1
i ← 32
do while n > 0
  if i = 32 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)$_{i:i+7}$
  MEM$_{tag}$(EA) ← 0
  i ← i + 8
  if i = 64 then i ← 32
  EA ← EA +$_{tea}$ 1
  n ← n – 1
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). Let n = XER$_{57:63}$; n is the number of bytes to store. Let nr = CEIL(n÷ 4); nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr–1. Data are stored from the low-order four bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

If n=0, no bytes are stored.

Special Registers Altered:
None

Store String Doubleword Indexed X-form

stsdx RS,RA,RB

31	RS	RA	RB	693	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +$_{tea}$ (RB)
n ← XER$_{57:63}$
r ← RS – 1
i ← 0
do while n > 0
  if i = 0 then r ← r + 1 (mod 32)
  MEM(EA, 1) ← GPR(r)$_{i:i+7}$
  MEM$_{tag}$(EA) ← 0
  i ← i + 8 (mod 64)
  EA ← EA +$_{tea}$ 1
  n ← n – 1
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB). Let n = XER$_{57:63}$: n is the number of bytes to store. Let nr = CEIL(n÷ 8): nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs RS through RS+nr–1. Data are stored from all eight bytes of each GPR.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR 0 if required.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

If n=0, no bytes are stored.

Special Registers Altered:
None

†

3.3.7 Other Fixed-Point Instructions

The remainder of the fixed-point instructions use the contents of the General Purpose Registers (GPRs) as source operands, and place results into GPRs, into the Fixed-Point Exception Register (XER), and into Condition Register fields. In addition, the *Trap* instructions compare the contents of one GPR with a second GPR or immediate data and, if the specified conditions are met, invoke the system trap handler.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation.

The X-form and XO-form instructions with Rc=1, and the D-form instructions ***addic., andi.,*** and ***andis.,*** set the first three bits of CR Field 0 to characterize the result placed into the target register. In 64-bit mode, these bits are set by signed comparison of the result to zero. In 32-bit mode, these bits are set by signed comparison of the low-order 32 bits of the result to zero.

Unless otherwise noted and when appropriate, when CR Field 0 and the XER are set they reflect the value placed into the target register.

Programming Note

Instructions with the OE bit set or that set CA may execute slowly or may prevent the execution of subsequent instructions until the instruction has completed.

3.3.8 Fixed-Point Arithmetic Instructions

The XO-form *Arithmetic* instructions with Rc=1, and the D-form *Arithmetic* instruction ***addic.***, set the first three bits of CR Field 0 as described in Section 3.3.7, “Other Fixed-Point Instructions” on page 58.

addic, addic., subfic, addc, subfc, adde, subfe, addme, subfme, addze, and ***subfze*** always set CA, to reflect the carry out of bit 0 in 64-bit mode and out of bit 32 in 32-bit mode. The XO-form *Arithmetic* instructions set SO and OV when OE=1 to reflect overflow of the result. Except for the *Multiply Low* and *Divide* instructions, the setting of these bits is mode-dependent, and reflects overflow of the 64-bit result in 64-bit mode and overflow of the low-order 32-bit result in 32-bit mode. For XO-form *Multiply Low* and *Divide* instructions, the setting of these bits is mode-independent, and reflects overflow of the 64-bit result for ***mulld, divd***, and ***divdu***, and overflow of the low-order 32-bit result for ***mullw, divw***, and ***divwu***.

Programming Note

Notice that CR Field 0 may not reflect the “true” (infinitely precise) result if overflow occurs.

Extended mnemonics for addition and subtraction

Several extended mnemonics are provided that use the *Add Immediate* and *Add Immediate Shifted* instructions to load an immediate value or an address into a target register. Some of these are shown as examples with the two instructions.

The PowerPC AS Architecture supplies *Subtract From* instructions, which subtract the second operand from the third. A set of extended mnemonics is provided that use the more “normal” order, in which the third operand is subtracted from the second, with the third operand being either an immediate field or a register. Some of these are shown as examples with the appropriate *Add* and *Subtract From* instructions.

See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Add Immediate D-form

addi RT,RA,SI

[POWER mnemonic: cal]

14	RT	RA	SI
0	6	11	16 31

```
if RA = 0 then RT ← EXTS(SI)
else           RT ← (RA) + EXTS(SI)
```

The sum (RA|0) + SI is placed into register RT.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate*:

Extended:		*Equivalent to:*	
li	Rx,value	addi	Rx,0,value
la	Rx,disp(Ry)	addi	Rx,Ry,disp
subi	Rx,Ry,value	addi	Rx,Ry,– value

Programming Note

addi, addis, add, and ***subf*** are the preferred instructions for addition and subtraction, because they set few status bits.

Notice that ***addi*** and ***addis*** use the value 0, not the contents of GPR 0, if RA=0.

Add Immediate Shifted D-form

addis RT,RA,SI

[POWER mnemonic: cau]

15	RT	RA	SI
0	6	11	16 31

```
if RA = 0 then RT ← EXTS(SI || $^{16}0$)
else           RT ← (RA) + EXTS(SI || $^{16}0$)
```

The sum (RA|0) + (SI || 0x0000) is placed into register RT.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Add Immediate Shifted*:

Extended:		*Equivalent to:*	
lis	Rx,value	addis	Rx,0,value
subis	Rx,Ry,value	addis	Rx,Ry,– value

Add XO-form

add	RT,RA,RB	(OE=0 Rc=0)
add.	RT,RA,RB	(OE=0 Rc=1)
addo	RT,RA,RB	(OE=1 Rc=0)
addo.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: cax, cax., caxo, caxo.]

31	RT	RA	RB	OE	266	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + (RB)
```

The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Add Immediate Carrying D-form

addic RT,RA,SI

[POWER mnemonic: ai]

12	RT	RA	SI
0	6	11	16 31

```
RT ← (RA) + EXTS(SI)
```

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CA OC

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying*:

Extended:	*Equivalent to:*
subic Rx,Ry,value	addic Rx,Ry,– value

Subtract From XO-form

subf	RT,RA,RB	(OE=0 Rc=0)
subf.	RT,RA,RB	(OE=0 Rc=1)
subfo	RT,RA,RB	(OE=1 Rc=0)
subfo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	40	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + (RB) + 1
```

The sum ¬(RA) + (RB) + 1 is placed into register RT.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From*:

Extended:	*Equivalent to:*
sub Rx,Ry,Rz	subf Rx,Rz,Ry

Add Immediate Carrying and Record D-form

addic. RT,RA,SI

[POWER mnemonic: ai.]

13	RT	RA	SI
0	6	11	16 31

```
RT ← (RA) + EXTS(SI)
```

The sum (RA) + SI is placed into register RT.

Special Registers Altered:

CR0 FXCC CA OC

Extended Mnemonics:

Example of extended mnemonics for *Add Immediate Carrying and Record*:

Extended:	*Equivalent to:*
subic. Rx,Ry,value	addic. Rx,Ry,– value

Subtract From Immediate Carrying D-form

subfic RT,RA,SI

[POWER mnemonic: sfi]

8	RT	RA	SI
0	6	11	16 31

RT ← ¬(RA) + EXTS(SI) + 1

The sum ¬(RA) + SI + 1 is placed into register RT.

| **Special Registers Altered:**
| CA
| OC (undefined)

Add Carrying XO-form

addc	RT,RA,RB	(OE=0 Rc=0)
addc.	RT,RA,RB	(OE=0 Rc=1)
addco	RT,RA,RB	(OE=1 Rc=0)
addco.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: a, a., ao, ao.]

31	RT	RA	RB	OE	10	Rc
0	6	11	16	21	22	31

RT ← (RA) + (RB)

The sum (RA) + (RB) is placed into register RT.

Special Registers Altered:

DC CA OC DS	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

addc and ***subfc*** are the only instructions that set Decimal Carries. In some implementations they may be slower than similar instructions that do not set Decimal Carries.

Subtract From Carrying XO-form

subfc	RT,RA,RB	(OE=0 Rc=0)
subfc.	RT,RA,RB	(OE=0 Rc=1)
subfco	RT,RA,RB	(OE=1 Rc=0)
subfco.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: sf, sf., sfo, sfo.]

31	RT	RA	RB	OE	8	Rc
0	6	11	16	21	22	31

RT ← ¬(RA) + (RB) + 1

The sum ¬(RA) + (RB) + 1 is placed into register RT.

| **Special Registers Altered:**

| DC CA DS	
| OC (undefined)	
| CR0 FXCC	(if Rc=1)
| SO OV	(if OE=1)

Extended Mnemonics:

Example of extended mnemonics for *Subtract From Carrying*:

Extended:	*Equivalent to:*
subc Rx,Ry,Rz	subfc Rx,Rz,Ry

Add Extended XO-form

adde	RT,RA,RB	(OE=0 Rc=0)
adde.	RT,RA,RB	(OE=0 Rc=1)
addeo	RT,RA,RB	(OE=1 Rc=0)
addeo.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: ae, ae., aeo, aeo.]

31	RT	RA	RB	OE	138	Rc
0	6	11	16	21	22	31

```
$RT \leftarrow (RA) + (RB) + CA$
```

The sum (RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Add to Minus One Extended XO-form

addme	RT,RA	(OE=0 Rc=0)
addme.	RT,RA	(OE=0 Rc=1)
addmeo	RT,RA	(OE=1 Rc=0)
addmeo.	RT,RA	(OE=1 Rc=1)

[POWER mnemonics: ame, ame., ameo, ameo.]

31	RT	RA	///	OE	234	Rc
0	6	11	16	21	22	31

```
$RT \leftarrow (RA) + CA - 1$
```

The sum (RA) + CA + ${}^{64}1$ is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Subtract From Extended XO-form

subfe	RT,RA,RB	(OE=0 Rc=0)
subfe.	RT,RA,RB	(OE=0 Rc=1)
subfeo	RT,RA,RB	(OE=1 Rc=0)
subfeo.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: sfe, sfe., sfeo, sfeo.]

31	RT	RA	RB	OE	136	Rc
0	6	11	16	21	22	31

```
$RT \leftarrow \neg(RA) + (RB) + CA$
```

The sum ¬(RA) + (RB) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Subtract From Minus One Extended XO-form

subfme	RT,RA	(OE=0 Rc=0)
subfme.	RT,RA	(OE=0 Rc=1)
subfmeo	RT,RA	(OE=1 Rc=0)
subfmeo.	RT,RA	(OE=1 Rc=1)

[POWER mnemonics: sfme, sfme., sfmeo, sfmeo.]

31	RT	RA	///	OE	232	Rc
0	6	11	16	21	22	31

```
$RT \leftarrow \neg(RA) + CA - 1$
```

The sum ¬(RA) + CA + ${}^{64}1$ is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Add to Zero Extended XO-form

addze	RT,RA	(OE=0 Rc=0)
addze.	RT,RA	(OE=0 Rc=1)
addzeo	RT,RA	(OE=1 Rc=0)
addzeo.	RT,RA	(OE=1 Rc=1)

[POWER mnemonics: aze, aze., azeo, azeo.]

31	RT	RA	///	OE	202	Rc
0	6	11	16	21	22	31

```
RT ← (RA) + CA
```

The sum (RA) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Subtract From Zero Extended XO-form

subfze	RT,RA	(OE=0 Rc=0)
subfze.	RT,RA	(OE=0 Rc=1)
subfzeo	RT,RA	(OE=1 Rc=0)
subfzeo.	RT,RA	(OE=1 Rc=1)

[POWER mnemonics: sfze, sfze., sfzeo, sfzeo.]

31	RT	RA	///	OE	200	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + CA
```

The sum ¬(RA) + CA is placed into register RT.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The setting of CA by the *Add* and *Subtract From* instructions, including the Extended versions thereof, is mode-dependent. If a sequence of these instructions is used to perform extended-precision addition or subtraction, the same mode should be used throughout the sequence.

Negate XO-form

neg	RT,RA	(OE=0 Rc=0)
neg.	RT,RA	(OE=0 Rc=1)
nego	RT,RA	(OE=1 Rc=0)
nego.	RT,RA	(OE=1 Rc=1)

31	RT	RA	///	OE	104	Rc
0	6	11	16	21	22	31

```
RT ← ¬(RA) + 1
```

The sum ¬(RA) + 1 is placed into register RT.

If executing in 64-bit mode and register RA contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number and, if OE=1, OV is set to 1. Similarly, if executing in 32-bit mode and $(RA)_{32:63}$ contain the most negative 32-bit number (0x8000_0000), the low-order 32 bits of the result contain the most negative 32-bit number and, if OE=1, OV is set to 1.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Multiply Low Immediate D-form

mulli RT,RA,SI

[POWER mnemonic: muli]

7	RT	RA	SI
0	6	11	16 31

```
prod$_{0:127}$ ← (RA) × EXTS(SI)
RT ← prod$_{64:127}$
```

The 64-bit first operand is (RA). The 64-bit second operand is the sign-extended value of the SI field. The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:
None

Multiply Low Doubleword XO-form

mulld	RT,RA,RB	(OE=0 Rc=0)
mulld.	RT,RA,RB	(OE=0 Rc=1)
mulldo	RT,RA,RB	(OE=1 Rc=0)
mulldo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	233	Rc
0	6	11	16	21	22	31

```
prod$_{0:127}$ ← (RA) × (RB)
RT ← prod$_{64:127}$
```

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The XO-form *Multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply Low Word XO-form

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: muls, muls., mulso, mulso.]

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

```
RT ← (RA)$_{32:63}$ × (RB)$_{32:63}$
```

The 32-bit operands are the low-order 32 bits of RA and of RB. The 64-bit product of the operands is placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

For ***mulli*** and ***mulld***, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers. For ***mulli*** and ***mullw***, the low-order 32 bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

Multiply High Doubleword XO-form

mulhd	RT,RA,RB	(Rc=0)
mulhd.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	73	Rc
0	6	11	16	21	22	31

```
$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$
```

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Multiply High Doubleword Unsigned XO-form

mulhdu	RT,RA,RB	(Rc=0)
mulhdu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	9	Rc
0	6	11	16	21	22	31

```
$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{0:63}$
```

The 64-bit operands are (RA) and (RB). The high-order 64 bits of the 128-bit product of the operands are placed into register RT.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Multiply High Word XO-form

mulhw	RT,RA,RB	(Rc=0)
mulhw.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	75	Rc
0	6	11	16	21	22	31

```
$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined
```

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit
† product of the operands are placed into $RT_{32:63}$. The
† contents of $RT_{0:31}$ are undefined.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0 FXCC (CR0(0:2) and FXCC(36:38) undefined in 64-bit mode)	(if Rc=1)

Multiply High Word Unsigned XO-form

mulhwu	RT,RA,RB	(Rc=0)
mulhwu.	RT,RA,RB	(Rc=1)

31	RT	RA	RB	/	11	Rc
0	6	11	16	21	22	31

```
$prod_{0:63} \leftarrow (RA)_{32:63} \times (RB)_{32:63}$
$RT_{32:63} \leftarrow prod_{0:31}$
$RT_{0:31} \leftarrow$ undefined
```

The 32-bit operands are the low-order 32 bits of RA and of RB. The high-order 32 bits of the 64-bit
† product of the operands are placed into $RT_{32:63}$. The
† contents of $RT_{0:31}$ are undefined.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero.

Special Registers Altered:

CR0 FXCC (CR0(0:2) and FXCC(36:38) undefined in 64-bit mode)	(if Rc=1)

Divide Doubleword XO-form

divd	RT,RA,RB	(OE=0 Rc=0)
divd.	RT,RA,RB	(OE=0 Rc=1)
divdo	RT,RA,RB	(OE=1 Rc=0)
divdo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	489	Rc
0	6	11	16	21	22	31

```
dividend_{0:63} ← (RA)
divisor_{0:63} ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000_0000_0000 ÷ –1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0 and of the FXCC. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 64-bit signed remainder of dividing (RA) by (RB) can be computed as follows, except in the case that (RA) = -2^{63} and (RB) = -1.

```
divd   RT,RA,RB    # RT = quotient
mulld  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Word XO-form

divw	RT,RA,RB	(OE=0 Rc=0)
divw.	RT,RA,RB	(OE=0 Rc=1)
divwo	RT,RA,RB	(OE=1 Rc=0)
divwo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	491	Rc
0	6	11	16	21	22	31

```
dividend_{0:63} ← EXTS((RA)_{32:63})
divisor_{0:63} ← EXTS((RB)_{32:63})
RT_{32:63} ← dividend ÷ divisor
RT_{0:31} ← undefined
```

The 64-bit dividend is the sign-extended value of $(RA)_{32:63}$. The 64-bit divisor is the sign-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into † $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \leq r < |divisor|$ if the dividend is nonnegative, and $-|divisor| < r \leq 0$ if the dividend is negative.

If an attempt is made to perform any of the divisions

```
0x8000_0000 ÷ –1
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0 and of the FXCC. In these cases, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 FXCC (CR0(0:2) and FXCC(36:38) undefined in 64-bit mode)	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 32-bit signed remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows, except in the case that $(RA)_{32:63} = -2^{31}$ and $(RB)_{32:63} = -1$.

```
divw   RT,RA,RB    # RT = quotient
mullw  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Doubleword Unsigned XO-form

divdu	RT,RA,RB	(OE=0 Rc=0)
divdu.	RT,RA,RB	(OE=0 Rc=1)
divduo	RT,RA,RB	(OE=1 Rc=0)
divduo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	457	Rc
0	6	11	16	21	22	31

```
dividend_{0:63} ← (RA)
divisor_{0:63} ← (RB)
RT ← dividend ÷ divisor
```

The 64-bit dividend is (RA). The 64-bit divisor is (RB). The 64-bit quotient of the dividend and divisor is placed into register RT. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0 and of the FXCC. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 FXCC	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 64-bit unsigned remainder of dividing (RA) by (RB) can be computed as follows.

```
divdu  RT,RA,RB    # RT = quotient
mulld  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

Divide Word Unsigned XO-form

divwu	RT,RA,RB	(OE=0 Rc=0)
divwu.	RT,RA,RB	(OE=0 Rc=1)
divwuo	RT,RA,RB	(OE=1 Rc=0)
divwuo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	459	Rc
0	6	11	16	21	22	31

```
dividend_{0:63} ← ^{32}0 || (RA)_{32:63}
divisor_{0:63} ← ^{32}0 || (RB)_{32:63}
RT_{32:63} ← dividend ÷ divisor
RT_{0:31} ← undefined
```

The 64-bit dividend is the zero-extended value of $(RA)_{32:63}$. The 64-bit divisor is the zero-extended value of $(RB)_{32:63}$. The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into † $RT_{32:63}$. The contents of $RT_{0:31}$ are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR Field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies

$$dividend = (quotient \times divisor) + r$$

where $0 \le r < divisor$.

If an attempt is made to perform the division

```
<anything> ÷ 0
```

then the contents of register RT are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR Field 0 and of the FXCC. In this case, if OE=1 then OV is set to 1.

Special Registers Altered:

CR0 FXCC (CR0(0:2) and FXCC(36:38) undefined in 64-bit mode)	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The 32-bit unsigned remainder of dividing $(RA)_{32:63}$ by $(RB)_{32:63}$ can be computed as follows.

```
divwu  RT,RA,RB    # RT = quotient
mullw  RT,RT,RB    # RT = quotient*divisor
subf   RT,RT,RA    # RT = remainder
```

3.3.9 Fixed-Point Compare Instructions

The fixed-point *Compare* instructions compare the contents of register RA with (1) the sign-extended value of the SI field, (2) the zero-extended value of the UI field, or (3) the contents of register RB. The comparison is signed for ***cmpi*** and ***cmp***, and unsigned for ***cmpli*** and ***cmpl***.

The L field controls whether the operands are treated as 64-bit or 32-bit quantities, as follows:

L	Operand length
0	32-bit operands
1	64-bit operands

When the operands are treated as 32-bit signed quantities, bit 32 of the register (RA or RB) is the sign bit.

The ***cmpi, cmp, cmpli***, and ***cmpl*** instructions set one bit in the leftmost three bits of the designated CR field to 1, and the other two to 0. XER$_{SO}$ is copied to bit 3 of the designated CR field. The ***cmpla*** instruction sets one bit in the designated CR field to 1, and the other three to 0.

The CR field is set as follows.

Bit	Name	Description
0	LT	(RA) < SI or (RB) (signed comparison) (RA) $\overset{u}{<}$ UI or (RB) (unsigned comparison)
1	GT	(RA) > SI or (RB) (signed comparison) (RA) $\overset{u}{>}$ UI or (RB) (unsigned comparison)
2	EQ	(RA) = SI, UI, or (RB)
3	SO,IC	Summary Overflow from the XER, or Incomparable (***cmpla*** only)

The *Compare* instructions also set XER$_{FXCC}$, as follows.

Bit	Name	Description
0	LT	(RA) < SI, UI, or (RB)
1	GT	(RA) > SI, UI, or (RB)
2	EQ	(RA) = SI, UI, or (RB)
3	IC	Incomparable (used by ***cmpla***; set to 0 by other compares)

Extended mnemonics for compares

A set of extended mnemonics is provided so that compares can be coded with the operand length as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Compare* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Compare Immediate D-form

cmpi BF,L,RA,SI

11	BF	/	L	RA	SI
0	6	9	10	11	16 31

```
if L = 0 then a ← EXTS((RA)$_{32:63}$)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                      c ← 0b001
CR$_{4×BF:4×BF+3}$ ← c || XER$_{SO}$
FXCC ← c || 0b0
```

The contents of register RA ((RA)$_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF and into the FXCC.

Special Registers Altered:
CR field BF, FXCC

Extended Mnemonics:

Examples of extended mnemonics for *Compare Immediate*:

Extended:		*Equivalent to:*	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare X-form

cmp BF,L,RA,RB

31	BF	/	L	RA	RB	0	/
0	6	9	10	11	16	21	31

```
if L = 0 then a ← EXTS((RA)$_{32:63}$)
              b ← EXTS((RB)$_{32:63}$)
         else a ← (RA)
              b ← (RB)
if      a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR$_{4×BF:4×BF+3}$ ← c || XER$_{SO}$
FXCC ← c || 0b0
```

The contents of register RA ((RA)$_{32:63}$ if L=0) are compared with the contents of register RB ((RB)$_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field BF and into the FXCC.

Special Registers Altered:
CR field BF, FXCC

Extended Mnemonics:

Examples of extended mnemonics for *Compare*:

Extended:		*Equivalent to:*	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

Compare Logical Immediate D-form

cmpli BF,L,RA,UI

10	BF	/	L	RA	UI	
0	6	9	10	11	16	31

```
if L = 0 then a ← $^{32}0 \,||\, (RA)_{32:63}$
         else a ← (RA)
if      a $\overset{u}{<}$ ($^{48}0$ || UI) then c ← 0b100
else if a $\overset{u}{>}$ ($^{48}0$ || UI) then c ← 0b010
else                         c ← 0b001
$CR_{4\times BF:4\times BF+3}$ ← c || $XER_{SO}$
FXCC ← c || 0b0
```

The contents of register RA ($(RA)_{32:63}$ zero-extended to 64 bits if L=0) are compared with $^{48}0$ || UI, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF and into the FXCC.

Special Registers Altered:
CR field BF, FXCC

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical Immediate*:

Extended:		*Equivalent to:*	
cmpldi	Rx,value	cmpli	0,1,Rx,value
cmplwi	cr3,Rx,value	cmpli	3,0,Rx,value

Compare Logical X-form

cmpl BF,L,RA,RB

31	BF	/	L	RA	RB	32	/
0	6	9	10	11	16	21	31

```
if L = 0 then a ← $^{32}0 \,||\, (RA)_{32:63}$
              b ← $^{32}0 \,||\, (RB)_{32:63}$
         else a ← (RA)
              b ← (RB)
if      a $\overset{u}{<}$ b then c ← 0b100
else if a $\overset{u}{>}$ b then c ← 0b010
else                c ← 0b001
$CR_{4\times BF:4\times BF+3}$ ← c || $XER_{SO}$
FXCC ← c || 0b0
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as unsigned integers. The result of the comparison is placed into CR field BF and into the FXCC.

Special Registers Altered:
CR field BF, FXCC

Extended Mnemonics:

Examples of extended mnemonics for *Compare Logical*:

Extended:		*Equivalent to:*	
cmpld	Rx,Ry	cmpl	0,1,Rx,Ry
cmplw	cr3,Rx,Ry	cmpl	3,0,Rx,Ry

Compare Logical Addresses X-form

cmpla BF,RA,RB

31	BF	/	1	RA	RB	64	/
0	6	9	10	11	16	21	31

```
if $(RA)_{0:15}$ = $(RB)_{0:15}$ &
   ( $(RA)_{0:15}$ = 0 |
     $(RA)_{0:39}$ = $(RB)_{0:39}$ ) then
  if      (RA) $<^u$ (RB) then c ← 0b1000
  else if (RA) $>^u$ (RB) then c ← 0b0100
  else                     c ← 0b0010
else                       c ← 0b0001
$CR_{4\times BF:4\times BF+3}$ ← c
FXCC ← c
```

If the high order 16 bits of the contents of register RA are zero and both operands have the same value in bit positions 0 to 15, or if the high order 16 bits of the contents of register RA are not all zero and both operands have the same value in bit positions 0 to 39, then the contents of register RA is compared with the contents of register RB, treating the operands as unsigned integers. The result of the comparison is placed into CR field BF and into the FXCC. The low-order bit of the FXCC is set to zero.

If the operands differ in the high-order 16 bits, or if the high order 16 bits of the contents of register RA are not all zero and the operands differ in the high-order 40 bits, the high-order three bits of CR field BF and of the FXCC are set to zero, and the low-order bit of CR field BF and of the FXCC is set to one. In this case the operands are said to be “incomparable”.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
CR field BF, FXCC

3.3.10 Fixed-Point Trap Instructions

The *Trap* instructions are provided to test for a specified set of conditions. If any of the conditions tested by a *Trap* instruction are met, the system trap handler is invoked. If none of the tested conditions are met, instruction execution continues normally.

Except for the ***txer*** instruction, the contents of register RA are compared with either the sign-extended value of the SI field or the contents of register RB, depending on the *Trap* instruction. For ***tdi*** and ***td***, the entire contents of RA (and RB) participate in the comparison; for ***twi*** and ***tw***, only the contents of the low-order 32 bits of RA (and RB) participate in the comparison.

In *tags inactive* mode (see Book III, *PowerPC AS Operating Environment Architecture*) or with TO equal to any value other than 0b11100 or 0b11110, this comparison results in five conditions which are ANDed with TO. If the result is not 0 the system trap handler is invoked. These conditions are as follows.

TO Bit	ANDed with Condition
0	Less Than, using signed comparison
1	Greater Than, using signed comparison
2	Equal
3	Less Than, using unsigned comparison
4	Greater Than, using unsigned comparison

For ***tdi*** and ***td*** in *tags active* mode and TO = 0b11100, this comparison results in two conditions that cause the system trap handler to be invoked. The system trap handler is invoked if any of the following conditions are true:

- The high-order 16 bits are not equal
- The high-order 16 bits of one operand are not all zero and either the high-order 40 bits are not equal or the first operand is less than the second, using signed comparison

For ***tdi*** and ***td*** in *tags active* mode and TO = 0b11110, this comparison results in three conditions that cause the system trap handler to be invoked. The system trap handler is invoked if any of the following conditions are true:

- Less Than, using signed comparison
- Logically Less Than
- The high-order 40 bits are not equal

twi and ***tw*** in *tags active* mode with TO = 0b11100 or TO = 0b11110 are invalid forms.

For the ***txer*** instruction, the contents of a specified XER bit are compared with the TO field. If the two are equal the system trap handler is invoked.

Extended mnemonics for traps

A set of extended mnemonics is provided so that traps can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Trap* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Trap Doubleword Immediate D-form

tdi TO,RA,SI

2	TO	RA	SI
0	6	11	16 31

```
a ← (RA)
b ← EXTS(SI)
if ( $a_{0:15} \ne b_{0:15}$ |
     ( $a_{0:15} \ne 0$ &
     ( ( $a_{16:39} \ne b_{16:39}$) | (a < b) ) ) ) &
    TO = 0b11100 & (tags active) then TRAP
if ($a_{0:39} \ne b_{0:39}$) &
(TO = 0b11110) & (tags active) then TRAP
if (a < EXTS(SI)) & $TO_0$ &
   ( TO ≠ 0b11100  | tags inactive) then TRAP
if (a > EXTS(SI)) & $TO_1$ &
   ((TO ≠ 0b11100  &
     TO ≠ 0b11110) | tags inactive) then TRAP
if (a = EXTS(SI)) & $TO_2$ &
   ((TO ≠ 0b11100  &
     TO ≠ 0b11110) | tags inactive) then TRAP
if (a $<^u$ EXTS(SI)) & $TO_3$ then TRAP
if (a $>^u$ EXTS(SI)) & $TO_4$ then TRAP
```

The contents of register RA are compared with the sign-extended value of the SI field. In *tags inactive* mode or if TO is equal to any value other than 0b11100 or 0b11110, if any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

In *tags active* mode with TO = 0b11100 the system trap handler is invoked if any of the following conditions are met.

- The high-order 16 bits of the contents of RA are not equal to the high-order 16 bits of the 64-bit sign-extended SI field.
- The high-order 16 bits of the contents of RA are not equal to zero, and either the high-order 40 bits of the contents of RA are not equal to the high-order 40 bits of the 64-bit sign-extended SI field or the contents of RA are less than the sign-extended SI field.

In *tags active* mode with TO = 0b11110, if the high-order 40 bits of the contents of RA are not equal to the high-order 40 bits of the 64-bit sign-extended SI field, or if the contents of RA are less than or logically less than the sign-extended SI field, the system trap handler is invoked.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword Immediate*:

Extended:		*Equivalent to:*	
tdlti	Rx,value	tdi	16,Rx,value
tdnei	Rx,value	tdi	24,Rx,value

Trap Word Immediate D-form

twi TO,RA,SI

[POWER mnemonic: ti]

3	TO	RA	SI
0	6	11	16 31

```
a ← EXTS($(RA)_{32:63}$)
if (a < EXTS(SI)) & $TO_0$ then TRAP
if (a > EXTS(SI)) & $TO_1$ then TRAP
if (a = EXTS(SI)) & $TO_2$ then TRAP
if (a $<^u$ EXTS(SI)) & $TO_3$ then TRAP
if (a $>^u$ EXTS(SI)) & $TO_4$ then TRAP
```

The contents of $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

In *tags active* mode (see Book III, *PowerPC AS Operating Environment Architecture*), TO = 0b11100 and TO = 0b11110 are invalid forms.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word Immediate*:

Extended:		*Equivalent to:*	
twgti	Rx,value	twi	8,Rx,value
twllei	Rx,value	twi	6,Rx,value

Trap Doubleword X-form

td TO,RA,RB

31	TO	RA	RB	68	/
0	6	11	16	21	31

```
a ← (RA)
b ← (RB)
if ( $a_{0:15}$ ≠ $b_{0:15}$  |
     ( $a_{0:15}$ ≠ 0  &
     ( ( $a_{16:39}$ ≠ $b_{16:39}$) | (a < b) ) ) ) &
    TO = 0b11100 & (tags active) then TRAP
if ($a_{0:39}$ ≠ $b_{0:39}$) &
(TO = 0b11110) & (tags active) then TRAP
if (a < b) & $TO_0$ &
   ( TO ≠ 0b11100  | tags inactive) then TRAP
if (a > b) & $TO_1$ &
   ((TO ≠ 0b11100  &
     TO ≠ 0b11110) | tags inactive) then TRAP
if (a = b) & $TO_2$ &
   ((TO ≠ 0b11100  &
     TO ≠ 0b11110) | tags inactive) then TRAP
if (a $\overset{u}{<}$ b) & $TO_3$ then TRAP
if (a $\overset{u}{>}$ b) & $TO_4$ then TRAP
```

The contents of register RA are compared with the contents of register RB. In *tags inactive* mode or if TO is equal to any value other than 0b11100 or 0b11110, if any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

In *tags active* mode with TO = 0b11100 the system trap handler is invoked if any of the following conditions are met.

- The high-order 16 bits of the contents of RA are not equal to the high-order 16 bits of the contents of RB.
- The high-order 16 bits of the contents of RA are not equal to zero, and either the high-order 40 bits of the contents of RA are not equal to the high-order 40 bits of the contents of RB or the contents of RA are less than the contents of RB.

In *tags active* mode with TO = 0b11110, if the high-order 40 bits of the contents of RA are not equal to the high-order 40 bits of the contents of RB, or if the contents of RA are less than or logically less than the contents of RB, the system trap handler is invoked.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Doubleword*:

Extended:		*Equivalent to:*	
tdge	Rx,Ry	td	12,Rx,Ry
tdlnl	Rx,Ry	td	5,Rx,Ry

Trap Word X-form

tw TO,RA,RB

[POWER mnemonic: t]

31	TO	RA	RB	4	/
0	6	11	16	21	31

```
a ← EXTS($(RA)_{32:63}$)
b ← EXTS($(RB)_{32:63}$)
if (a < b) & $TO_0$ then TRAP
if (a > b) & $TO_1$ then TRAP
if (a = b) & $TO_2$ then TRAP
if (a $\overset{u}{<}$ b) & $TO_3$ then TRAP
if (a $\overset{u}{>}$ b) & $TO_4$ then TRAP
```

The contents of $RA_{32:63}$ are compared with the contents of $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

In *tags active* mode (see Book III, *PowerPC AS Operating Environment Architecture*), TO = 0b11100 and TO = 0b11110 are invalid forms.

Special Registers Altered:

None

Extended Mnemonics:

Examples of extended mnemonics for *Trap Word*:

Extended:		*Equivalent to:*	
tweq	Rx,Ry	tw	4,Rx,Ry
twlge	Rx,Ry	tw	5,Rx,Ry
trap		tw	31,0,0

Trap on XER TX-form

txer TO,UI,XBI

31	TO	UI	XBI	36	/
0	6	11	21	25	31

```
if XER$_{XBI+32}$ = TO$_4$ then TRAP
```

The XER bit at position XBI+32 is tested. If it equals TO_4, the system trap handler is invoked.

The UI field is ignored by the processor.

If TO is not 0 or 1, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Trap on XER*:

Extended:		*Equivalent to:*	
txereq	256	txer	1,256,38
txeric		txer	1,0,39
txerntag	5	txer	0,5,43

Programming Note

The UI field can be used to pass a parameter to the system trap handler. The system trap handler can examine the instruction that caused the trap to obtain this parameter. One use of this parameter is to indicate what function was being performed when the instruction was executed.

3.3.11 Fixed-Point Select Instructions

The *Select* instructions set a target register to one of two values, according to the value of a specified bit in the Fixed-Point Exception Register. Only bits 32 through 47 can be tested.

If Rc=1, the *Select* instructions set CR Field 0 and the FXCC according to the value in register RA at the completion of the instruction.

Architecture Note

For all four *Select* instructions, the result if XBI+32 = 1 is specified by instruction bits 6:10 and the result if XBI+32 = 0 is specified by instruction bits 16:20.

Programming Note

In some implementations, testing one of the first three bits of the FXCC may be faster than testing other XER bits.

Programming Note

The *Select* instructions are intended to be used to improve program execution speed by reducing branching. For example, they can be used, often after a *Compare* instruction, to implement the fixed-point minimum, maximum, and absolute value functions, to obtain 0/1 or 0/–1 values for relational expressions, and to implement certain simple forms of C conditional expressions and if-then-else constructions.

Extended mnemonics for selects

A set of extended mnemonics is provided so that selects can be coded with the condition as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the *Select* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Select Immediate-Immediate MDS-form

selii	RA,IS,IB,XBI	(Rc=0)
selii.	RA,IS,IB,XBI	(Rc=1)

30	IS	RA	IB	XBI	//	12	Rc
0	6	11	16	21	25	27	31

```
if $XER_{XBI+32}$ then RA ← EXTS(IS)
                  else RA ← EXTS(IB)
```

The XER bit at position XBI+32 is tested. If it is 1, register RA is set to the sign-extended value of IS. Otherwise register RA is set to the sign-extended value of IB.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Select Immediate-Immediate*:

Extended:		*Equivalent to:*	
seleqii	Rx,valy,valz	selii	Rx,valy,valz,38
seldsii	Rx,valy,valz	selii	Rx,valy,valz,40

Select Immediate-Register MDS-form

selir	RA,IS,RB,XBI	(Rc=0)
selir.	RA,IS,RB,XBI	(Rc=1)

30	IS	RA	RB	XBI	//	13	Rc
0	6	11	16	21	25	27	31

```
if $XER_{XBI+32}$ then RA ← EXTS(IS)
else RA ← (RB)
```

The XER bit at position XBI+32 is tested. If it is 1, register RA is set to the sign-extended value of IS. Otherwise register RA is set to (RB).

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Select Immediate-Register*:

Extended:		*Equivalent to:*	
selltir	Rx,valy,Rz	selir	Rx,valy,Rz,36
selcair	Rx,valy,Rz	selir	Rx,valy,Rz,34

Select Register-Immediate MDS-form

selri	RA,RS,IB,XBI	(Rc=0)
selri.	RA,RS,IB,XBI	(Rc=1)

30	RS	RA	IB	XBI	//	14	Rc
0	6	11	16	21	25	27	31

```
if XER$_{XBI+32}$ then RA ← (RS)
else RA ← EXTS(IB)
```

The XER bit at position XBI+32 is tested. If it is 1, register RA is set to (RS). Otherwise register RA is set to the sign-extended value of IB.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Select Register-Immediate*:

Extended:		*Equivalent to:*	
selgtri	Rx,Ry,valz	selri	Rx,Ry,valz,37
seltcri	Rx,Ry,valz	selri	Rx,Ry,valz,35

Select Register-Register MDS-form

selrr	RA,RS,RB,XBI	(Rc=0)
selrr.	RA,RS,RB,XBI	(Rc=1)

30	RS	RA	RB	XBI	//	15	Rc
0	6	11	16	21	25	27	31

```
if XER$_{XBI+32}$ then RA ← (RS)
else RA ← (RB)
```

The XER bit at position XBI+32 is tested. If it is 1, register RA is set to (RS). Otherwise register RA is set to (RB).

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Select Register-Register*:

Extended:		*Equivalent to:*	
selovrr	Rx,Ry,Rz	selrr	Rx,Ry,Rz,33
selicrr	Rx,Ry,Rz	selrr	Rx,Ry,Rz,39

3.3.12 Fixed-Point Logical Instructions

The *Logical* instructions perform bit-parallel operations on 64-bit operands.

The X-form *Logical* instructions with Rc=1, and the D-form *Logical* instructions ***andi.*** and ***andis.***, set the first three bits of CR Field 0 as described in Section 3.3.7, "Other Fixed-Point Instructions" on page 58. The *Logical* instructions do not change the SO, OV, and CA bits in the XER.

Extended mnemonics for logical operations

An extended mnemonic is provided that generates the preferred form of "no-op" (an instruction that does nothing). This is shown as an example with the *OR Immediate* instruction.

Extended mnemonics are provided that use the *OR* and *NOR* instructions to copy the contents of one register to another, with and without complementing. These are shown as examples with the two instructions.

See Appendix B, "Assembler Extended Mnemonics" on page 161 for additional extended mnemonics.

AND Immediate D-form

andi. RA,RS,UI

[POWER mnemonic: andil.]

28	RS	RA	UI	
0	6	11	16	31

RA ← (RS) & (^{48}0 || UI)

The contents of register RS are ANDed with ^{48}0 || UI and the result is placed into register RA.

Special Registers Altered:
CR0 FXCC

AND Immediate Shifted D-form

andis. RA,RS,UI

[POWER mnemonic: andiu.]

29	RS	RA	UI	
0	6	11	16	31

RA ← (RS) & (^{32}0 || UI || ^{16}0)

The contents of register RS are ANDed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:
CR0 FXCC

OR Immediate D-form

ori RA,RS,UI

[POWER mnemonic: oril]

24	RS	RA	UI
0	6	11	16 31

RA ← (RS) | (^{48}0 || UI)

The contents of register RS are ORed with ^{48}0 || UI and the result is placed into register RA.

The preferred “no-op” (an instruction that does nothing) is:

```
    ori  0,0,0
```

Special Registers Altered:
None

Extended Mnemonics:

Example of extended mnemonics for *OR Immediate*:

Extended:	*Equivalent to:*
nop	ori 0,0,0

Engineering Note

It is desirable for implementations to make the preferred form of no-op execute quickly, since this form should be used by compilers.

OR Immediate Shifted D-form

oris RA,RS,UI

[POWER mnemonic: oriu]

25	RS	RA	UI
0	6	11	16 31

RA ← (RS) | (^{32}0 || UI || ^{16}0)

The contents of register RS are ORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:
None

XOR Immediate D-form

xori RA,RS,UI

[POWER mnemonic: xoril]

26	RS	RA	UI
0	6	11	16 31

RA ← (RS) ⊕ (^{48}0 || UI)

The contents of register RS are XORed with ^{48}0 || UI and the result is placed into register RA.

Special Registers Altered:
None

XOR Immediate Shifted D-form

xoris RA,RS,UI

[POWER mnemonic: xoriu]

27	RS	RA	UI
0	6	11	16 31

RA ← (RS) ⊕ (^{32}0 || UI || ^{16}0)

The contents of register RS are XORed with ^{32}0 || UI || ^{16}0 and the result is placed into register RA.

Special Registers Altered:
None

AND X-form

and	RA,RS,RB	(Rc=0)
and.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	28	Rc
0	6	11	16	21	31

```
RA ← (RS) & (RB)
```

The contents of register RS are ANDed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 FXCC (if Rc=1)

OR X-form

or	RA,RS,RB	(Rc=0)
or.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	444	Rc
0	6	11	16	21	31

```
RA ← (RS) | (RB)
```

The contents of register RS are ORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 FXCC (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *OR*:

Extended:		*Equivalent to:*	
mr	Rx,Ry	or	Rx,Ry,Ry

XOR X-form

xor	RA,RS,RB	(Rc=0)
xor.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	316	Rc
0	6	11	16	21	31

```
RA ← (RS) ⊕ (RB)
```

The contents of register RS are XORed with the contents of register RB and the result is placed into register RA.

Special Registers Altered:
CR0 FXCC (if Rc=1)

NAND X-form

nand	RA,RS,RB	(Rc=0)
nand.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	476	Rc
0	6	11	16	21	31

```
RA ← ¬((RS) & (RB))
```

The contents of register RS are ANDed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:
CR0 FXCC (if Rc=1)

Programming Note

nand or ***nor*** with RS=RB can be used to obtain the one's complement.

NOR X-form

nor	RA,RS,RB	(Rc=0)
nor.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	124	Rc
0	6	11	16	21	31

```
RA ← ¬((RS) | (RB))
```

The contents of register RS are ORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *NOR*:

Extended:		*Equivalent to:*	
not	Rx,Ry	nor	Rx,Ry,Ry

Equivalent X-form

eqv	RA,RS,RB	(Rc=0)
eqv.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	284	Rc
0	6	11	16	21	31

```
RA ← (RS) ≡ (RB)
```

The contents of register RS are XORed with the contents of register RB and the complemented result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

AND with Complement X-form

andc	RA,RS,RB	(Rc=0)
andc.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	60	Rc
0	6	11	16	21	31

```
RA ← (RS) & ¬(RB)
```

The contents of register RS are ANDed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

OR with Complement X-form

orc	RA,RS,RB	(Rc=0)
orc.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	412	Rc
0	6	11	16	21	31

```
RA ← (RS) | ¬(RB)
```

The contents of register RS are ORed with the complement of the contents of register RB and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extend Sign Byte X-form

extsb	RA,RS	(Rc=0)
extsb.	RA,RS	(Rc=1)

31	RS	RA	///	954	Rc
0	6	11	16	21	31

```
s ← $(RS)_{56}$
$RA_{56:63}$ ← $(RS)_{56:63}$
$RA_{0:55}$ ← $^{56}s$
```

$(RS)_{56:63}$ are placed into $RA_{56:63}$. Bit 56 of register RS is placed into $RA_{0:55}$.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extend Sign Halfword X-form

extsh	RA,RS	(Rc=0)
extsh.	RA,RS	(Rc=1)

[POWER mnemonics: exts, exts.]

31	RS	RA	///	922	Rc
0	6	11	16	21	31

```
s ← $(RS)_{48}$
$RA_{48:63}$ ← $(RS)_{48:63}$
$RA_{0:47}$ ← $^{48}s$
```

$(RS)_{48:63}$ are placed into $RA_{48:63}$. Bit 48 of register RS is placed into $RA_{0:47}$.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extend Sign Word X-form

extsw	RA,RS	(Rc=0)
extsw.	RA,RS	(Rc=1)

31	RS	RA	///	986	Rc
0	6	11	16	21	31

```
s ← $(RS)_{32}$
$RA_{32:63}$ ← $(RS)_{32:63}$
$RA_{0:31}$ ← $^{32}s$
```

$(RS)_{32:63}$ are placed into $RA_{32:63}$. Bit 32 of register RS is placed into $RA_{0:31}$.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Count Leading Zeros Doubleword X-form

cntlzd	RA,RS	(Rc=0)
cntlzd.	RA,RS	(Rc=1)

31	RS	RA	///	58	Rc
0	6	11	16	21	31

```
n ← 0
do while n < 64
  if (RS)_n = 1 then leave
  n ← n + 1
RA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of register RS is placed into RA. This number ranges from 0 to 64, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Count Leading Zeros Word X-form

cntlzw	RA,RS	(Rc=0)
cntlzw.	RA,RS	(Rc=1)

[POWER mnemonics: cntlz, cntlz.]

31	RS	RA	///	26	Rc
0	6	11	16	21	31

```
n ← 32
do while n < 64
  if (RS)_n = 1 then leave
  n ← n + 1
RA ← n – 32
```

A count of the number of consecutive zero bits starting at bit 32 of register RS is placed into RA. This number ranges from 0 to 32, inclusive.

If Rc=1, CR Field 0 is set to reflect the result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Programming Note

For both *Count Leading Zeros* instructions, if Rc=1 then LT is set to 0 in CR Field 0 and in the FXCC.

3.3.13 Fixed-Point Rotate and Shift Instructions

The Fixed-Point Processor performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR.

The rotation operations rotate a 64-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 63.

Two types of rotation operation are supported.

For the first type, denoted rotate_{64} or ROTL_{64}, the value rotated is the given 64-bit value. The rotate_{64} operation is used to rotate a given 64-bit quantity.

For the second type, denoted rotate_{32} or ROTL_{32}, the value rotated consists of two copies of bits 32:63 of the given 64-bit value, one copy in bits 0:31 and the other in bits 32:63. The rotate_{32} operation is used to rotate a given 32-bit quantity.

The *Rotate and Shift* instructions employ a mask generator. The mask is 64 bits long, and consists of 1-bits from a start bit, *mstart*, through and including a stop bit, *mstop*, and 0-bits elsewhere. The values of *mstart* and *mstop* range from 0 to 63. If *mstart* > *mstop*, the 1-bits wrap around from position 63 to position 0. Thus the mask is formed as follows:

```
if mstart ≤ mstop then
    $\text{mask}_{\text{mstart:mstop}}$ = ones
    $\text{mask}_{\text{all other bits}}$ = zeros
else
    $\text{mask}_{\text{mstart:63}}$ = ones
    $\text{mask}_{\text{0:mstop}}$ = ones
    $\text{mask}_{\text{all other bits}}$ = zeros
```

There is no way to specify an all-zero mask.

For instructions that use the rotate_{32} operation, the mask start and stop positions are always in the low-order 32 bits of the mask.

The use of the mask is described in following sections.

The *Rotate and Shift* instructions with Rc=1 set the first three bits of CR field 0 as described in Section 3.3.7, “Other Fixed-Point Instructions” on page 58. *Rotate and Shift* instructions do not change the OV and SO bits. *Rotate and Shift* instructions, except algebraic right shifts, do not change the CA bit.

Extended mnemonics for rotates and shifts

The *Rotate and Shift* instructions, while powerful, can be complicated to code (they have up to five operands). A set of extended mnemonics is provided that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and performing simple rotates and shifts. Some of these are shown as examples with the *Rotate* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

3.3.13.1 Fixed-Point Rotate Instructions

These instructions rotate the contents of a register. The result of the rotation is

- inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged); or
- ANDed with a mask before being placed into the target register.

The *Rotate Left* instructions allow right-rotation of the contents of a register to be performed (in concept) by a left-rotation of 64− *n*, where *n* is the number of bits by which to rotate right. They allow right-rotation of the contents of the low-order 32 bits of a register to be performed (in concept) by a left-rotation of 32− *n*, where *n* is the number of bits by which to rotate right.

Architecture Note

For MD-form and MDS-form instructions, the MB and ME fields are used in permuted rather than sequential order because this is easier for the processor. Permuting the MB field permits the processor to obtain the low-order five bits of the MB value from the same place for all instructions having an MB field (M-form and MD-form instructions). Permuting the ME field permits the processor to treat bits 21:26 of all MD-form instructions uniformly.

Rotate Left Doubleword Immediate then Clear Left MD-form

rldicl	RA,RS,SH,MB	(Rc=0)
rldicl.	RA,RS,SH,MB	(Rc=1)

30	RS	RA	sh	mb	0	sh	Rc
0	6	11	16	21	27	30	31

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((RS), n)
b ← $mb_5$ || $mb_{0:4}$
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Left*:

Extended:		*Equivalent to:*	
extrdi	Rx,Ry,n,b	rldicl	Rx,Ry,b+n,64−n
srdi	Rx,Ry,n	rldicl	Rx,Ry,64−n,n
clrldi	Rx,Ry,n	rldicl	Rx,Ry,0,n

Programming Note

rldicl can be used to extract an *n*-bit field that starts at bit position *b* in register RS, right-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting SH=*b*+*n* and MB=64−*n*. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64−*n*) and MB=0. It can be used to shift the contents of a register right by *n* bits, by setting SH=64−*n* and MB=*n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*.

Extended mnemonics are provided for all of these uses: see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Doubleword Immediate then Clear Right MD-form

rldicr	RA,RS,SH,ME	(Rc=0)
rldicr.	RA,RS,SH,ME	(Rc=1)

30	RS	RA	sh	me	1	sh	Rc
0	6	11	16	21	27	30	31

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((RS), n)
e ← $me_5$ || $me_{0:4}$
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Doubleword Immediate then Clear Right*:

Extended:		*Equivalent to:*	
extldi	Rx,Ry,n,b	rldicr	Rx,Ry,b,n−1
sldi	Rx,Ry,n	rldicr	Rx,Ry,n,63−n
clrrdi	Rx,Ry,n	rldicr	Rx,Ry,0,63−n

Programming Note

rldicr can be used to extract an *n*-bit field that starts at bit position *b* in register RS, left-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting SH=*b* and ME=*n*−1. It can be used to rotate the contents of a register left (right) by *n* bits, by setting SH=*n* (64−*n*) and ME=63. It can be used to shift the contents of a register left by *n* bits, by setting SH=*n* and ME=63−*n*. It can be used to clear the low-order *n* bits of a register, by setting SH=0 and ME=63−*n*.

Extended mnemonics are provided for all of these uses (some devolve to ***rldicl***): see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Doubleword Immediate then Clear MD-form

rldic	RA,RS,SH,MB	(Rc=0)
rldic.	RA,RS,SH,MB	(Rc=1)

30	RS	RA	sh	mb	2	sh	Rc
0	6	11	16	21	27	30	31

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((RS), n)
b ← $mb_5$ || $mb_{0:4}$
m ← MASK(b, ¬n)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63−SH and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Clear*:

Extended:		*Equivalent to:*	
clrlsldi	Rx,Ry,b,n	rldic	Rx,Ry,n,b−n

Programming Note

rldic can be used to clear the high-order *b* bits of the contents of a register and then shift the result left by *n* bits, by setting SH=*n* and MB=*b*−*n*. It can be used to clear the high-order *n* bits of a register, by setting SH=0 and MB=*n*.

Extended mnemonics are provided for both of these uses (the second devolves to ***rldicl***): see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Word Immediate then AND with Mask M-form

rlwinm	RA,RS,SH,MB,ME	(Rc=0)
rlwinm.	RA,RS,SH,MB,ME	(Rc=1)

[POWER mnemonics: rlinm, rlinm.]

21	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← $ROTL_{32}$($(RS)_{32:63}$, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Extended Mnemonics:

Examples of extended mnemonics for *Rotate Left Word Immediate then AND with Mask*:

Extended:		*Equivalent to:*
extlwi	Rx,Ry,n,b	rlwinm Rx,Ry,b,0,n−1
srwi	Rx,Ry,n	rlwinm Rx,Ry,32−n,n,31
clrrwi	Rx,Ry,n	rlwinm Rx,Ry,0,0,31−n

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwinm can be used to extract an *n*-bit field that starts at bit position *b* in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32− *n* bits of the low-order 32 bits of RA), by setting SH=*b*+ *n*, MB=32− *n*, and ME=31. It can be used to extract an *n*-bit field that starts at bit position *b* in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32− *n* bits of the low-order 32 bits of RA), by setting SH=*b*, MB = 0, and ME=*n*− 1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by *n* bits, by setting SH=*n* (32− *n*), MB=0, and ME=31. It can be used to shift the contents of the low-order 32 bits of a register right by *n* bits, by setting SH=32− *n*, MB=*n*, and ME=31. It can be used to clear the high-order *b* bits of the low-order 32 bits of the contents of a register and then shift the result left by *n* bits, by setting SH=*n*, MB=*b*− *n* and ME=31− *n*. It can be used to clear the low-order *n* bits of the low-order 32 bits of a register, by setting SH=0, MB=0, and ME=31− *n*.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for all of these uses: see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Doubleword then Clear Left MDS-form

rldcl	RA,RS,RB,MB	(Rc=0)
rldcl.	RA,RS,RB,MB	(Rc=1)

30	RS	RA	RB	mb	8	Rc
0	6	11	16	21	27	31

```
n ← (RB)$_{58:63}$
r ← ROTL$_{64}$((RS), n)
b ← mb$_5$ || mb$_{0:4}$
m ← MASK(b, 63)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit MB through bit 63 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword then Clear Left*:

Extended:		*Equivalent to:*	
rotld	Rx,Ry,Rz	rldcl	Rx,Ry,Rz,0

Programming Note

rldcl can be used to extract an *n*-bit field that starts at variable bit position *b* in register RS, right-justified into register RA (clearing the remaining 64− *n* bits of RA), by setting RB$_{58:63}$= *b*+ *n* and MB=64− *n*. It can be used to rotate the contents of a register left (right) by variable *n* bits, by setting RB$_{58:63}$= *n* (64− *n*) and MB=0.

Extended mnemonics are provided for some of these uses: see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Doubleword then Clear Right MDS-form

rldcr	RA,RS,RB,ME	(Rc=0)
rldcr.	RA,RS,RB,ME	(Rc=1)

30	RS	RA	RB	me	9	Rc
0	6	11	16	21	27	31

```
n ← (RB)$_{58:63}$
r ← ROTL$_{64}$((RS), n)
e ← me$_5$ || me$_{0:4}$
m ← MASK(0, e)
RA ← r & m
```

The contents of register RS are rotated$_{64}$ left the number of bits specified by (RB)$_{58:63}$. A mask is generated having 1-bits from bit 0 through bit ME and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Programming Note

rldcr can be used to extract an *n*-bit field that starts at variable bit position *b* in register RS, left-justified into register RA (clearing the remaining 64−*n* bits of RA), by setting RB$_{58:63}$= *b* and ME=*n*−1. It can be used to rotate the contents of a register left (right) by variable *n* bits, by setting RB$_{58:63}$= *n* (64−*n*) and ME=63.

Extended mnemonics are provided for some of these uses (some devolve to ***rldcl***) see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Word then AND with Mask M-form

rlwnm	RA,RS,RB,MB,ME	(Rc=0)
rlwnm.	RA,RS,RB,MB,ME	(Rc=1)

[POWER mnemonics: rlnm, rlnm.]

23	RS	RA	RB	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← (RB)$_{59:63}$
r ← ROTL$_{32}$((RS)$_{32:63}$, n)
m ← MASK(MB+32, ME+32)
RA ← r & m
```

The contents of register RS are rotated$_{32}$ left the number of bits specified by (RB)$_{59:63}$. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are ANDed with the generated mask and the result is placed into register RA.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word then AND with Mask*:

Extended:	*Equivalent to:*
rotlw Rx,Ry,Rz	rlwnm Rx,Ry,Rz,0,31

Programming Note

Let RSL represent the low-order 32 bits of register RS, with the bits numbered from 0 through 31.

rlwnm can be used to extract an *n*-bit field that starts at variable bit position *b* in RSL, right-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting RB$_{59:63}$= *b*+*n*, MB=32−*n*, and ME=31. It can be used to extract an *n*-bit field that starts at variable bit position *b* in RSL, left-justified into the low-order 32 bits of register RA (clearing the remaining 32−*n* bits of the low-order 32 bits of RA), by setting RB$_{59:63}$= *b*, MB = 0, and ME=*n*−1. It can be used to rotate the contents of the low-order 32 bits of a register left (right) by variable *n* bits, by setting RB$_{59:63}$= *n* (32−*n*), MB=0, and ME=31.

For all the uses given above, the high-order 32 bits of register RA are cleared.

Extended mnemonics are provided for some of these uses: see Appendix B, “Assembler Extended Mnemonics” on page 161.

Rotate Left Doubleword Immediate then Mask Insert MD-form

rldimi	RA,RS,SH,MB	(Rc=0)
rldimi.	RA,RS,SH,MB	(Rc=1)

30	RS	RA	sh	mb	3	sh	Rc
0	6	11	16	21	27	30	31

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((RS), n)
b ← $mb_5$ || $mb_{0:4}$
m ← MASK(b, ¬n)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{64}$ left SH bits. A mask is generated having 1-bits from bit MB through bit 63− SH and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Doubleword Immediate then Mask Insert*:

Extended:	*Equivalent to:*
insrdi Rx,Ry,n,b	rldimi Rx,Ry,64− (b+n),b

Programming Note

rldimi can be used to insert an *n*-bit field that is right-justified in register RS, into register RA starting at bit position *b*, by setting SH=64− (*b*+ *n*) and MB=*b*.

An extended mnemonic is provided for this use: see Appendix B, "Assembler Extended Mnemonics" on page 161.

Rotate Left Word Immediate then Mask Insert M-form

rlwimi	RA,RS,SH,MB,ME	(Rc=0)
rlwimi.	RA,RS,SH,MB,ME	(Rc=1)

[POWER mnemonics: rlimi, rlimi.]

20	RS	RA	SH	MB	ME	Rc
0	6	11	16	21	26	31

```
n ← SH
r ← $ROTL_{32}$($(RS)_{32:63}$, n)
m ← MASK(MB+32, ME+32)
RA ← r&m | (RA)&¬m
```

The contents of register RS are rotated$_{32}$ left SH bits. A mask is generated having 1-bits from bit MB+32 through bit ME+32 and 0-bits elsewhere. The rotated data are inserted into register RA under control of the generated mask.

Special Registers Altered:

CR0 FXCC (if Rc=1)

Extended Mnemonics:

Example of extended mnemonics for *Rotate Left Word Immediate then Mask Insert*:

Extended:	*Equivalent to:*
inslwi Rx,Ry,n,b	rlwimi Rx,Ry,32− b,b,b+n− 1

Programming Note

Let RAL represent the low-order 32 bits of register RA, with the bits numbered from 0 through 31.

rlwimi can be used to insert an *n*-bit field that is left-justified in the low-order 32 bits of register RS, into RAL starting at bit position *b*, by setting SH=32− *b*, MB=*b*, and ME=(*b*+*n*)− 1. It can be used to insert an *n*-bit field that is right-justified in the low-order 32 bits of register RS, into RAL starting at bit position *b*, by setting SH=32− (*b*+ *n*), MB=*b*, and ME=(*b*+*n*)− 1.

Extended mnemonics are provided for both of these uses: see Appendix B, "Assembler Extended Mnemonics" on page 161.

3.3.13.2 Fixed-Point Shift Instructions

The instructions in this section perform left and right shifts.

Extended mnemonics for shifts

Immediate-form logical (unsigned) shift operations are obtained by specifying appropriate masks and shift values for certain *Rotate* instructions. A set of extended mnemonics is provided to make coding of such shifts simpler and easier to understand. Some of these are shown as examples with the *Rotate* instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Programming Note

Any *Shift Right Algebraic* instruction, followed by ***addze***, can be used to divide quickly by 2^n. The setting of the CA bit by the *Shift Right Algebraic* instructions is independent of mode.

Programming Note

Multiple-precision shifts can be programmed as shown in Section C.1, “Multiple-Precision Shifts” on page 177.

Engineering Note

The instructions intended for use with 32-bit data are shown as doing a rotate_{32} operation. This is strictly necessary only for setting the CA bit for ***srawi*** and ***sraw***. ***slw*** and ***srw*** could do a rotate_{64} operation if that is easier.

Shift Left Doubleword X-form

sld	RA,RS,RB	(Rc=0)
sld.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	27	Rc
0	6	11	16	21	31

```
n ← $(RB)_{58:63}$
r ← $ROTL_{64}((RS), n)$
if $(RB)_{57}$ = 0 then
    m ← MASK(0, 63–n)
else m ← $^{64}0$
RA ← r & m
```

The contents of register RS are shifted left the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Shift Left Word X-form

slw	RA,RS,RB	(Rc=0)
slw.	RA,RS,RB	(Rc=1)

[POWER mnemonics: sl, sl.]

31	RS	RA	RB	24	Rc
0	6	11	16	21	31

```
n ← $(RB)_{59:63}$
r ← $ROTL_{32}((RS)_{32:63}, n)$
if $(RB)_{58}$ = 0 then
    m ← MASK(32, 63–n)
else m ← $^{64}0$
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted left the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Shift Right Doubleword X-form

srd	RA,RS,RB	(Rc=0)
srd.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	539	Rc
0	6	11	16	21	31

```
n ← (RB)$_{58:63}$
r ← ROTL$_{64}$((RS), 64−n)
if (RB)$_{57}$ = 0 then
    m ← MASK(n, 63)
else m ← $^{64}$0
RA ← r & m
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into register RA. Shift amounts from 64 to 127 give a zero result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Shift Right Word X-form

srw	RA,RS,RB	(Rc=0)
srw.	RA,RS,RB	(Rc=1)

[POWER mnemonics: sr, sr.]

31	RS	RA	RB	536	Rc
0	6	11	16	21	31

```
n ← (RB)$_{59:63}$
r ← ROTL$_{32}$((RS)$_{32:63}$, 64−n)
if (RB)$_{58}$ = 0 then
    m ← MASK(n+32, 63)
else m ← $^{64}$0
RA ← r & m
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. $RA_{0:31}$ are set to zero. Shift amounts from 32 to 63 give a zero result.

Special Registers Altered:

CR0 FXCC	(if Rc=1)

Shift Right Algebraic Doubleword Immediate XS-form

sradi	RA,RS,SH	(Rc=0)
sradi.	RA,RS,SH	(Rc=1)

31	RS	RA	sh	413	sh	Rc
0	6	11	16	21	30	31

```
n ← $sh_5 \| sh_{0:4}$
r ← $ROTL_{64}((RS), 64-n)$
m ← MASK(n, 63)
s ← $(RS)_0$
RA ← r&m | ($^{64}$s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)

Shift Right Algebraic Word Immediate X-form

srawi	RA,RS,SH	(Rc=0)
srawi.	RA,RS,SH	(Rc=1)

[POWER mnemonics: srai, srai.]

31	RS	RA	SH	824	Rc
0	6	11	16	21	31

```
n ← SH
r ← $ROTL_{32}((RS)_{32:63}, 64-n)$
m ← MASK(n+32, 63)
s ← $(RS)_{32}$
RA ← r&m | ($^{64}$s)&¬m
CA ← s & $((r\&\neg m)_{32:63} \neq 0)$
```

The contents of the low-order 32 bits of register RS are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)

Shift Right Algebraic Doubleword X-form

srad	RA,RS,RB	(Rc=0)
srad.	RA,RS,RB	(Rc=1)

31	RS	RA	RB	794	Rc
0	6	11	16	21	31

```
n ← (RB)$_{58:63}$
r ← ROTL$_{64}$((RS), 64−n)
if (RB)$_{57}$ = 0 then
    m ← MASK(n, 63)
else m ← $^{64}$0
s ← (RS)$_0$
RA ← r&m | ($^{64}$s)&¬m
CA ← s & ((r&¬m)≠0)
```

The contents of register RS are shifted right the number of bits specified by $(RB)_{57:63}$. Bits shifted out of position 63 are lost. Bit 0 of RS is replicated to fill the vacated positions on the left. The result is placed into register RA. CA is set to 1 if (RS) is negative and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to be set equal to (RS), and CA to be set to 0. Shift amounts from 64 to 127 give a result of 64 sign bits in RA, and cause CA to receive the sign bit of (RS).

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)

Shift Right Algebraic Word X-form

sraw	RA,RS,RB	(Rc=0)
sraw.	RA,RS,RB	(Rc=1)

[POWER mnemonics: sra, sra.]

31	RS	RA	RB	792	Rc
0	6	11	16	21	31

```
n ← (RB)$_{59:63}$
r ← ROTL$_{32}$((RS)$_{32:63}$, 64−n)
if (RB)$_{58}$ = 0 then
    m ← MASK(n+32, 63)
else m ← $^{64}$0
s ← (RS)$_{32}$
RA ← r&m | ($^{64}$s)&¬m
CA ← s & ((r&¬m)$_{32:63}$≠0)
```

The contents of the low-order 32 bits of register RS are shifted right the number of bits specified by $(RB)_{58:63}$. Bits shifted out of position 63 are lost. Bit 32 of RS is replicated to fill the vacated positions on the left. The 32-bit result is placed into $RA_{32:63}$. Bit 32 of RS is replicated to fill $RA_{0:31}$. CA is set to 1 if the low-order 32 bits of (RS) contain a negative number and any 1-bits are shifted out of position 63; otherwise CA is set to 0. A shift amount of zero causes RA to receive $EXTS((RS)_{32:63})$, and CA to be set to 0. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive the sign bit of $(RS)_{32:63}$.

Special Registers Altered:

CA	
CR0 FXCC	(if Rc=1)

3.3.14 Decimal Assist Instructions

For the *Decimal Assist* instructions, the affected General Purpose Registers are considered to contain packed decimal numbers, formatted as follows.

The register is considered to consist of 16 4-bit fields, numbered from 0 through 15 starting at the high-order end of the register. Field *n* consists of bits $4 \times n$ through $4 \times n+3$. Fields 0:14 each contain a decimal digit, while field 15 can contain either a decimal digit or a sign. Increasing field number corresponds to decreasing digit significance. A decimal digit can have any value from 0 through 9. A sign can have any value: a sign value of 0xB or 0xD represents a minus sign, and any other value represents a plus sign.

Decimal Sixes X-form

dsixes RA

31	///	RA	///	61	/
0	6	11	16	21	31

```
c ← $^4(DC_0)$ || $^4(DC_1)$ || ... || $^4(DC_{15})$
RA ← (¬c) & 0x6666_6666_6666_6666
```

A doubleword is composed from the Decimal Carry bits in the XER, and placed into RA. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding bit in XER_{DC} is zero, and a zero (0b0000) in every position for which the corresponding bit in XER_{DC} is one. Bit *i* of XER_{DC} corresponds to decimal digit position *i* of RA, for *i* = 0, 1, ..., 15.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
None

Decimal Test and Clear Sign X-form

dtcs. RA,RS

31	RS	RA	///	93	1
0	6	11	16	21	31

```
s ← $(RS)_{60}$ & ($(RS)_{61}$ ⊕ $(RS)_{62}$) & $(RS)_{63}$
if s = 1 then $CR_{0:3}$ ← 0b100 || $XER_{SO}$
              FXCC ← 0b1000
         else $CR_{0:3}$ ← 0b010 || $XER_{SO}$
              FXCC ← 0b0100
RA ← $(RS)_{0:59}$ || 0b0000
CA ← 0
```

CR0 and the FXCC are set to reflect “Less Than” if the sign in the low-order four bits of (RS) is 0xB or 0xD, and to reflect “Greater Than” otherwise. $RA_{0:59}$ is set to $(RS)_{0:59}$. $RA_{60:63}$ are set to 0. XER_{CA} is set to 0.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
CA CR0 FXCC

3.3.15 Move To/From System Register Instructions

| The *Move To Condition Register Fields* instruction has a preferred form: see Section 1.9.1, “Preferred Instruction Forms” on page 14. In the preferred form, the FXM field satisfies the following rule.

- Exactly one bit of the FXM field is set to 1.

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. An extended mnemonic is provided for the ***mtcrf*** instruction for compatibility with old software (written for a version of the architecture that precedes Version 2.00) that uses it to set the entire Condition Register. Some of these extended mnemonics are shown as examples with the relevant instructions. See Appendix B, “Assembler Extended Mnemonics” on page 161 for additional extended mnemonics.

Move To Special Purpose Register XFX-form

mtspr SPR,RS

31	RS	spr	467	/
0	6	11	21	31

```
n ← spr$_{5:9}$ || spr$_{0:4}$
if length(SPREG(n)) = 64 then
  SPREG(n) ← (RS)
else
  SPREG(n) ← (RS)$_{32:63\{0:31\}}$
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. 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.

SPR* decimal	$spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR

* Note that the order of the two 5-bit halves of the SPR number is reversed.

If the SPR field contains any value other than one of the values shown above then one of the following occurs.

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

A complete description of this instruction can be found in Book III, *PowerPC AS Operating Environment Architecture*.

Special Registers Altered:
See above

Extended Mnemonics:

Examples of extended mnemonics for *Move To Special Purpose Register*:

Extended:		*Equivalent to:*	
mtxer	Rx	mtspr	1,Rx
mtlr	Rx	mtspr	8,Rx
mtctr	Rx	mtspr	9,Rx

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.

Compatibility Note

For a discussion of POWER compatibility with respect to SPR numbers not shown in the instruction descriptions for ***mtspr*** and ***mfspr***, see Appendix E, “Incompatibilities with the POWER Architecture” on page 185.

Engineering Note

If $MSR_{PR}=1$, the only effect of executing this instruction with an SPR number in which $spr_0=1$ is to cause either an Illegal Instruction type Program interrupt or a Privileged Instruction type Program interrupt.

Engineering Note

Any assignment of SPR numbers not shown in the Book I instruction descriptions for ***mtspr*** and ***mfspr*** must be done in a manner consistent with the section that describes these instructions in Book III. †

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← $spr_{5:9}$ || $spr_{0:4}$
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 the table below. 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.

decimal	**SPR*** $\mathbf{spr_{5:9}}$ $\mathbf{spr_{0:4}}$	**Register Name**
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR

* Note that the order of the two 5-bit halves of the SPR number is reversed.

If the SPR field contains any value other than one of the values shown above then one of the following occurs.

- The system illegal instruction error handler is invoked.
- The system privileged instruction error handler is invoked.
- The results are boundedly undefined.

A complete description of this instruction can be found in Book III, *PowerPC AS Operating Environment Architecture*.

Special Registers Altered:
None

Extended Mnemonics:

Examples of extended mnemonics for *Move From Special Purpose Register*:

Extended:	*Equivalent to:*
mfxer Rx	mfspr Rx,1
mflr Rx	mfspr Rx,8
mfctr Rx	mfspr Rx,9

Note

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

Set XER TAG XFX-form

settag

31	///	XO2	499	/
0	6	11	21	31

```
$XER_{43}$ ← 1
```

Bit 43 of the XER is set to 1.

If the XO2 field contains any value other than 32, the instruction form is invalid.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
TAG

Move To Condition Register Fields XFX-form

mtcrf FXM,RS

31	RS	0	FXM	/	144	/
0	6	11	12	20	21	31

```
mask ← $^4(FXM_0) \,||\, ^4(FXM_1) \,||\, ...\, ^4(FXM_7)$
CR ← $((RS)_{32:63}$ & mask) | (CR & ¬mask)
```

The contents of bits 32:63 of register RS are placed into the Condition Register under control of the field mask specified by FXM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If $FXM_i = 1$ then CR field i (CR bits $4\times i:4\times i+3$) is set to the contents of the corresponding field of the low-order 32 bits of RS.

Special Registers Altered:
CR fields selected by mask

Extended Mnemonics:

Example of extended mnemonics for *Move To Condition Register Fields*:

Extended:	*Equivalent to:*
mtcr Rx	mtcrf 0xFF,Rx

Programming Note

In the preferred form of this instruction (see the introduction to Section 3.3.15), only one Condition Register field is updated.

Engineering Note

See the description of the optional version of ***mtcrf*** in Section 5.1.1 for additional information about this instruction.

Move to Condition Register from XER X-form

mcrxr BF

31	BF	//	///	///	512	/
0	6	9	11	16	21	31

```
$CR_{4\times BF:4\times BF+3}$ ← $XER_{32:35}$
$XER_{32:35}$ ← 0b0000
```

The contents of $XER_{32:35}$ are copied to Condition Register field BF. $XER_{32:35}$ are set to zero.

Special Registers Altered:
CR field BF XER $_{32:35}$

Move From Condition Register XFX-form

mfcr RT

31	RT	0	///	19	/
0	6	11	12	21	31

```
RT ← $^{32}0$ || CR
```

The contents of the Condition Register are placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:
None

Engineering Note

See the description of the optional version of ***mfcr*** in Section 5.1.1 for additional information about this instruction.

Move to Condition Register from XER TGCC X-form

mcrxrt BF

31	BF	//	///	///	544	/
0	6	9	11	16	21	31

```
$CR_{4\times BF:4\times BF+3}$ ← 0b0 || $XER_{41:43}$
```

0b0 concatenated with the contents of $XER_{41:43}$ is copied into the Condition Register field designated by BF.

In *tags inactive* mode, this instruction is an illegal instruction and an attempt to execute this instruction will invoke the system illegal instruction error handler.

Special Registers Altered:
CR field BF

Chapter 4. Floating-Point Processor

4.1 Floating-Point Processor Overview

This chapter describes the registers and instructions that make up the Floating-Point Processor facility. Section 4.2, “Floating-Point Processor Registers” on page 100 describes the registers associated with the Floating-Point Processor. Section 4.6, “Floating-Point Processor Instructions” on page 116 describes the instructions associated with the Floating-Point Processor.

This architecture specifies that the processor implement a floating-point system as defined in ANSI/IEEE Standard 754-1985, “IEEE Standard for Binary Floating-Point Arithmetic” (hereafter referred to as “the IEEE standard”), but requires software support in order to conform fully with that standard. That standard defines certain required “operations” (addition, subtraction, etc.); the term “floating-point operation” is used in this chapter to refer to one of these required operations, or to the operation performed by one of the *Multiply-Add* or *Reciprocal Estimate* instructions. All floating-point operations conform to that standard.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the Floating-Point Status and Control Register explicitly.

These instructions are divided into two categories.

- computational instructions

  The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.5 through 4.6.7 and Section 5.2.1.

- non-computational instructions

  The non-computational instructions are those that perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the Floating-Point Status and Control Register explicitly, and select the value from one of two floating-point registers based on the value in a third floating-point register. The operations performed by these instructions are not considered floating-point operations. With the exception of the instructions that manipulate the Floating-Point Status and Control Register explicitly, they do not alter the Floating-Point Status and Control Register. They are the instructions described in Sections 4.6.2 through 4.6.4, 4.6.8, and 5.2.2.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number 2^{exponent}. Encodings are provided in the data format to represent finite numeric values, ± Infinity, and values that are "Not a Number" (NaN). Operations involving infinities produce results obeying traditional mathematical conventions. NaNs have no mathematical interpretation. Their encoding permits a variable diagnostic information field. They may be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

There is one class of exceptional events that occur during instruction execution that is unique to the Floating-Point Processor: the Floating-Point Exception. Floating-point exceptions are signaled with bits set in the Floating-Point Status and Control Register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked, precisely or imprecisely, if the proper control bits are set.

Floating-Point Exceptions

The following floating-point exceptions are detected by the processor:

■ Invalid Operation Exception	(VX)
SNaN	(VXSNAN)
Infinity − Infinity	(VXISI)
Infinity ÷ Infinity	(VXIDI)
Zero ÷ Zero	(VXZDZ)
Infinity × Zero	(VXIMZ)
Invalid Compare	(VXVC)
Software Request	(VXSOFT)
Invalid Square Root	(VXSQRT)
Invalid Integer Convert	(VXCVI)
■ Zero Divide Exception	(ZX)
■ Overflow Exception	(OX)
■ Underflow Exception	(UX)
■ Inexact Exception	(XX)

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. See Section 4.2.2, "Floating-Point Status and Control Register" on page 101 for a description of these exception and enable bits, and Section 4.4, "Floating-Point Exceptions" on page 108 for a detailed discussion of floating-point exceptions, including the effects of the enable bits.

4.2 Floating-Point Processor Registers

4.2.1 Floating-Point Registers

Implementations of this architecture provide 32 floating-point registers (FPRs). The floating-point instruction formats provide 5-bit fields for specifying the FPRs to be used in the execution of the instruction. The FPRs are numbered 0-31. See Figure 28 on page 101.

Each FPR contains 64 bits that support the floating-point double format. Every instruction that interprets the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the *Move* and *Select* instructions, operate on data located in FPRs and, with the exception of the *Compare* instructions, place the result value into an FPR and optionally place status information into the Condition Register.

Load Double and *Store Double* instructions are provided that transfer 64 bits of data between storage and the FPRs with no conversion. *Load Single* instructions are provided to transfer and convert floating-point values in floating-point single format from storage to the same value in floating-point double format in the FPRs. *Store Single* instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in storage.

Instructions are provided that manipulate the Floating-Point Status and Control Register and the Condition Register explicitly. Some of these instructions copy data from an FPR to the Floating-Point Status and Control Register or vice versa.

The computational instructions and the *Select* instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

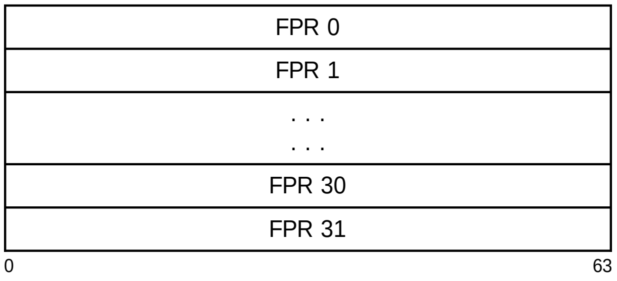

Figure 28. Floating-Point Registers

4.2.2 Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) controls the handling of floating-point exceptions and records status resulting from the floating-point operations. Bits 0:23 are status bits. Bits 24:31 are control bits.

The exception bits in the FPSCR (bits 3:12, 21:23) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an ***mcrfs, mtfsfi, mtfsf,*** or ***mtfsb0*** instruction. The exception summary bits in the FPSCR (FX, FEX, and VX, which are bits 0:2) are not considered to be “exception bits”, and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits affected by the various instructions.

Figure 29. Floating-Point Status and Control Register

The bit definitions for the FPSCR are as follows.

Bit(s) Description

0 ***Floating-Point Exception Summary*** (FX)
Every floating-point instruction, except ***mtfsfi*** and ***mtfsf***, implicitly sets FPSCR_{FX} to 1 if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0,*** and ***mtfsb1*** can alter FPSCR_{FX} explicitly.

1 ***Floating-Point Enabled Exception Summary*** (FEX)
This bit is the OR of all the floating-point exception bits masked by their respective enable bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter $\text{FPSCR}_{\text{FEX}}$ explicitly.

2 ***Floating-Point Invalid Operation Exception Summary*** (VX)
This bit is the OR of all the Invalid Operation exception bits. ***mcrfs***, ***mtfsfi***, ***mtfsf***, ***mtfsb0***, and ***mtfsb1*** cannot alter FPSCR_{VX} explicitly.

3 ***Floating-Point Overflow Exception*** (OX)
See Section 4.4.3, “Overflow Exception” on page 111.

4 ***Floating-Point Underflow Exception*** (UX)
See Section 4.4.4, “Underflow Exception” on page 112.

5 ***Floating-Point Zero Divide Exception*** (ZX)
See Section 4.4.2, "Zero Divide Exception" on page 111.

6 ***Floating-Point Inexact Exception*** (XX)
See Section 4.4.5, "Inexact Exception" on page 112.

FPSCR_{XX} is a sticky version of FPSCR_{FI} (see below). Thus the following rules completely describe how FPSCR_{XX} is set by a given instruction.

- If the instruction affects FPSCR_{FI}, the new value of FPSCR_{XX} is obtained by ORing the old value of FPSCR_{XX} with the new value of FPSCR_{FI}.
- If the instruction does not affect FPSCR_{FI}, the value of FPSCR_{XX} is unchanged.

7 ***Floating-Point Invalid Operation Exception (SNaN)*** (VXSNAN)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

8 ***Floating-Point Invalid Operation Exception*** ($\infty - \infty$ ***)*** (VXISI)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

9 ***Floating-Point Invalid Operation Exception*** ($\infty \div \infty$ ***)*** (VXIDI)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

10 ***Floating-Point Invalid Operation Exception*** ***(0***$\div$ ***0)*** (VXZDZ)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

11 ***Floating-Point Invalid Operation Exception*** ($\infty \times$ ***0)*** (VXIMZ)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

12 ***Floating-Point Invalid Operation Exception (Invalid Compare)*** (VXVC)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

13 ***Floating-Point Fraction Rounded*** (FR)
The last *Arithmetic* or *Rounding and Conversion* instruction incremented the fraction during rounding. See Section 4.3.6, "Rounding" on page 107. This bit is not sticky.

14 ***Floating-Point Fraction Inexact*** (FI)
The last *Arithmetic* or *Rounding and Conversion* instruction either produced an inexact result during rounding or caused a disabled Overflow Exception. See Section 4.3.6, "Rounding" on page 107. This bit is not sticky.

See the definition of FPSCR_{XX}, above, regarding the relationship between FPSCR_{FI} and FPSCR_{XX}.

15:19 ***Floating-Point Result Flags*** (FPRF)
This field is set as described below. For arithmetic, rounding, and conversion instructions, the field is set based on the result placed into the target register, except that if any portion of the result is undefined then the value placed into FPRF is undefined.

15 ***Floating-Point Result Class Descriptor*** (C)
Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 30 on page 103.

16:19 ***Floating-Point Condition Code*** (FPCC)
Floating-point *Compare* instructions set one of the FPCC bits to 1 and the other three FPCC bits to 0. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit, to indicate the class of the result as shown in Figure 30 on page 103. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.

16 ***Floating-Point Less Than or Negative*** (FL or <)

17 ***Floating-Point Greater Than or Positive*** (FG or >)

18 ***Floating-Point Equal or Zero*** (FE or =)

19 ***Floating-Point Unordered or NaN*** (FU or ?)

20 Reserved

21 ***Floating-Point Invalid Operation Exception (Software Request)*** (VXSOFT)
This bit can be altered only by ***mcrfs, mtfsfi, mtfsf, mtfsb0,*** or ***mtfsb1***. See Section 4.4.1, "Invalid Operation Exception" on page 110.

22 ***Floating-Point Invalid Operation Exception (Invalid Square Root)*** (VXSQRT)
See Section 4.4.1, "Invalid Operation Exception" on page 110.

Architecture Note

This bit is defined even for implementations that do not support either of the two optional instructions that set it, namely *Floating Square Root* and *Floating Reciprocal Square Root Estimate*. Defining it for all implementations gives software a standard interface for handling square root exceptions.

Programming Note

If the implementation does not support the optional *Floating Square Root* or *Floating Reciprocal Square Root Estimate* instruction, software can simulate the instruction and set this bit to reflect the exception.

23 ***Floating-Point Invalid Operation Exception (Invalid Integer Convert)*** (VXCVI)
See Section 4.4.1, “Invalid Operation Exception” on page 110.

24 ***Floating-Point Invalid Operation Exception Enable*** (VE)
See Section 4.4.1, “Invalid Operation Exception” on page 110.

25 ***Floating-Point Overflow Exception Enable*** (OE)
See Section 4.4.3, “Overflow Exception” on page 111.

26 ***Floating-Point Underflow Exception Enable*** (UE)
See Section 4.4.4, “Underflow Exception” on page 112.

27 ***Floating-Point Zero Divide Exception Enable*** (ZE)
See Section 4.4.2, “Zero Divide Exception” on page 111.

28 ***Floating-Point Inexact Exception Enable*** (XE)
See Section 4.4.5, “Inexact Exception” on page 112.

29 Reserved

Architecture Note

This bit will be among the last to be assigned a meaning. It was the NI (Non-IEEE Mode) bit in earlier versions of the architecture.

30:31 ***Floating-Point Rounding Control*** (RN)
See Section 4.3.6, “Rounding” on page 107.

00	Round to Nearest
01	Round toward Zero
10	Round toward + Infinity
11	Round toward – Infinity

Result Flags	**Result Value Class**
C < > = ?	
1 0 0 0 1	Quiet NaN
0 1 0 0 1	– Infinity
0 1 0 0 0	– Normalized Number
1 1 0 0 0	– Denormalized Number
1 0 0 1 0	– Zero
0 0 0 1 0	+ Zero
1 0 1 0 0	+ Denormalized Number
0 0 1 0 0	+ Normalized Number
0 0 1 0 1	+ Infinity

Figure 30. Floating-Point Result Flags

4.3 Floating-Point Data

4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format format may be used for data in storage and for data in floating-point registers.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats is shown below.

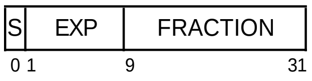

Figure 31. Floating-point single format

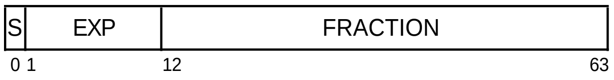

Figure 32. Floating-point double format

Values in floating-point format are composed of three fields:

S	sign bit
EXP	exponent+bias
FRACTION	fraction

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (i.e., the first bit to the left of the binary point). Values representable within the two floating-point formats can be specified by the parameters listed in Figure 33 on page 104.

	Format	
	Single	**Double**
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	−126	−1022
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

Figure 33. IEEE floating-point fields

The architecture requires that the FPRs of the Floating-Point Processor support the floating-point double format only.

4.3.2 Value Representation

This architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numeric values representable are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below. The relative location on the real number line for each of the defined entities is shown in Figure 34.

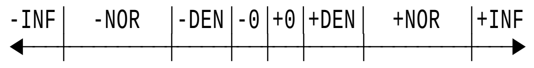

Figure 34. Approximation to real numbers

The NaNs are not related to the numeric values or infinities by order or value but are encodings used to convey diagnostic information such as the representation of uninitialized variables.

The following is a description of the different floating-point values defined in the architecture:

Binary floating-point numbers
Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

Normalized numbers (± NOR)
These are values that have a biased exponent value in the range:

1 to 254 in single format
1 to 2046 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$\text{NOR} = (-1)^s \times 2^E \times (1.\text{fraction})$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

Single Format:

$$1.2\text{x}10^{-38} \leq M \leq 3.4\text{x}10^{38}$$

Double Format:

$$2.2\text{x}10^{-308} \leq M \leq 1.8\text{x}10^{308}$$

Zero values (± 0)
These are values that have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (i.e., comparison regards +0 as equal to −0).

Denormalized numbers (± DEN)
These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as follows:

$$\text{DEN} = (-1)^s \times 2^{Emin} \times (0.\text{fraction})$$

where Emin is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

Infinities (± ∞)
These are values that have the maximum biased exponent value:

255 in single format
2047 in double format

and a zero fraction value. They are used to approximate values greater in magnitude than the maximum normalized value.

Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense:

$$-\infty < \text{every finite number} < +\infty$$

Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.4.1, “Invalid Operation Exception” on page 110.

Not a Numbers (NaNs)
These are values that have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (i.e., NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a *Signaling NaN*; otherwise it is a *Quiet NaN*.

Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions.

Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when Invalid Operation Exception is disabled ($FPSCR_{VE}=0$). Quiet NaNs propagate through all floating-point operations except ordered comparison, *Floating Round to Single-Precision*, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN is the result of a floating-point operation because one of the operands is a NaN or because a QNaN was generated due to a disabled Invalid Operation Exception, then the following rule is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```
if (FRA) is a NaN
    then FRT ← (FRA)
    else if (FRB) is a NaN
        then if instruction is frsp
            then FRT ← $(FRB)_{0:34}$ || $^{29}0$
            else FRT ← (FRB)
        else if (FRC) is a NaN
            then FRT ← (FRC)
            else if generated QNaN
                then FRT ← generated QNaN
```

If the operand specified by FRA is a NaN, then that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), then that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is ***frsp***. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), then that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled Invalid Operation Exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled Invalid Operation must generate this QNaN (i.e., 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even when the operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation $x-y$ is the same as the sign of the result of the add operation $x+(-y)$.

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward $-$Infinity, in which mode the sign is negative.
- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a *Square Root* or *Reciprocal Square Root Estimate* operation is always

positive, except that the square root of – 0 is – 0 and the reciprocal square root of – 0 is – Infinity.

- The sign of the result of a *Round to Single-Precision* or *Convert To/From Integer* operation is the sign of the operand being converted.

For the *Multiply-Add* instructions, the rules given above are applied first to the multiply operation and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

4.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or ***frsp*** instruction may require normalization and/or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or ***frsp*** instruction produces an intermediate result, consisting of a sign bit, an exponent, and a nonzero significand with a 0 leading bit, it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by 1 for each bit shifted, until the leading significand bit becomes 1. The Guard bit and the Round bit (see Section 4.5.1, “Execution Model for IEEE Operations” on page 113) participate in the shift with zeros shifted into the Round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 4.4.4, “Underflow Exception” on page 112. These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format's minimum value. If any significant bits are lost in this shifting process then “Loss of Accuracy” has occurred (See Section 4.4.4, “Underflow Exception” on page 112) and Underflow Exception is signaled.

Engineering Note

When denormalized numbers are operands of multiply, divide, and square root operations, some implementations may prenormalize the operands internally before performing the operations.

4.3.5 Data Handling and Precision

Instructions are defined to move floating-point data between the FPRs and storage. For double format data, the data are not altered during the move. For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions.

All computational, *Move*, and *Select* instructions use the floating-point double format.

Floating-point single-precision is obtained with the implementation of four types of instruction.

1. Load Floating-Point Single

   This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.

2. Round to Floating-Point Single-Precision

   The *Floating Round to Single-Precision* instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the *Floating Round to Single-Precision* instruction, this operation does not alter the value.

3. Single-Precision Arithmetic Instructions

   This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the Condition Register, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format.

   All input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the Condition Register (if Rc=1), are undefined.

4. Store Floating-Point Single

   This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions.

(The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

When the result of a *Load Floating-Point Single*, *Floating Round to Single-Precision*, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Programming Note

The *Floating Round to Single-Precision* instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by ***fcfid***) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a *Floating Round to Single-Precision* instruction.

Programming Note

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.

Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

4.3.6 Rounding

The material in this section applies to operations that have numeric operands (i.e., operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an Overflow Exception, an Underflow Exception, or an Inexact Exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 4.3.2, "Value Representation" on page 104 and Section 4.4, "Floating-Point Exceptions" on page 108 for the cases not covered here.

The arithmetic, rounding, and conversion instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. This intermediate result is normalized or denormalized if required, then rounded to the destination format. The final result is then placed into the target FPR in double format or in fixed-point integer format, depending on the instruction.

The instructions that round their intermediate result are the *Arithmetic* and *Rounding and Conversion* instructions. Each of these instructions sets FPSCR bits FR and FI. If the fraction was incremented during rounding then FR is set to 1, otherwise FR is set to 0. If the rounded result is inexact then FI is set to 1, otherwise FI is set to 0.

The two *Estimate* instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

Four user-selectable rounding modes are provided through the Floating-Point Rounding Control field in the FPSCR. See Section 4.2.2, "Floating-Point Status and Control Register" on page 101. These are encoded as follows:

RN	Rounding Mode
00	Round to Nearest
01	Round toward Zero
10	Round toward + Infinity
11	Round toward – Infinity

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Figure 35 shows the relation of Z, Z1, and Z2 in this case. The following rules specify the rounding in the four modes. "LSB" means "least significant bit".

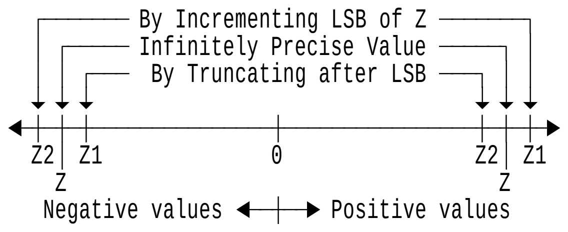

Figure 35. Selection of Z1 and Z2

Round to Nearest
Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit 0).

Round toward Zero
Choose the smaller in magnitude (Z1 or Z2).

Round toward + Infinity
Choose Z1.

Round toward – Infinity
Choose Z2.

See Section 4.5.1, “Execution Model for IEEE Operations” on page 113 for a detailed explanation of rounding.

4.4 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid Operation Exception
  - SNaN
  - Infinity – Infinity
  - Infinity ÷ Infinity
  - Zero ÷ Zero
  - Infinity × Zero
  - Invalid Compare
  - Software Request
  - Invalid Square Root
  - Invalid Integer Convert
- Zero Divide Exception
- Overflow Exception
- Underflow Exception
- Inexact Exception

These exceptions may occur during execution of computational instructions. In addition, an Invalid Operation Exception occurs when a *Move To FPSCR* instruction sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Each floating-point exception, and each category of Invalid Operation Exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see page 109), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an Underflow Exception may depend on the setting of the enable bit.)

A single instruction, other than ***mtfsfi*** or ***mtfsf***, may set more than one exception bit only in the following cases:

- Inexact Exception may be set with Overflow Exception.
- Inexact Exception may be set with Underflow Exception.
- Invalid Operation Exception (SNaN) is set with Invalid Operation Exception ($\infty \times 0$) for *Multiply-Add* instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Compare) for *Compare Ordered* instructions.
- Invalid Operation Exception (SNaN) may be set with Invalid Operation Exception (Invalid Integer Convert) for *Convert To Integer* instructions.

When an exception occurs the instruction execution may be suppressed or a result may be delivered, depending on the exception.

Instruction execution is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled Invalid Operation
- Enabled Zero Divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled Invalid Operation
- Disabled Zero Divide
- Disabled Overflow
- Disabled Underflow
- Disabled Inexact
- Enabled Overflow
- Enabled Underflow
- Enabled Inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

The IEEE standard specifies the handling of exceptional conditions in terms of “traps” and “trap handlers”. In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case: the expectation is that the exception will be detected by software, which will revise the result. An FPSCR exception enable bit of 0 causes generation of the “default result” value specified for the “trap disabled” (or “no trap occurs” or “trap is not implemented”) case: the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the sections below.

The IEEE default behavior when an exception occurs is to generate a default value and not to notify software. In this architecture, if the IEEE default behavior when an exception occurs is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and Ignore Exceptions Mode (see below) should be used. In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur: software can inspect the FPSCR exception bits if necessary, to determine whether exceptions have occurred.

In this architecture, if software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than Ignore Exceptions Mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a *Move To FPSCR* instruction causes an exception bit and the corresponding enable bit both to be 1; the *Move To FPSCR* instruction is considered to cause the enabled exception.

The FE0 and FE1 bits control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The location of these bits and the requirements for altering them are described in Book III, *PowerPC AS Operating Environment Architecture*. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) The effects of the four possible settings of these bits are as follows.

FE0	FE1	Description
0	0	**Ignore Exceptions Mode** Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	**Imprecise Nonrecoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked.
1	0	**Imprecise Recoverable Mode** The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked.
1	1	**Precise Mode** The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

Architecture Note

The FE0 and FE1 bits must be defined in Book III in a manner such that they can be changed dynamically and can easily be treated as part of a process' state.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. (Recall that, for the two Imprecise modes, the instruction at which the system floating-point enabled exception error handler is invoked need not be the instruction that caused the exception.) The instruction at which the system floating-point enabled exception error handler is invoked has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not among those listed on page 108 as suppressed.

Programming Note

In any of the three non-Precise modes, a *Floating-Point Status and Control Register* instruction can be used to force any exceptions, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to be recorded in the FPSCR. (This forcing is superfluous for Precise Mode.)

In either of the Imprecise modes, a *Floating-Point Status and Control Register* instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to instructions initiated before the *Floating-Point Status and Control Register* instruction, to occur. (This forcing has no effect in Ignore Exceptions Mode, and is superfluous for Precise Mode.)

In order to obtain the best performance across the widest range of implementations, the programmer should obey the following guidelines.

- If the IEEE default results are acceptable to the application, Ignore Exceptions Mode should be used with all FPSCR exception enable bits set to 0.
- If the IEEE default results are not acceptable to the application, Imprecise Nonrecoverable Mode should be used, or Imprecise Recoverable Mode if recoverability is needed, with FPSCR exception enable bits set to 1 for those exceptions for which the system floating-point enabled exception error handler is to be invoked.
- Ignore Exceptions Mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise Mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

Engineering Note

It is permissible for the implementation to be precise in any of the three modes that permit interrupts, or to be recoverable in Nonrecoverable Mode.

4.4.1 Invalid Operation Exception

4.4.1.1 Definition

An Invalid Operation Exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- Any floating-point operation on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (Invalid Compare)
- Square root or reciprocal square root of a negative (and nonzero) number (Invalid Square Root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (Invalid Integer Convert)

In addition, an Invalid Operation Exception occurs if software explicitly requests this by executing an ***mtfsfi***, ***mtfsf***, or ***mtfsb1*** instruction that sets $FPSCR_{VXSOFT}$ to 1 (Software Request).

Programming Note

The purpose of $FPSCR_{VXSOFT}$ is to allow software to cause an Invalid Operation Exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root, if the source operand is negative.

4.4.1.2 Action

The action to be taken depends on the setting of the Invalid Operation Exception Enable bit of the FPSCR.

When Invalid Operation Exception is enabled ($FPSCR_{VE}=1$) and Invalid Operation occurs or software explicitly requests the exception, the following actions are taken:

1. One or two Invalid Operation Exceptions are set

$FPSCR_{VXSNAN}$	(if SNaN)
$FPSCR_{VXISI}$	(if $\infty - \infty$)
$FPSCR_{VXIDI}$	(if $\infty \div \infty$)
$FPSCR_{VXZDZ}$	(if $0 \div 0$)
$FPSCR_{VXIMZ}$	(if $\infty \times 0$)
$FPSCR_{VXVC}$	(if invalid comp)
$FPSCR_{VXSOFT}$	(if software req)
$FPSCR_{VXSQRT}$	(if invalid sqrt)
$FPSCR_{VXCVI}$	(if invalid int cvrt)

2. If the operation is an arithmetic, *Floating Round to Single-Precision*, or convert to integer operation,
   the target FPR is unchanged
   $FPSCR_{FR\ FI}$ are set to zero
   $FPSCR_{FPRF}$ is unchanged
3. If the operation is a compare,
   $FPSCR_{FR\ FI\ C}$ are unchanged
   $FPSCR_{FPCC}$ is set to reflect unordered
4. If software explicitly requests the exception,
   $FPSCR_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi, mtfsf,*** or ***mtfsb1*** instruction

When Invalid Operation Exception is disabled ($FPSCR_{VE}=0$) and Invalid Operation occurs or software explicitly requests the exception, the following actions are taken:

1. One or two Invalid Operation Exceptions are set

$FPSCR_{VXSNAN}$	(if SNaN)
$FPSCR_{VXISI}$	(if $\infty - \infty$)
$FPSCR_{VXIDI}$	(if $\infty \div \infty$)
$FPSCR_{VXZDZ}$	(if $0 \div 0$)
$FPSCR_{VXIMZ}$	(if $\infty \times 0$)
$FPSCR_{VXVC}$	(if invalid comp)
$FPSCR_{VXSOFT}$	(if software req)
$FPSCR_{VXSQRT}$	(if invalid sqrt)
$FPSCR_{VXCVI}$	(if invalid int cvrt)

2. If the operation is an arithmetic or *Floating Round to Single-Precision* operation,
   the target FPR is set to a Quiet NaN
   $FPSCR_{FR\ FI}$ are set to zero

FPSCR$_{FPRF}$ is set to indicate the class of the result (Quiet NaN)

3. If the operation is a convert to 64-bit integer operation,
   the target FPR is set as follows:
   FRT is set to the most positive 64-bit integer if the operand in FRB is a positive number or $+\infty$, and to the most negative 64-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
   FPSCR$_{FR\ FI}$ are set to zero
   FPSCR$_{FPRF}$ is undefined
4. If the operation is a convert to 32-bit integer operation,
   the target FPR is set as follows:
   FRT$_{0:31}$ ← undefined
   FRT$_{32:63}$ are set to the most positive 32-bit integer if the operand in FRB is a positive number or $+\infty$, and to the most negative 32-bit integer if the operand in FRB is a negative number, $-\infty$, or NaN
   FPSCR$_{FR\ FI}$ are set to zero
   FPSCR$_{FPRF}$ is undefined
5. If the operation is a compare,
   FPSCR$_{FR\ FI\ C}$ are unchanged
   FPSCR$_{FPCC}$ is set to reflect unordered
6. If software explicitly requests the exception,
   FPSCR$_{FR\ FI\ FPRF}$ are as set by the ***mtfsfi, mtfsf,*** or ***mtfsb1*** instruction

4.4.2 Zero Divide Exception

4.4.2.1 Definition

A Zero Divide Exception occurs when a *Divide* instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a *Reciprocal Estimate* instruction (***fres*** or ***frsqrte***) is executed with an operand value of zero.

Architecture Note

The name is a misnomer used for historical reasons. The proper name for this exception should be “Exact Infinite Result from Finite Operands” corresponding to what mathematicians call a “pole”.

4.4.2.2 Action

The action to be taken depends on the setting of the Zero Divide Exception Enable bit of the FPSCR.

When Zero Divide Exception is enabled (FPSCR$_{ZE}$=1) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   FPSCR$_{ZX}$ ← 1
2. The target FPR is unchanged
3. FPSCR$_{FR\ FI}$ are set to zero
4. FPSCR$_{FPRF}$ is unchanged

When Zero Divide Exception is disabled (FPSCR$_{ZE}$=0) and Zero Divide occurs, the following actions are taken:

1. Zero Divide Exception is set
   FPSCR$_{ZX}$ ← 1
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands
3. FPSCR$_{FR\ FI}$ are set to zero
4. FPSCR$_{FPRF}$ is set to indicate the class and sign of the result (± Infinity)

4.4.3 Overflow Exception

4.4.3.1 Definition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

4.4.3.2 Action

The action to be taken depends on the setting of the Overflow Exception Enable bit of the FPSCR.

When Overflow Exception is enabled (FPSCR$_{OE}$=1) and exponent overflow occurs, the following actions are taken:

1. Overflow Exception is set
   FPSCR$_{OX}$ ← 1
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192
4. The adjusted rounded result is placed into the target FPR
5. FPSCR$_{FPRF}$ is set to indicate the class and sign of the result (± Normal Number)

When Overflow Exception is disabled (FPSCR$_{OE}$=0) and overflow occurs, the following actions are taken:

1. Overflow Exception is set
   FPSCR$_{OX}$ ← 1
2. Inexact Exception is set
   FPSCR$_{XX}$ ← 1
3. The result is determined by the rounding mode (FPSCR$_{RN}$) and the sign of the intermediate result as follows:
   A. Round to Nearest
      Store ± Infinity, where the sign is the sign of the intermediate result
   B. Round toward Zero
      Store the format's largest finite number with the sign of the intermediate result

   C. Round toward + Infinity
      For negative overflow, store the format's most negative finite number; for positive overflow, store + Infinity
   D. Round toward – Infinity
      For negative overflow, store – Infinity; for positive overflow, store the format's largest finite number
4. The result is placed into the target FPR
5. $FPSCR_{FR}$ is undefined
6. $FPSCR_{FI}$ is set to 1
7. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Infinity or ± Normal Number)

4.4.4 Underflow Exception

4.4.4.1 Definition

Underflow Exception is defined separately for the enabled and disabled states:

- Enabled:
  Underflow occurs when the intermediate result is “Tiny”.
- Disabled:
  Underflow occurs when the intermediate result is “Tiny” and there is “Loss of Accuracy”.

A “Tiny” result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is “Tiny” and Underflow Exception is disabled ($FPSCR_{UE}$=0) then the intermediate result is denormalized (see Section 4.3.4, “Normalization and Denormalization” on page 106) and rounded (see Section 4.3.6, “Rounding” on page 107) before being placed into the target FPR.

“Loss of Accuracy” is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

4.4.4.2 Action

The action to be taken depends on the setting of the Underflow Exception Enable bit of the FPSCR.

When Underflow Exception is enabled ($FPSCR_{UE}$=1) and exponent underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536
3. For single-precision arithmetic instructions and the *Floating Round to Single-Precision* instruction, the exponent of the normalized intermediate result is adjusted by adding 192
4. The adjusted rounded result is placed into the target FPR
5. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number)

Programming Note

The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an Underflow Exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When Underflow Exception is disabled ($FPSCR_{UE}$=0) and underflow occurs, the following actions are taken:

1. Underflow Exception is set
   $FPSCR_{UX} \leftarrow 1$
2. The rounded result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero)

4.4.5 Inexact Exception

4.4.5.1 Definition

An Inexact Exception occurs when one of two conditions occur during rounding:

1. The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled Overflow Exception or an enabled Underflow Exception, an Inexact Exception also occurs only if the significands of the rounded result and the intermediate result differ.)
2. The rounded result overflows and Overflow Exception is disabled.

4.4.5.2 Action

The action to be taken does not depend on the setting of the Inexact Exception Enable bit of the FPSCR.

When Inexact Exception occurs, the following actions are taken:

1. Inexact Exception is set
   $FPSCR_{XX} \leftarrow 1$
2. The rounded or overflowed result is placed into the target FPR
3. $FPSCR_{FPRF}$ is set to indicate the class and sign of the result

Programming Note

In some implementations, enabling Inexact Exceptions may degrade performance more than does enabling other types of floating-point exception.

4.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (i.e., operands and result that are not infinities or NaNs), and that cause no exceptions. See Section 4.3.2, “Value Representation” on page 104 and Section 4.4, “Floating-Point Exceptions” on page 108 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. The PowerPC AS Architecture follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, while single-precision arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions and ***fcfid*** produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

4.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION is a 23-bit field, and the single-precision Guard, Round, and Sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field.

IEEE-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

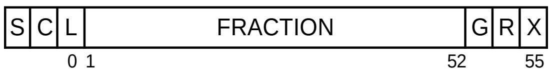

Figure 36. IEEE 64-bit execution model

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for postnormalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. Figure 37 on page 114 shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

G R X	Interpretation
0 0 0	IR is exact

0 0 1
0 1 0
0 1 1
	IR closer to NL

1 0 0	IR midway between NL and NH

1 0 1
1 1 0
1 1 1
	IR closer to NH

Figure 37. Interpretation of G, R, and X bits

After normalization, the intermediate result is rounded, using the rounding mode specified by FPSCR_{RN}. If rounding results in a carry into C, the significand is shifted right one position and the exponent incremented by one. This yields an inexact result and possibly also exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through FPSCR_{RN} as described in Section 4.3.6, "Rounding" on page 107. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 38 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the IEEE execution model.

Format	Guard	Round	Sticky
Double Single	G bit 24	R bit 25	X bit OR of 26:52, G, R, X

Figure 38. Location of the Guard, Round, and Sticky bits in the IEEE execution model

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the Guard, Round, or Sticky bits is nonzero, then the result is inexact.

Z1 and Z2, as defined on page 108, can be used to approximate the result in the target format when one of the following rules is used.

- ***Round to Nearest***

  Guard bit = 0
  The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011))

  Guard bit = 1
  Depends on Round and Sticky bits:

  Case a
  If the Round or Sticky bit is 1 (inclusive), the result is incremented. (Result closest to next higher value in magnitude (GRX = 101, 110, or 111))

  Case b
  If the Round and Sticky bits are 0 (result midway between closest representable values), then if the low-order bit of the result is 1 the result is incremented. Otherwise (the low-order bit of the result is 0) the result is truncated (this is the case of a tie rounded to even).

- ***Round toward Zero***
  Choose the smaller in magnitude of Z1 or Z2. If the Guard, Round, or Sticky bit is nonzero, the result is inexact.

- ***Round toward* + *Infinity***
  Choose Z1.

- ***Round toward* – *Infinity***
  Choose Z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a *Floating Round to Single-Precision* or single-precision arithmetic instruction, the intermediate result is either normalized or placed in correct denormalized form before being rounded.

4.5.2 Execution Model for Multiply-Add Type Instructions

The PowerPC AS Architecture provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

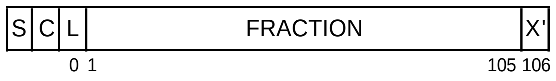

Figure 39. Multiply-add 64-bit execution model

The first part of the operation is a multiplication. The multiplication has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, the FRACTION) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount that is added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the addition is then normalized, with all bits of the addition result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. Figure 40 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Format	**Guard**	**Round**	**Sticky**
Double	53	54	OR of 55:105, X'
Single	24	25	OR of 26:105, X'

Figure 40. Location of the Guard, Round, and Sticky bits in the multiply-add execution model

The rules for rounding the intermediate result are the same as those given in Section 4.5.1, “Execution Model for IEEE Operations” on page 113.

If the instruction is *Floating Negative Multiply-Add* or *Floating Negative Multiply-Subtract*, the final result is negated.

4.6 Floating-Point Processor Instructions

Architecture Note

The rules followed in assigning new primary and extended opcodes, for instructions that are not in the POWER Architecture, are the following.

1. A new primary opcode, 59, has been added. It is used for the single-precision arithmetic instructions.
2. The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as that double-precision instruction.
3. In assigning new extended opcodes for primary opcode 63, the following regularities, present in the POWER Architecture, have been maintained. In addition, all new X-form instructions in primary opcode 63 have bits 21:22 = 0b11, which distinguishes them from the X-form instructions present in POWER Architecture.
   - Bit 26 = 1 iff the instruction is A-form.
   - Bits 26:29 = 0b0000 iff the instruction is a comparison or ***mcrfs*** (i.e., iff the instruction sets an explicitly-designated CR field).
   - Bits 26:28 = 0b001 iff the instruction explicitly refers to or sets the FPSCR (i.e., is a *Floating-Point Status and Control Register* instruction) and is not ***mcrfs***.
   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the FPSCR.
4. In assigning extended opcodes for primary opcode 59, the following regularities have been maintained. They are based on those rules for primary opcode 63 that apply to the instructions having primary opcode 59. In particular, primary opcode 59 has no *Floating-Point Status and Control Register* instructions, so the corresponding rule does not apply.
   - If there is a corresponding instruction with primary opcode 63, its extended opcode is used.
   - Bit 26 = 1 iff the instruction is A-form.
   - Bits 26:30 = 0b01000 iff the instruction is a *Move Register* instruction, or any other instruction that does not refer to or set the FPSCR.

4.6.1 Floating-Point Storage Access Instructions

The *Storage Access* instructions compute the effective address (EA) of the storage to be accessed as described in Section 1.12.3, “Effective Address Calculation” on page 17.

†

Programming Note

The ***la*** extended mnemonic permits computing an effective address as a *Load* or *Store* instruction would, but loads the address itself into a GPR rather than loading the value that is in storage at
† that address. Unlike a *Load* or *Store* instruction,
† ***la*** cannot cause an Effective Address Overflow
† exception. This extended mnemonic is described
† in Section B.11, “Miscellaneous Mnemonics” on
† page 174.

†

Programming Note

† See the Programming Note on page 6 regarding
† base register usage for X-form *Load* and *Store*
† instructions in *tags active* mode.

4.6.1.1 Storage Access Exceptions

† Storage accesses will cause the system data storage error handler to be invoked if the program is not allowed to modify the target storage (*Store* only), or if the program attempts to access storage that is unavailable.

4.6.2 Floating-Point Load Instructions

There are two basic forms of load instruction: single-precision and double-precision. Because the FPRs support only floating-point double format, single-precision *Load Floating-Point* instructions convert single-precision data to double format prior to loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

```
if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
    $FRT_{0:1} \leftarrow WORD_{0:1}$
    $FRT_2 \leftarrow \neg WORD_1$
    $FRT_3 \leftarrow \neg WORD_1$
    $FRT_4 \leftarrow \neg WORD_1$
    $FRT_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$
```

Denormalized Operand

```
if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
    sign $\leftarrow WORD_0$
    exp $\leftarrow -126$
    $frac_{0:52} \leftarrow$ 0b0 $\| WORD_{9:31} \| {}^{29}0$
    normalize the operand
        do while $frac_0 = 0$
            frac $\leftarrow frac_{1:52} \|$ 0b0
            exp $\leftarrow$ exp $- 1$
    $FRT_0 \leftarrow$ sign
    $FRT_{1:11} \leftarrow$ exp $+ 1023$
    $FRT_{12:63} \leftarrow frac_{1:52}$
```

Zero / Infinity / NaN

```
if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
    $FRT_{0:1} \leftarrow WORD_{0:1}$
    $FRT_2 \leftarrow WORD_1$
    $FRT_3 \leftarrow WORD_1$
    $FRT_4 \leftarrow WORD_1$
    $FRT_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$
```

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Load Floating-Point* instructions no conversion is required, as the data from storage are copied directly into the FPR.

Many of the *Load Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if $RA \neq 0$, the effective address is placed into register RA and the storage element (word or doubleword) addressed by EA is loaded into FRT.

Note: Recall that RA and RB denote General Purpose Registers, while FRT denotes a Floating-Point Register.

†

Load Floating-Point Single D-form

lfs FRT,D(RA)

48	FRT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)$+_{tea}$D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 117) and placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Single Indexed X-form

lfsx FRT,RA,RB

31	FRT	RA	RB	535	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b $+_{tea}$ (RB)
FRT ← DOUBLE(MEM(EA, 4))
```

Let the effective address (EA) be the sum (RA|0)$+_{tea}$(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 117) and placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Single with Update D-form

lfsu FRT,D(RA)

49	FRT	RA	D
0	6	11	16 31

```
EA ← (RA) $+_{tea}$ EXTS(D)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)$+_{tea}$D.

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 117) and placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Single with Update Indexed X-form

lfsux FRT,RA,RB

31	FRT	RA	RB	567	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
FRT ← DOUBLE(MEM(EA, 4))
RA ← EA
```

Let the effective address (EA) be the sum (RA)$+_{tea}$(RB).

The word in storage addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double format (see page 117) and placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double D-form

lfd FRT,D(RA)

50	FRT	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ EXTS(D)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D.

The doubleword in storage addressed by EA is placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double Indexed X-form

lfdx FRT,RA,RB

31	FRT	RA	RB	599	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b $+_{tea}$ (RB)
FRT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

The doubleword in storage addressed by EA is placed into register FRT.

Special Registers Altered:
None

Load Floating-Point Double with Update D-form

lfdu FRT,D(RA)

51	FRT	RA	D
0	6	11	16 31

```
EA ← (RA) $+_{tea}$ EXTS(D)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D.

The doubleword in storage addressed by EA is placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Load Floating-Point Double with Update Indexed X-form

lfdux FRT,RA,RB

31	FRT	RA	RB	631	/
0	6	11	16	21	31

```
EA ← (RA) $+_{tea}$ (RB)
FRT ← MEM(EA, 8)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB).

The doubleword in storage addressed by EA is placed into register FRT.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

4.6.3 Floating-Point Store Instructions

There are three basic forms of store instruction: single-precision, double-precision, and integer. The integer form is provided by the optional *Store Floating-Point as Integer Word* instruction, described on page 123. Because the FPRs support only floating-point double format for floating-point data, single-precision *Store Floating-Point* instructions convert double-precision data to single format prior to storing the operand into storage. The conversion steps are as follows.

Let $\text{WORD}_{0:31}$ be the word in storage written to.

No Denormalization Required (includes Zero / Infinity / NaN)

```
if $\text{FRS}_{1:11} > 896$ or $\text{FRS}_{1:63} = 0$ then
    $\text{WORD}_{0:1} \leftarrow \text{FRS}_{0:1}$
    $\text{WORD}_{2:31} \leftarrow \text{FRS}_{5:34}$
```

Denormalization Required

```
if $874 \le \text{FRS}_{1:11} \le 896$ then
    sign $\leftarrow \text{FRS}_0$
    exp $\leftarrow \text{FRS}_{1:11} - 1023$
    frac $\leftarrow$ 0b1 || $\text{FRS}_{12:63}$
    denormalize operand
        do while exp $< -126$
            frac $\leftarrow$ 0b0 || $\text{frac}_{0:62}$
            exp $\leftarrow$ exp $+ 1$
    $\text{WORD}_0 \leftarrow$ sign
    $\text{WORD}_{1:8} \leftarrow$ 0x00
    $\text{WORD}_{9:31} \leftarrow \text{frac}_{1:23}$
else WORD $\leftarrow$ undefined
```

Notice that if the value to be stored by a single-precision *Store Floating-Point* instruction is larger in magnitude than the maximum number representable in single format, the first case above (No Denormalization Required) applies. The result stored in WORD is then a well-defined value, but is not numerically equal to the value in the source register (i.e., the result of a single-precision *Load Floating-Point* from WORD will not compare equal to the contents of the original source register).

Engineering Note

The above description of the conversion steps is a model only. The actual implementation may vary from this but must produce results equivalent to what this model would produce.

For double-precision *Store Floating-Point* instructions and for the *Store Floating-Point as Integer Word* instruction no conversion is required, as the data from the FPR are copied directly into storage.

For all *Store Floating-Point* instructions, the tag of every tag block affected is set to zero.

Many of the *Store Floating-Point* instructions have an “update” form, in which register RA is updated with the effective address. For these forms, if RA$\neq$0, the effective address is placed into register RA.

Note: Recall that RA and RB denote General Purpose Registers, while FRS denotes a Floating-Point Register.

†

Store Floating-Point Single D-form

stfs FRS,D(RA)

52	FRS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
MEMtag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D.

The contents of register FRS are converted to single format (see page 120) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single Indexed X-form

stfsx FRS,RA,RB

31	FRS	RA	RB	663	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 4) ← SINGLE((FRS))
MEMtag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

The contents of register FRS are converted to single format (see page 120) and stored into the word in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Single with Update D-form

stfsu FRS,D(RA)

53	FRS	RA	D
0	6	11	16 31

```
EA ← (RA) +tea EXTS(D)
MEM(EA, 4) ← SINGLE((FRS))
MEMtag(EA, 4) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D.

The contents of register FRS are converted to single format (see page 120) and stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Single with Update Indexed X-form

stfsux FRS,RA,RB

31	FRS	RA	RB	695	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
MEM(EA, 4) ← SINGLE((FRS))
MEMtag(EA, 4) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB).

The contents of register FRS are converted to single format (see page 120) and stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double D-form

stfd FRS,D(RA)

54	FRS	RA	D
0	6	11	16 31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea EXTS(D)
MEM(EA, 8) ← (FRS)
MEMtag(EA, 8) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$D.

The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double Indexed X-form

stfdx FRS,RA,RB

31	FRS	RA	RB	727	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 8) ← (FRS)
MEMtag(EA, 8) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

Special Registers Altered:
None

Store Floating-Point Double with Update D-form

stfdu FRS,D(RA)

55	FRS	RA	D
0	6	11	16 31

```
EA ← (RA) +tea EXTS(D)
MEM(EA, 8) ← (FRS)
MEMtag(EA, 8) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$D.

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point Double with Update Indexed X-form

stfdux FRS,RA,RB

31	FRS	RA	RB	759	/
0	6	11	16	21	31

```
EA ← (RA) +tea (RB)
MEM(EA, 8) ← (FRS)
MEMtag(EA, 8) ← 0
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ $_{tea}$(RB).

The contents of register FRS are stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Floating-Point as Integer Word Indexed X-form

stfiwx FRS,RA,RB

31	FRS	RA	RB	983	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
MEM(EA, 4) ← (FRS)32:63
MEMtag(EA, 4) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

The contents of the low-order 32 bits of register FRS are stored, without conversion, into the word in storage addressed by EA.

If the contents of register FRS were produced, either directly or indirectly, by a *Load Floating-Point Single* instruction, a single-precision *Arithmetic* instruction, or ***frsp***, then the value stored is undefined. (The contents of register FRS are produced directly by such an instruction if FRS is the target register for the instruction. The contents of register FRS are produced indirectly by such an instruction if FRS is the final target register of a sequence of one or more *Floating-Point Move* instructions, with the input to the sequence having been produced directly by such an instruction.)

Special Registers Altered:
None

Architecture Note

Allowing the value stored to be undefined if the input to ***stfiwx*** was produced by a single-precision-producing instruction (i.e., a *Load Floating-Point Single* instruction, a single-precision arithmetic instruction, or ***frsp***) seems gratuitous at the architectural level. The background and reasons for allowing it are as follows.

- The implementors agreed to support ***stfiwx*** partly because they understood it to be easy to implement.
- In some implementations (e.g., those that keep single-precision numbers in registers in a non-architected format), storing the architected low-order 32 bits of a register that was set by a single-precision-producing instruction may be harder (and slower, and more trouble to verify) than simply storing whatever happens to be in the low-order 32 bits of the register.
- Software can think of no use for storing the low-order 32 bits of the result of a single-precision producing instruction.

4.6.4 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described below for ***fneg, fabs***, and ***fnabs***. These instructions treat NaNs just like any other kind of value (e.g., the sign bit of a NaN may be altered by ***fneg, fabs***, and ***fnabs***). These instructions do not alter the FPSCR.

Floating Move Register X-form

fmr	FRT,FRB	(Rc=0)
fmr.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	72	Rc
0	6	11	16	21	31

The contents of register FRB are placed into register FRT.

Special Registers Altered:

CR1 (if Rc=1)

Floating Negate X-form

fneg	FRT,FRB	(Rc=0)
fneg.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	40	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 inverted are placed into register FRT.

Special Registers Altered:

CR1 (if Rc=1)

Floating Absolute Value X-form

fabs	FRT,FRB	(Rc=0)
fabs.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	264	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 set to zero are placed into register FRT.

Special Registers Altered:

CR1 (if Rc=1)

Floating Negative Absolute Value X-form

fnabs	FRT,FRB	(Rc=0)
fnabs.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	136	Rc
0	6	11	16	21	31

The contents of register FRB with bit 0 set to one are placed into register FRT.

Special Registers Altered:

CR1 (if Rc=1)

4.6.5 Floating-Point Arithmetic Instructions

4.6.5.1 Floating-Point Elementary Arithmetic Instructions

Floating Add [*Single*] *A-form*

fadd	FRT,FRA,FRB	(Rc=0)
fadd.	FRT,FRA,FRB	(Rc=1)

[POWER mnemonics: fa, fa.]

63	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

fadds	FRT,FRA,FRB	(Rc=0)
fadds.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	///	21	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is added to the floating-point operand in register FRB.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Subtract [*Single*] *A-form*

fsub	FRT,FRA,FRB	(Rc=0)
fsub.	FRT,FRA,FRB	(Rc=1)

[POWER mnemonics: fs, fs.]

63	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

fsubs	FRT,FRA,FRB	(Rc=0)
fsubs.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	///	20	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRB is subtracted from the floating-point operand in register FRA.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

The execution of the *Floating Subtract* instruction is identical to that of *Floating Add*, except that the contents of FRB participate in the operation with the sign bit (bit 0) inverted.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1 (if Rc=1)

Floating Multiply [*Single*] *A-form*

fmul	FRT,FRA,FRC	(Rc=0)
fmul.	FRT,FRA,FRC	(Rc=1)

[POWER mnemonics: fm, fm.]

63	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

fmuls	FRT,FRA,FRC	(Rc=0)
fmuls.	FRT,FRA,FRC	(Rc=1)

59	FRT	FRA	///	FRC	25	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXIMZ
CR1 (if Rc=1)

Floating Divide [*Single*] *A-form*

fdiv	FRT,FRA,FRB	(Rc=0)
fdiv.	FRT,FRA,FRB	(Rc=1)

[POWER mnemonics: fd, fd.]

63	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

fdivs	FRT,FRA,FRB	(Rc=0)
fdivs.	FRT,FRA,FRB	(Rc=1)

59	FRT	FRA	FRB	///	18	Rc
0	6	11	16	21	26	31

The floating-point operand in register FRA is divided by the floating-point operand in register FRB. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Floating-point division is based on exponent subtraction and division of the significands.

$FPSCR_{FPRF}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $FPSCR_{VE}=1$ and Zero Divide Exceptions when $FPSCR_{ZE}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX ZX XX
VXSNAN VXIDI VXZDZ
CR1 (if Rc=1)

4.6.5.2 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows.

- Overflow, Underflow, and Inexact Exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, and not on the result of the multiplication.
- Invalid Operation Exception bits are set as if the multiplication and the addition were performed using two separate instructions (***fmul***[***s***], followed by ***fadd***[***s***] or ***fsub***[***s***]). That is, multiplication of infinity by 0 or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

Floating Multiply-Add [*Single*] *A-form*

fmadd	FRT,FRA,FRC,FRB	(Rc=0)
fmadd.	FRT,FRA,FRC,FRB	(Rc=1)

[POWER mnemonics: fma, fma.]

63	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

fmadds	FRT,FRA,FRC,FRB	(Rc=0)
fmadds.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	29	Rc
0	6	11	16	21	26	31

The operation

$$\text{FRT} \leftarrow [(\text{FRA})\times(\text{FRC})] + (\text{FRB})$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Multiply-Subtract [*Single*] *A-form*

fmsub	FRT,FRA,FRC,FRB	(Rc=0)
fmsub.	FRT,FRA,FRC,FRB	(Rc=1)

[POWER mnemonics: fms, fms.]

63	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

fmsubs	FRT,FRA,FRC,FRB	(Rc=0)
fmsubs.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	28	Rc
0	6	11	16	21	26	31

The operation

$$\text{FRT} \leftarrow [(\text{FRA})\times(\text{FRC})] - (\text{FRB})$$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Add [*Single*] *A-form*

fnmadd	FRT,FRA,FRC,FRB	(Rc=0)
fnmadd.	FRT,FRA,FRC,FRB	(Rc=1)

[POWER mnemonics: fnma, fnma.]

63	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

fnmadds	FRT,FRA,FRC,FRB	(Rc=0)
fnmadds.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	31	Rc
0	6	11	16	21	26	31

The operation

$\text{FRT} \leftarrow -\ (\ [(\text{FRA})\times(\text{FRC})]\ +\ (\text{FRB})\ )$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Add* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

Floating Negative Multiply-Subtract [*Single*] *A-form*

fnmsub	FRT,FRA,FRC,FRB	(Rc=0)
fnmsub.	FRT,FRA,FRC,FRB	(Rc=1)

[POWER mnemonics: fnms, fnms.]

63	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

fnmsubs	FRT,FRA,FRC,FRB	(Rc=0)
fnmsubs.	FRT,FRA,FRC,FRB	(Rc=1)

59	FRT	FRA	FRB	FRC	30	Rc
0	6	11	16	21	26	31

The operation

$\text{FRT} \leftarrow -\ (\ [(\text{FRA})\times(\text{FRC})]\ -\ (\text{FRB})\ )$

is performed.

The floating-point operand in register FRA is multiplied by the floating-point operand in register FRC. The floating-point operand in register FRB is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR, then negated and placed into register FRT.

This instruction produces the same result as would be obtained by using the *Floating Multiply-Subtract* instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their “sign” bit.
- QNaNs that are generated as the result of a disabled Invalid Operation Exception have a “sign” bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled Invalid Operation Exception retain the “sign” bit of the SNaN.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1 (if Rc=1)

4.6.6 Floating-Point Rounding and Conversion Instructions

Programming Note

Examples of uses of these instructions to perform various conversions can be found in Section C.2, “Floating-Point Conversions” on page 180.

Floating Round to Single-Precision X-form

frsp	FRT,FRB	(Rc=0)
frsp.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	12	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is rounded to single-precision, using the rounding mode specified by $\mathrm{FPSCR}_{\mathrm{RN}}$, and placed into register FRT.

The rounding is described fully in Section A.1, “Floating-Point Round to Single-Precision Model” on page 151.

$\mathrm{FPSCR}_{\mathrm{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\mathrm{FPSCR}_{\mathrm{VE}}=1$.

Special Registers Altered:

FPRF FR FI
FX OX UX XX
VXSNAN
CR1 (if Rc=1)

Floating Convert To Integer Doubleword X-form

fctid	FRT,FRB	(Rc=0)
fctid.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	814	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

If the operand in FRB is greater than $2^{63} - 1$, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 156.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI	
FX XX	
VXSNAN VXCVI	
CR1	(if Rc=1)

Floating Convert To Integer Doubleword with round toward Zero X-form

fctidz	FRT,FRB	(Rc=0)
fctidz.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	815	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 64-bit signed fixed-point integer, using the rounding mode Round toward Zero, and placed into register FRT.

If the operand in FRB is greater than $2^{63} - 1$, then FRT is set to 0x7FFF_FFFF_FFFF_FFFF. If the operand in FRB is less than -2^{63}, then FRT is set to 0x8000_0000_0000_0000.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 156.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI	
FX XX	
VXSNAN VXCVI	
CR1	(if Rc=1)

Floating Convert To Integer Word X-form

fctiw	FRT,FRB	(Rc=0)
fctiw.	FRT,FRB	(Rc=1)

[POWER2 mnemonics: fcir, fcir.]

63	FRT	///	FRB	14	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$, and placed into
† $FRT_{32:63}$. The contents of $FRT_{0:31}$ are undefined.

If the operand in FRB is greater than $2^{31} - 1$, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 156.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI	
FX XX	
VXSNAN VXCVI	
CR1	(if Rc=1)

Floating Convert To Integer Word with round toward Zero X-form

fctiwz	FRT,FRB	(Rc=0)
fctiwz.	FRT,FRB	(Rc=1)

[POWER2 mnemonics: fcirz, fcirz.]

63	FRT	///	FRB	15	Rc
0	6	11	16	21	31

The floating-point operand in register FRB is converted to a 32-bit signed fixed-point integer, using the rounding mode Round toward Zero, and placed into
† $FRT_{32:63}$. The contents of $FRT_{0:31}$ are undefined.

If the operand in FRB is greater than $2^{31} - 1$, then bits 32:63 of FRT are set to 0x7FFF_FFFF. If the operand in FRB is less than -2^{31}, then bits 32:63 of FRT are set to 0x8000_0000.

The conversion is described fully in Section A.2, “Floating-Point Convert to Integer Model” on page 156.

Except for enabled Invalid Operation Exceptions, $FPSCR_{FPRF}$ is undefined. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF (undefined) FR FI	
FX XX	
VXSNAN VXCVI	
CR1	(if Rc=1)

Floating Convert From Integer Doubleword X-form

fcfid	FRT,FRB	(Rc=0)
fcfid.	FRT,FRB	(Rc=1)

63	FRT	///	FRB	846	Rc
0	6	11	16	21	31

The 64-bit signed fixed-point operand in register FRB is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, using the rounding mode specified by $FPSCR_{RN}$, and placed into register FRT.

The conversion is described fully in Section A.3, “Floating-Point Convert from Integer Model” on page 159.

$FPSCR_{FPRF}$ is set to the class and sign of the result. $FPSCR_{FR}$ is set if the result is incremented when rounded. $FPSCR_{FI}$ is set if the result is inexact.

Special Registers Altered:

FPRF FR FI	
FX XX	
CR1	(if Rc=1)

4.6.7 Floating-Point Compare Instructions

The floating-point *Compare* instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (i.e., regards +0 as equal to –0). The comparison can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three to 0. The FPCC is set in the same way.

The CR field and the FPCC are set as follows.

Bit	**Name**	**Description**
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Floating Compare Unordered X-form

fcmpu BF,FRA,FRB

63	BF	//	FRA	FRB	0	/
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                       c ← 0b0010
FPCC ← c
CR_{4×BF:4×BF+3} ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN

Floating Compare Ordered X-form

fcmpo BF,FRA,FRB

63	BF	//	FRA	FRB	32	/
0	6	9	11	16	21	31

```
if (FRA) is a NaN or
   (FRB) is a NaN then c ← 0b0001
else if (FRA) < (FRB) then c ← 0b1000
else if (FRA) > (FRB) then c ← 0b0100
else                       c ← 0b0010
FPCC ← c
CR_{4×BF:4×BF+3} ← c
if (FRA) is an SNaN or
   (FRB) is an SNaN then
     VXSNAN ← 1
     if VE = 0 then VXVC ← 1
else if (FRA) is a QNaN or
   (FRB) is a QNaN then VXVC ← 1
```

The floating-point operand in register FRA is compared to the floating-point operand in register FRB. The result of the compare is placed into CR field BF and the FPCC.

If either of the operands is a NaN, either quiet or signaling, then CR field BF and the FPCC are set to reflect unordered. If either of the operands is a Signaling NaN, then VXSNAN is set and, if Invalid Operation is disabled (VE=0), VXVC is set. If neither operand is a Signaling NaN but at least one operand is a Quiet NaN, then VXVC is set.

Special Registers Altered:

CR field BF
FPCC
FX
VXSNAN VXVC

4.6.8 Floating-Point Status and Control Register Instructions

Every *Floating-Point Status and Control Register* instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a *Floating-Point Status and Control Register* instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the *Floating-Point Status and Control Register* instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the *Floating-Point Status and Control Register* instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the *Floating-Point Status and Control Register* instruction is initiated.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the *Floating-Point Status and Control Register* instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the *Floating-Point Status and Control Register* instruction has completed.

(Floating-point *Storage Access* instructions are not affected.)

Move From FPSCR X-form

mffs	FRT	(Rc=0)
mffs.	FRT	(Rc=1)

63	FRT	///	///	583	Rc
0	6	11	16	21	31

† The contents of the FPSCR are placed into $FRT_{32:63}$.
† The contents of $FRT_{0:31}$ are undefined.

Special Registers Altered:

CR1	(if Rc=1)

Move to Condition Register from FPSCR X-form

mcrfs BF,BFA

63	BF	//	BFA	//	///	64	/
0	6	9	11	14	16	21	31

The contents of FPSCR field BFA are copied to Condition Register field BF. All exception bits copied are set to 0 in the FPSCR. If the FX bit is copied, it is set to 0 in the FPSCR.

Special Registers Altered:

CR field BF	
FX OX	(if BFA=0)
UX ZX XX VXSNAN	(if BFA=1)
VXISI VXIDI VXZDZ VXIMZ	(if BFA=2)
VXVC	(if BFA=3)
VXSOFT VXSQRT VXCVI	(if BFA=5)

Move To FPSCR Field Immediate X-form

mtfsfi	BF,U	(Rc=0)
mtfsfi.	BF,U	(Rc=1)

63	BF	//	///	U	/	134	Rc
0	6	9	11	16	20	21	31

The value of the U field is placed into FPSCR field BF.

$FPSCR_{FX}$ is altered only if BF = 0.

Special Registers Altered:
FPSCR field BF
CR1 (if Rc=1)

Programming Note

When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of U_0 and U_3 (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from U_0 and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on page 101, and not from $U_{1:2}$.

Move To FPSCR Fields XFL-form

mtfsf	FLM,FRB	(Rc=0)
mtfsf.	FLM,FRB	(Rc=1)

63	/	FLM	/	FRB	711	Rc
0	6	7	15	16	21	31

The contents of bits 32:63 of register FRB are placed into the FPSCR under control of the field mask specified by FLM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0-7. If $FLM_i = 1$ then FPSCR field i (FPSCR bits $4\times i$:$4\times i+3$) is set to the contents of the corresponding field of the low-order 32 bits of register FRB.

$FPSCR_{FX}$ is altered only if FLM_0 = 1.

Special Registers Altered:
FPSCR fields selected by mask
CR1 (if Rc=1)

Programming Note

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

Programming Note

When $FPSCR_{0:3}$ is specified, bits 0 (FX) and 3 (OX) are set to the values of $(FRB)_{32}$ and $(FRB)_{35}$ (i.e., even if this instruction causes OX to change from 0 to 1, FX is set from $(FRB)_{32}$ and not by the usual rule that FX is set to 1 when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given on page 101, and not from $(FRB)_{33:34}$.

Move To FPSCR Bit 0 X-form

mtfsb0	BT	(Rc=0)
mtfsb0.	BT	(Rc=1)

63	BT	///	///	70	Rc
0	6	11	16	21	31

Bit BT of the FPSCR is set to 0.

Special Registers Altered:

FPSCR bit BT
CR1 (if Rc=1)

Programming Note

Bits 1 and 2 (FEX and VX) cannot be explicitly reset.

Move To FPSCR Bit 1 X-form

mtfsb1	BT	(Rc=0)
mtfsb1.	BT	(Rc=1)

63	BT	///	///	38	Rc
0	6	11	16	21	31

Bit BT of the FPSCR is set to 1.

Special Registers Altered:

FPSCR bits BT and FX
CR1 (if Rc=1)

Programming Note

Bits 1 and 2 (FEX and VX) cannot be explicitly set.

† Chapter 5. Optional Facilities and Instructions

† The facilities and instructions described in this chapter are optional. An implementation may provide all, some, or none of them, except as described in
† Section 5.2.

†

5.1 Fixed-Point Processor Instructions

5.1.1 Move To/From System Register Instructions

The optional versions of the *Move To Condition Register Field* and *Move From Condition Register* instructions move to or from a single CR field.

Move To Condition Register Field XFX-form

mtcrf FXM,RS

31	RS	1	FXM	/	144	/
0	6	11	12	20	21	31

```
count ← 0
do i = 0 to 7
  if $FXM_i$ = 1 then
    n ← i
    count ← count + 1
if count = 1 then $CR_{4×n:4×n+3}$ ← $(RS)_{32+4×n:32+4×n+3}$
else CR ← undefined
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \leq n \leq 7$). The contents of bits 32+4×n:32+4×n+3 of register RS are placed into CR field n (CR bits 4×n:4×n+3). Otherwise, the contents of the Condition Register are undefined.

Special Registers Altered:
CR field selected by FXM

Programming Note

These forms of the ***mtcrf*** and ***mfcr*** instructions are intended to replace the old forms of the instructions (the forms shown in Section 3.3.15), which will eventually be phased out of the architecture. The new forms are backward compatible with most processors that comply with versions of the architecture that precede Version 2.00. On those processors, the new forms are treated as the old forms.

However, on some processors that comply with versions of the architecture that precede Version 2.00 the new forms may be treated as follows:

mtcrf: may cause the system illegal instruction error handler to be invoked
mfcr: may copy the contents of an SPR, possibly a privileged SPR, into register RT

Assembler Note

There is no direct way for the programmer to specify whether the Assembler should generate the old forms of these instructions or the new forms. The Assembler should determine which form to generate based on the target machine, as well as on how the instruction is coded (i.e., whether an FXM field is given for ***mfcr*** and, for both instructions, whether the FXM field has exactly one bit set to 1).

Move From Condition Register XFX-form

mfcr RT,FXM

31	RT	1	FXM	/	19	/
0	6	11	12	20	21	31

```
RT ← undefined
count ← 0
do i = 0 to 7
  if $FXM_i$ = 1 then
    n ← i
    count ← count + 1
if count = 1 then $RT_{32+4×n:32+4×n+3}$ ← $CR_{4×n:4×n+3}$
```

If exactly one bit of the FXM field is set to 1, let n be the position of that bit in the field ($0 \leq n \leq 7$). The contents of CR field n (CR bits 4×n:4×n+3) are placed into bits 32+4×n:32+4×n+3 of register RT and the contents of the remaining bits of register RT are undefined. Otherwise, the contents of register RT are undefined.

Special Registers Altered:
None

Engineering Note

These forms of the ***mtcrf*** and ***mfcr*** instructions are being phased into the architecture, and must be implemented in processors that comply with Version 2.00 of the architecture specification or with any subsequent version.

Architecture Note

The processors for which the new forms of these instructions are *not* treated as the old forms are as follows:

mtcrf: versions of the 630 processor that predate 630 SOI (Illegal Instruction type Program interrupt)
mfcr: Northstar processors (incorrect results)

When the performance of systems based on these processors is less important than the performance of newer systems, the new forms of the instructions can be moved into the architecture proper. After that time, it is expected that systems based on Northstar processors can be configured to generate a Program interrupt when the new form of ***mfcr*** is executed. If this expectation is met, the new forms of the instructions will generate a Program interrupt on all processors for which they are treated neither as the old forms nor as the new forms, and operating systems on the affected systems would be expected to emulate the new forms.

5.2 Floating-Point Processor Instructions

† The optional instructions described in this section are divided into two groups. Additional groups may be defined in the future.

- General Purpose group: ***fsqrt***, ***fsqrts***
- Graphics group: ***fres, frsqrte***, ***fsel***

An implementation that claims to support a given group implements all the instructions in the group.

5.2.1 Floating-Point Arithmetic Instructions

5.2.1.1 Floating-Point Elementary Arithmetic Instructions

Floating Square Root [*Single*] *A-form*

```
fsqrt    FRT,FRB          (Rc=0)
fsqrt.   FRT,FRB          (Rc=1)
```

63	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

```
fsqrts   FRT,FRB          (Rc=0)
fsqrts.  FRT,FRB          (Rc=1)
```

59	FRT	///	FRB	///	22	Rc
0	6	11	16	21	26	31

The square root of the floating-point operand in register FRB is placed into register FRT.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the Floating-Point Rounding Control field RN of the FPSCR and placed into register FRT.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[1]	VXSQRT
< 0	QNaN[1]	VXSQRT
-0	-0	None
$+\infty$	$+\infty$	None
SNaN	QNaN[1]	VXSNAN
QNaN	QNaN	None

[1]No result if $\text{FPSCR}_{\text{VE}} = 1$.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$.

†

Special Registers Altered:

FPRF FR FI
FX XX
VXSNAN VXSQRT
CR1 (if Rc=1)

Floating Reciprocal Estimate Single A-form

```
fres     FRT,FRB          (Rc=0)
fres.    FRT,FRB          (Rc=1)
```

59	FRT	///	FRB	///	24	Rc
0	6	11	16	21	26	31

A single-precision estimate of the reciprocal of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 256 of the reciprocal of (FRB), i.e.,

$$\text{ABS}\left(\frac{\text{estimate} - 1/x}{1/x}\right) \le \frac{1}{256}$$

where x is the initial value in FRB. Note that the value placed into register FRT may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty$[1]	ZX
$+0$	$+\infty$[1]	ZX
$+\infty$	$+0$	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1]No result if $\text{FPSCR}_{\text{ZE}} = 1$.
[2]No result if $\text{FPSCR}_{\text{VE}} = 1$.

$\text{FPSCR}_{\text{FPRF}}$ is set to the class and sign of the result, except for Invalid Operation Exceptions when $\text{FPSCR}_{\text{VE}}=1$ and Zero Divide Exceptions when $\text{FPSCR}_{\text{ZE}}=1$.

†

Special Registers Altered:

FPRF FR (undefined) FI (undefined)
FX OX UX ZX
VXSNAN
CR1 (if Rc=1)

Architecture Note

No double-precision version of this instruction is provided because graphics applications are expected to need only the single-precision version, and no other important performance-critical applications are expected to need a double-precision version.

Floating Reciprocal Square Root Estimate A-form

```
frsqrte     FRT,FRB                 (Rc=0)
frsqrte.    FRT,FRB                 (Rc=1)
```

63	FRT	///	FRB	///	26	Rc
0	6	11	16	21	26	31

A double-precision estimate of the reciprocal of the square root of the floating-point operand in register FRB is placed into register FRT. The estimate placed into register FRT is correct to a precision of one part in 32 of the reciprocal of the square root of (FRB), i.e.,

$$\mathrm{ABS}\left(\frac{\text{estimate} - 1/\sqrt{x}}{1/\sqrt{x}}\right) \le \frac{1}{32}$$

where x is the initial value in FRB. Note that the value placed into register FRT may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
$-\infty$	QNaN[2]	VXSQRT
< 0	QNaN[2]	VXSQRT
−0	$-\infty$[1]	ZX
+0	$+\infty$[1]	ZX
$+\infty$	+0	None
SNaN	QNaN[2]	VXSNAN
QNaN	QNaN	None

[1]No result if $\mathrm{FPSCR}_{ZE} = 1$.
[2]No result if $\mathrm{FPSCR}_{VE} = 1$.

FPSCR_{FPRF} is set to the class and sign of the result, except for Invalid Operation Exceptions when $\mathrm{FPSCR}_{VE}=1$ and Zero Divide Exceptions when $\mathrm{FPSCR}_{ZE}=1$.

†

Special Registers Altered:
FPRF FR (undefined) FI (undefined)
FX ZX
VXSNAN VXSQRT
CR1 (if Rc=1)

Architecture Note

No single-precision version of this instruction is provided because it would be superfluous: if (FRB) is representable in single format, then so is (FRT).

5.2.2 Floating-Point Select Instruction

Floating Select A-form

```
fsel     FRT,FRA,FRC,FRB                 (Rc=0)
fsel.    FRT,FRA,FRC,FRB                 (Rc=1)
```

63	FRT	FRA	FRB	FRC	23	Rc
0	6	11	16	21	26	31

```
if (FRA) ≥ 0.0 then FRT ← (FRC)
else FRT ← (FRB)
```

The floating-point operand in register FRA is compared to the value zero. If the operand is greater than or equal to zero, register FRT is set to the contents of register FRC. If the operand is less than zero or is a NaN, register FRT is set to the contents of register FRB. The comparison ignores the sign of zero (i.e., regards +0 as equal to −0).

†

Special Registers Altered:
CR1 (if Rc=1)

Architecture Note

The *Select* instruction is similar to a *Move* instruction, and therefore does not alter the FPSCR.

Programming Note

Examples of uses of this instruction can be found in Sections C.2, “Floating-Point Conversions” on page 180 and C.3, “Floating-Point Selection” on page 182.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section C.3.4, “Notes” on page 182.

† 5.3 Little-Endian

It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy ....

Jonathan Swift, *Gulliver's Travels*

† The Little-Endian facility permits a program to access
† storage using Little-Endian byte ordering.

5.3.1 Byte Ordering

If scalars (individual data items and instructions) were indivisible, then there would be no such concept as “byte ordering”. It is meaningless to talk of the “order” of bits or groups of bits within the smallest addressable unit of storage, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can be made up of more than one addressable unit of storage does the question of “order” arise.

For a machine in which the smallest addressable unit of storage is the 64-bit doubleword, there is no question of the ordering of “bytes” within doublewords. All transfers of individual scalars to and from storage (e.g., between registers and storage) are of doublewords, and the address of the “byte” containing the high-order 8 bits of a scalar is no different from the address of a “byte” containing any other part of the scalar.

For PowerPC AS, as for most computers currently available, the smallest addressable unit of storage is the 8-bit byte. Many scalars are halfwords, words, or doublewords, which consist of groups of bytes. When a word-length scalar is moved from a register to storage, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order 8 bits of the scalar, which byte contains the next-highest-order 8 bits, and so on.

Given a scalar that spans multiple bytes, the choice of byte ordering is essentially arbitrary. There are 4! = 24 ways to specify the ordering of four bytes within a word, but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order (“leftmost”) 8 bits of the scalar, the next sequential address to the next-highest-order 8 bits, and so on. This is called *Big-Endian* because the “big end” of the scalar, considered as a binary number, comes first in storage. IBM RISC System/6000, IBM System/370, and Motorola 680x0 are examples of computers using this byte ordering.
- The ordering that assigns the lowest address to the lowest-order (“rightmost”) 8 bits of the scalar, the next sequential address to the next-lowest-order 8 bits, and so on. This is called *Little-Endian* because the “little end” of the scalar, considered as a binary number, comes first in storage. DEC VAX and Intel x86 are examples of computers using this byte ordering.

5.3.2 Structure Mapping Examples

Figure 41 on page 143 shows an example of a C language structure **s** containing an assortment of scalars and one character string. The value assumed to be in each structure element is shown in hex in the C comments; these values are used below to show how the bytes making up each structure element are mapped into storage.

C structure mapping rules permit the use of padding (skipped bytes) in order to align the scalars on desirable boundaries. Figures 42 and 43 show each scalar aligned at its natural boundary. This alignment introduces padding of four bytes between **a** and **b**, one byte between **d** and **e**, and two bytes between **e** and **f**. The same amount of padding is present for both Big-Endian and Little-Endian mappings.

5.3.2.1 Big-Endian Mapping

The Big-Endian mapping of structure **s** is shown in Figure 42. Addresses are shown in hex at the left of each doubleword, and in small figures below each byte. The contents of each byte, as indicated in the C example in Figure 41, are shown in hex (as characters for the elements of the string).

```
struct {
    int       a;      /*  0x1112_1314                          word             */
    double    b;      /*  0x2122_2324_2526_2728                doubleword       */
    char *    c;      /*  0x3132_3334                          word             */
    char      d[7];   /*  'A', 'B', 'C', 'D', 'E', 'F', 'G'    array of bytes   */
    short     e;      /*  0x5152                               halfword         */
    int       f;      /*  0x6162_6364                          word             */
} s;
```

Figure 41. C structure 's', showing values of elements

00	11	12	13	14				
	00	01	02	03	04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	31	32	33	34	'A'	'B'	'C'	'D'
	10	11	12	13	14	15	16	17
18	'E'	'F'	'G'		51	52		
	18	19	1A	1B	1C	1D	1E	1F
20	61	62	63	64				
	20	21	22	23				

Figure 42. Big-Endian mapping of structure 's'

5.3.2.2 Little-Endian Mapping

The same structure **s** is shown mapped Little-Endian in Figure 43. Doublewords are shown laid out from right to left, which is the common way of showing storage maps for Little-Endian machines.

				11	12	13	14	00
07	06	05	04	03	02	01	00	
21	22	23	24	25	26	27	28	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	'C'	'B'	'A'	31	32	33	34	10
17	16	15	14	13	12	11	10	
		51	52		'G'	'F'	'E'	18
1F	1E	1D	1C	1B	1A	19	18	
				61	62	63	64	20
				23	22	21	20	

Figure 43. Little-Endian mapping of structure 's'

5.3.3 PowerPC AS Byte Ordering

The body of each of the three PowerPC AS Architecture Books, Book I, *PowerPC AS User Instruction Set Architecture*, Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*, is written as if a PowerPC AS system runs only in Big-Endian mode. In fact, a PowerPC AS system can instead run in Little-Endian mode, in which the instruction set behaves as if the byte ordering were Little-Endian, and can change Endian mode dynamically. The remainder of Section 5.3 describes how the mode is controlled, and how running in Little-Endian mode differs from running in Big-Endian mode.

5.3.3.1 Controlling PowerPC AS Byte Ordering

The Endian mode of a PowerPC AS processor is controlled by two bits: the LE (Little-Endian Mode) bit specifies the current mode of the processor, and the ILE (Interrupt Little-Endian Mode) bit specifies the mode that the processor enters when the system error handler is invoked. For both bits, a value of 0 specifies Big-Endian mode and a value of 1 specifies Little-Endian mode. The location of these bits and the requirements for altering them are described in Book III, *PowerPC AS Operating Environment Architecture*.

When a PowerPC AS system comes up after power-on-reset, Big-Endian mode is in effect (see Book III, *PowerPC AS Operating Environment Architecture*). Thereafter, methods described in Book III can be used to change the mode, as can both invoking the system error handler and returning from the system error handler.

Programming Note

For a discussion of software synchronization requirements when altering the LE and ILE bits, see Book III (e.g., to the chapter entitled “Synchronization Requirements for Special Registers and for Lookaside Buffers” in Book III).

Architecture Note

The LE and ILE bits must be defined in Book III in a manner such that they can be changed dynamically and that the LE bit can easily be treated as part of a process' state.

5.3.3.2 PowerPC AS Little-Endian Byte Ordering

One might expect that a PowerPC AS system running in Little-Endian mode would have to perform a 2-way, 4-way, or 8-way byte swap when transferring a halfword, word, or doubleword to or from storage, e.g., when transferring data between storage and a General Purpose Register or Floating-Point Register, when fetching instructions, and when transferring data between storage and an Input/Output (I/O) device. PowerPC AS systems do not do such swapping, but instead achieve the effect of Little-Endian byte ordering by modifying the low-order three bits of the effective address (EA) as described below. Individual scalars actually appear in storage in Big-Endian byte order.

The modification affects only the addresses presented to the storage subsystem (see Book III, *PowerPC AS Operating Environment Architecture*). All effective addresses in architecturally defined registers, as well as the Current Instruction Address (CIA) and Next Instruction Address (NIA), are independent of Endian mode. For example:

- The effective address placed into the Link Register by a *Branch* instruction with LK=1 is equal to the CIA of the *Branch* instruction + 4;
- The effective address placed into RA by a *Load/Store with Update* instruction is the value computed as described in the instruction description; and
- The effective addresses placed into System Registers when the system error handler is invoked (e.g., SRR0, DAR: see Book III, *PowerPC AS Operating Environment Architecture*) are those that were computed or would have been computed by the interrupted program.

Architecture Note

In fact, the modification is performed on the real address (see Book III, *PowerPC AS Operating Environment Architecture*), and not on the effective address at all. Describing the modification this way makes it obvious why all effective addresses in architecturally defined registers, and in the CIA and NIA, are unaffected. However, this simple description cannot be used here, because real addresses are not defined in Book I.

The modification is performed regardless of whether address translation is enabled or disabled and, if address translation is enabled, regardless of the translation mechanism used (see Book III, *PowerPC AS Operating Environment Architecture*). The actual transfer of data and instructions to and from storage is unaffected (and thus unencumbered by multiplexors for byte swapping).

The modification of the low-order three bits of the effective address in Little-Endian mode is done as follows, for access to an individual aligned scalar. (Alignment is as determined before this modification.) Access to an individual unaligned scalar or to multiple scalars is described in subsequent sections, as is access to certain architecturally defined data in storage, data in caches (e.g., see Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*), etc.

In Little-Endian mode, the effective address is computed in the same way as in Big-Endian mode. Then, in Little-Endian mode only, the low-order three bits of the effective address are Exclusive ORed with a three-bit value that depends on the length of the operand (1, 2, 4, or 8 bytes), as shown in Table 2. This modified effective address is then presented to the storage subsystem, and data of the specified length are transferred to or from the addressed (as modified) storage locations(s).

Data Length (bytes)	EA Modification
1	XOR with 0b111
2	XOR with 0b110
4	XOR with 0b100
8	(no change)

Table 2. PowerPC AS Little-Endian, effective address modification for individual aligned scalars

The effective address modification makes it appear to the processor that individual aligned scalars are stored Little-Endian, while in fact they are stored Big-Endian but in different bytes within doublewords from the order in which they are stored in Big-Endian mode.

For example, in Little-Endian mode structure **s** would be placed in storage as follows, from the point of view of the storage subsystem (i.e., after the effective address modification described above).

00					**11**	**12**	**13**	**14**
	00	01	02	03	04	05	06	07
08	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
	08	09	0A	0B	0C	0D	0E	0F
10	**'D'**	**'C'**	**'B'**	**'A'**	**31**	**32**	**33**	**34**
	10	11	12	13	14	15	16	17
18			**51**	**52**		**'G'**	**'F'**	**'E'**
	18	19	1A	1B	1C	1D	1E	1F
20					**61**	**62**	**63**	**64**
	20	21	22	23	24	25	26	27

Figure 44. PowerPC AS Little-Endian, structure 's' in storage subsystem

Figure 44 is identical to Figure 43 except that the byte numbers within each doubleword are reversed. (This identity is in some sense an artifact of depicting storage as a sequence of doublewords. If storage is instead depicted as a sequence of words, a single byte stream, etc., then no such identity appears. However, regardless of the unit in which storage is depicted or accessed, the address of a given byte in Figure 44 differs from the address of the same byte in Figure 43 only in the low-order three bits, and the sum of the two 3-bit values that comprise the low-order three bits of the two addresses is equal to 7. Depicting storage as a sequence of doublewords makes this relationship easy to see.)

Because of the modification performed on effective addresses, structure **s** appears to the processor to be mapped into storage as follows when the processor is in Little-Endian mode.

				11	**12**	**13**	**14**	00
07	06	05	04	03	02	01	00	
21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	08
0F	0E	0D	0C	0B	0A	09	08	
'D'	**'C'**	**'B'**	**'A'**	**31**	**32**	**33**	**34**	10
17	16	15	14	13	12	11	10	
		51	**52**		**'G'**	**'F'**	**'E'**	18
1F	1E	1D	1C	1B	1A	19	18	
				61	**62**	**63**	**64**	20
				23	22	21	20	

Figure 45. PowerPC AS Little-Endian, structure 's' as seen by processor

Notice that, as seen by the program executing in the processor, the mapping for structure **s** is identical to the Little-Endian mapping shown in Figure 43. From a point of view outside the processor, however, the addresses of the bytes making up structure **s** are as shown in Figure 44. These addresses match neither the Big-Endian mapping of Figure 42 nor the Little-Endian mapping of Figure 43; allowance must be made for this in certain circumstances (e.g., when performing I/O: see Section 5.3.7).

The following four sections describe in greater detail the effects of running in Little-Endian mode on accessing data, on fetching instructions, on explicitly accessing the caches and any address translation lookaside buffers (e.g., see Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*), and on doing I/O.

Architecture Note

The capability of running in Little-Endian mode is provided in order to facilitate porting Little-Endian application programs and operating systems to PowerPC AS systems.

5.3.4 PowerPC AS Data Addressing in Little-Endian Mode

5.3.4.1 Individual Aligned Scalars

When the storage operand is aligned for any instruction in the following classes, the effective address presented to the storage subsystem is computed as described in Section 5.3.3.2: *Fixed-Point Load*, *Fixed-*
† *Point Store*, *Load and Store with Byte Reversal*,
† *Floating-Point Load*, *Floating-Point Store* (including
† ***stfiwx***), and *Load And Reserve* and *Store Conditional*
† (see Book II).

The *Load and Store with Byte Reversal* instructions have the effect of loading or storing data in the opposite Endian mode from that in which the processor is running. That is, data are loaded or stored in Little-Endian order if the processor is running in Big-Endian mode, and in Big-Endian order if the processor is running in Little-Endian mode.

5.3.4.2 Other Scalars

As described below, the system alignment error handler may be (see Section “Individual Unaligned Scalars”) or is (see Section “Multiple Scalars” on page 146) invoked if attempt is made in Little-Endian mode to execute any of the instructions described in the following two subsections.

Individual Unaligned Scalars

The “trick” of Exclusive ORing the low-order three bits of the effective address of an individual scalar does not work unless the scalar is aligned. In Little-Endian mode, PowerPC AS processors may cause the system alignment error handler to be invoked whenever any of the *Load* or *Store* instructions listed in Section 5.3.4.1 is issued with an unaligned effective address, regardless of whether such an access could be handled without invoking the system alignment error handler in Big-Endian mode.

PowerPC AS processors are not *required* to invoke the system alignment error handler when an unaligned access is attempted in Little-Endian mode. The implementation may handle some or all such accesses without invoking the system alignment error handler, just as in Big-Endian mode. The architectural requirement is that halfwords, words, and doublewords be placed in storage such that the Little-Endian effective address of the lowest-order byte is the effective address computed by the *Load* or *Store* instruction, the Little-Endian address of the next-
† lowest-order byte is one greater, and so on. (*Load*
† *And Reserve* and *Store Conditional* differ somewhat from the rest of the instructions listed in Section 5.3.4.1, in that neither the implementation nor the system alignment error handler is expected to handle

these four instructions “correctly” if their operands are not aligned.)

Figure 46 shows an example of a word **w** stored at Little-Endian address 5. The word is assumed to contain the binary value 0x1112_1314.

12	**13**	**14**						00
07	06	05	04	03	02	01	00	
							11	08
0F	0E	0D	0C	0B	0A	09	08	

Figure 46. Little-Endian mapping of word 'w' stored at address 5

In Little-Endian mode word **w** would be placed in storage as follows, from the point of view of the storage subsystem (i.e., after the effective address modification described in Section 5.3.3.2).

00	**12**	**13**	**14**					
	00	01	02	03	04	05	06	07
08								**11**
	08	09	0A	0B	0C	0D	0E	0F

Figure 47. PowerPC AS Little-Endian, word 'w' stored at address 5 in storage subsystem

Notice that the unaligned word **w** in Figure 47 spans two doublewords. The two parts of the unaligned word are not contiguous as seen by the storage subsystem.

An implementation may choose to support some but not all unaligned Little-Endian accesses. For example, an unaligned Little-Endian access that is contained within a single doubleword may be supported, while one that spans doublewords may cause the system alignment error handler to be invoked.

Multiple Scalars

PowerPC AS has two classes of instructions that handle multiple scalars, namely the *Load and Store Multiple* instructions and the *Move Assist* instructions. Because both classes of instructions potentially deal with more than one word-length scalar, neither class is amenable to the effective address modification described in Section 5.3.3.2 (e.g., pairs of aligned words would be accessed in reverse order from what the program would expect). Attempting to execute any of these instructions in Little-Endian mode causes the system alignment error handler to be invoked.

Quadword Instructions

If $MSR_{LE}=1$ and $MSR_{TA}=1$ and ***lq*** or ***stq*** is executed, either the system alignment error handler is invoked or the results are boundedly undefined.

5.3.4.3 Page Table

The layout of the Page Table in storage (see Book III, *PowerPC AS Operating Environment Architecture*) is independent of Endian mode. A given byte in the Page Table must be accessed using an effective address appropriate to the mode of the executing program (e.g., the high-order byte of a Page Table Entry must be accessed with an effective address ending with 0b000 in Big-Endian mode, and with an effective address ending with 0b111 in Little-Endian mode).

Engineering Note

An implementation that uses software assistance to facilitate the hardware's searching and alteration of the Page Table must supply two separate software routines, one for Big-Endian mode and one for Little-Endian mode.

5.3.5 PowerPC AS Instruction Addressing in Little-Endian Mode

Each PowerPC AS instruction occupies an aligned word in storage. The processor fetches and executes instructions as if the CIA were advanced by four for each sequentially fetched instruction. When the processor is in Little-Endian mode, the effective address presented to the storage subsystem in order to fetch an instruction is the value from the CIA, modified as described in Section 5.3.3.2 for aligned word-length scalars. A Little-Endian program is thus an array of aligned Little-Endian words, with each word fetched and executed in order (discounting branches and invocations of the system error handler).

Figure 48 shows an example of a small assembly language program **p**.

```
loop:
     cmplwi   r5,0
     beq      done
     lwzux    r4,r5,r6
     add      r7,r7,r4
     subi     r5,r5,4
     b        loop
done:
     stw      r7,total
```

Figure 48. Assembly language program 'p'

The Big-Endian mapping for program **p** is shown in Figure 49 (assuming the program starts at address 0).

00	loop: cmplwi r5,0	beq done
	00 01 02 03	04 05 06 07
08	lwzux r4,r5,r6	add r7,r7,r4
	08 09 0A 0B	0C 0D 0E 0F
10	subi r5,r5,4	b loop
	10 11 12 13	14 15 16 17
18	done: stw r7,total	
	18 19 1A 1B	1C 1D 1E 1F

Figure 49. Big-Endian mapping of program 'p'

The same program **p** is shown mapped Little-Endian in Figure 50.

beq done	loop: cmplwi r5,0	00
07 06 05 04	03 02 01 00	
add r7,r7,r4	lwzux r4,r5,r6	08
0F 0E 0D 0C	0B 0A 09 08	
b loop	subi r5,r5,4	10
17 16 15 14	13 12 11 10	
	done: stw r7,total	18
1F 1E 1D 1C	1B 1A 19 18	

Figure 50. Little-Endian mapping of program 'p'

In Little-Endian mode program **p** would be placed in storage as follows, from the point of view of the storage subsystem (i.e., after the effective address modification described in Section 5.3.3.2).

00	beq done	loop: cmplwi r5,0
	00 01 02 03	04 05 06 07
08	add r7,r7,r4	lwzux r4,r5,r6
	08 09 0A 0B	0C 0D 0E 0F
10	b loop	subi r5,r5,4
	10 11 12 13	14 15 16 17
18		done: stw r7,total
	18 19 1A 1B	1C 1D 1E 1F

Figure 51. PowerPC AS Little-Endian, program 'p' in storage subsystem

Figure 51 is identical to Figure 50 except that the byte numbers within each doubleword are reversed. (This identity is in some sense an artifact of depicting storage as a sequence of doublewords. If storage is instead depicted as a sequence of words, a single byte stream, etc., then no such identity appears. However, regardless of the unit in which storage is depicted or accessed, the address of a given byte in Figure 51 differs from the address of the same byte in Figure 50 only in the low-order three bits, and the sum of the two 3-bit values that comprise the low-order three bits of the two addresses is equal to 7. Depicting storage as a sequence of doublewords makes this relationship easy to see.)

Each individual machine instruction appears in storage as a 32-bit integer containing the value described in the instruction description, regardless of the Endian mode. This is a consequence of the fact that individual aligned scalars are mapped in storage in Big-Endian byte order.

Notice that, as seen by the processor when executing program **p**, the mapping for program **p** is identical to the Little-Endian mapping shown in Figure 50. From a point of view outside the processor, however, the addresses of the bytes making up program **p** are as shown in Figure 51. These addresses match neither the Big-Endian mapping of Figure 49 nor the Little-Endian mapping of Figure 50.

All instruction effective addresses visible to an executing program are the effective addresses that are computed by that program or, in the case of the system error handler, effective addresses that were or could have been computed by the interrupted program. These effective addresses are independent of Endian mode. Examples for Little-Endian mode include the following.

- An instruction address placed into the Link Register by a *Branch* instruction with LK=1, or an instruction address saved in a System Register when the system error handler is invoked, is the effective address that a program executing in Little-Endian mode would use to access the instruction as a data word using a *Load* instruction.
- An offset in a relative *Branch* instruction (*Branch* or *Branch Conditional* with AA=0) reflects the difference between the addresses of the branch and target instructions, using the addresses that a program executing in Little-Endian mode would use to access the instructions as data words using *Load* instructions.
- A target address in an absolute *Branch* instruction (*Branch* or *Branch Conditional* with AA=1) is the address that a program executing in Little-Endian mode would use to access the target instruction as a data word using a *Load* instruction.
- The storage locations that contain the first set of instructions executed by each kind of system error handler must be set in a manner consistent with the Endian mode in which the system error handler will be invoked. (These sets of instructions occupy architecturally defined locations: see Book III, *PowerPC AS Operating Environment Architecture*.) Thus if the system error handler is to be invoked in Little-Endian mode, the first set of instructions for each kind of system error handler must appear in storage, from the point of view of the storage subsystem (i.e., after the effective address modification described in Section 5.3.3.2), with the pair of instructions within each doubleword reversed

from the order in which they are to be executed. (If the instructions are placed into storage by a program running in the same Endian mode as that in which the system error handler will be invoked, the appropriate order will be achieved naturally.)

Programming Note

In general, a given subroutine in storage cannot be shared between programs running in different Endian modes. This affects the sharing of subroutine libraries.

Engineering Note

If the Endian mode changes because an ***sc***, *Trap*, or ***rfid*** (see Book III) instruction was executed or because an interrupt occurred, subsequent instructions must be executed in the correct order as determined by the new Endian mode (MSR_{LE}) regardless of the Endian mode that was in effect when the instructions were fetched into the instruction cache. Implementations that conditionally reverse the order of instructions within doublewords depending on the current Endian mode when placing instructions into the instruction cache must correct the instruction order when the Endian mode is changed by the occurrences listed at the beginning of this Note. However, restrictions may apply when the Endian mode is changed by the execution of an ***mtmsr***[***d***] or ***rfscv*** instruction; e.g., see the chapter entitled "Synchronization Requirements for Special Registers and for Lookaside Buffers" in Book III.

5.3.6 PowerPC AS Cache Management Instructions in Little-Endian Mode

Instructions for explicitly accessing the caches (see Book II, *PowerPC AS Virtual Environment Architecture*) are unaffected by Endian mode. (Identification of the block to be accessed is not affected by the low-order three bits of the effective address.)

5.3.7 PowerPC AS I/O in Little-Endian Mode

Input/output (I/O), such as writing the contents of a large area of storage to disk, transfers a byte stream on both Big-Endian and Little-Endian systems. For the disk transfer, the first byte of the area is written to the first byte of the disk record and so on.

For a PowerPC AS system running in Big-Endian mode, I/O transfers happen "naturally" because the byte that the processor sees as byte 0 is the same one that the storage subsystem sees as byte 0.

For a PowerPC AS system running in Little-Endian mode this is not the case, because of the modification of the low-order three bits of the effective address when the processor accesses storage. In order for I/O transfers to transfer byte streams properly, in Little-Endian mode I/O transfers must be performed as if the bytes transferred were accessed one byte at a time, using the address modification described in Section 5.3.3.2 for single-byte scalars. This does not mean that I/O on Little-Endian PowerPC AS systems must use only 1-byte-wide transfers; data transfers can be as wide as desired, but the order of the bytes transferred within doublewords must appear as if the bytes were fetched or stored one byte at a time. See the System Architecture documentation for a given PowerPC AS system for details on the transfer width and byte ordering on that system.

However, not all I/O done on PowerPC AS systems is for large areas of storage as described above. I/O can be performed with certain devices merely by storing to or loading from addresses that are associated with the devices (the terms "memory-mapped I/O" and "programmed I/O" or "PIO" are used for this). For such PIO transfers, care must be taken when defining the addresses to be used, for these addresses are subject to the effective address modification shown in Table 2 on page 144. A *Load* or *Store* instruction that maps to a control register on a device may require that the value loaded or stored have its bytes reversed; if this is required, the *Load and Store with Byte Reversal* instructions can be used. Any requirement for such byte reversal for a particular I/O device register is independent of whether the PowerPC AS system is running in Big-Endian or Little-Endian mode.

Similarly, the address sent to an I/O device by an ***eciwx*** or ***ecowx*** instruction (see Book II, *PowerPC AS Virtual Environment Architecture*) is subject to the effective address modification shown in Table 2.

5.3.8 Origin of Endian

The terms *Big-Endian* and *Little-Endian* come from Part I, Chapter 4, of Jonathan Swift's *Gulliver's Travels*. Here is the complete passage, from the edition printed in 1734 by George Faulkner in Dublin.

> ... our Histories of six Thousand Moons make no Mention of any other Regions, than the two great Empires of *Lilliput* and *Blefuscu.* Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past. It began upon the following Occasion. It is allowed on all Hands, that the primitive Way of breaking Eggs before we eat them, was upon the larger End: But his present Majes-

ty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father, published an Edict, commanding all his Subjects, upon great Penalties, to break the smaller End of their Eggs. The People so highly resented this Law, that our Histories tell us, there have been six Rebellions raised on that Account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of *Blefuscu*; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy: But the Books of the *Big-Endians* have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of *Blefuscu* did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet *Lustrog*, in the fifty-fourth Chapter of the *Brundrecal*, (which is their *Alcoran*.) This, however, is thought to be a mere Strain upon the text: For the Words are these; *That all true Believers shall break their Eggs at the convenient End*: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the Power of the chief Magistrate to determine. Now the *Big-Endian* Exiles have found so much Credit in the Emperor of *Blefuscu's* Court; and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which Time we have lost Forty Capital Ships, and a much greater Number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckoned to be somewhat greater than ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us: and his Imperial Majesty, placing great Confidence in your Valour and Strength, hath commanded me to lay this Account of his Affairs before you.

Appendix A. Suggested Floating-Point Models

A.1 Floating-Point Round to Single-Precision Model

The following describes algorithmically the operation of the *Floating Round to Single-Precision* instruction.

```
If $(FRB)_{1:11} < 897$ and $(FRB)_{1:63} > 0$ then
    Do
        If $FPSCR_{UE} = 0$ then goto Disabled Exponent Underflow
        If $FPSCR_{UE} = 1$ then goto Enabled Exponent Underflow
    End

If $(FRB)_{1:11} > 1150$ and $(FRB)_{1:11} < 2047$ then
    Do
        If $FPSCR_{OE} = 0$ then goto Disabled Exponent Overflow
        If $FPSCR_{OE} = 1$ then goto Enabled Exponent Overflow
    End

If $(FRB)_{1:11} > 896$ and $(FRB)_{1:11} < 1151$ then goto Normal Operand

If $(FRB)_{1:63} = 0$ then goto Zero Operand

If $(FRB)_{1:11} = 2047$ then
    Do
        If $(FRB)_{12:63} = 0$ then goto Infinity Operand
        If $(FRB)_{12} = 1$ then goto QNaN Operand
        If $(FRB)_{12} = 0$ and $(FRB)_{13:63} > 0$ then goto SNaN Operand
    End
```

Disabled Exponent Underflow:

```
sign ← $(FRB)_0$
If $(FRB)_{1:11}$ = 0 then
    Do
        exp ← – 1022
        $frac_{0:52}$ ← 0b0 || $(FRB)_{12:63}$
    End
If $(FRB)_{1:11}$ > 0 then
    Do
        exp ← $(FRB)_{1:11}$ – 1023
        $frac_{0:52}$ ← 0b1 || $(FRB)_{12:63}$
    End
Denormalize operand:
    G || R || X ← 0b000
    Do while exp < – 126
        exp ← exp + 1
        $frac_{0:52}$ || G || R || X ← 0b0 || $frac_{0:52}$ || G || (R | X)
    End
$FPSCR_{UX}$ ← ($frac_{24:52}$ || G || R || X) > 0
Round Single(sign,exp,$frac_{0:52}$,G,R,X)
$FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$
If $frac_{0:52}$ = 0 then
    Do
        $FRT_0$ ← sign
        $FRT_{1:63}$ ← 0
        If sign = 0 then $FPSCR_{FPRF}$ ← "+ zero"
        If sign = 1 then $FPSCR_{FPRF}$ ← "– zero"
    End
If $frac_{0:52}$ > 0 then
    Do
        If $frac_0$ = 1 then
            Do
                If sign = 0 then $FPSCR_{FPRF}$ ← "+ normal number"
                If sign = 1 then $FPSCR_{FPRF}$ ← "– normal number"
            End
        If $frac_0$ = 0 then
            Do
                If sign = 0 then $FPSCR_{FPRF}$ ← "+ denormalized number"
                If sign = 1 then $FPSCR_{FPRF}$ ← "– denormalized number"
            End
        Normalize operand:
            Do while $frac_0$ = 0
                exp ← exp– 1
                $frac_{0:52}$ ← $frac_{1:52}$ || 0b0
            End
        $FRT_0$ ← sign
        $FRT_{1:11}$ ← exp + 1023
        $FRT_{12:63}$ ← $frac_{1:52}$
    End
Done
```

Enabled Exponent Underflow:

```
FPSCR$_{UX}$ ← 1
sign ← (FRB)$_0$
If (FRB)$_{1:11}$ = 0 then
    Do
        exp ← – 1022
        frac$_{0:52}$ ← 0b0 || (FRB)$_{12:63}$
    End
If (FRB)$_{1:11}$ > 0 then
    Do
        exp ← (FRB)$_{1:11}$ – 1023
        frac$_{0:52}$ ← 0b1 || (FRB)$_{12:63}$
    End
Normalize operand:
    Do while frac$_0$ = 0
        exp ← exp – 1
        frac$_{0:52}$ ← frac$_{1:52}$ || 0b0
    End
Round Single(sign,exp,frac$_{0:52}$,0,0,0)
FPSCR$_{XX}$ ← FPSCR$_{XX}$ | FPSCR$_{FI}$
exp ← exp + 192
FRT$_0$ ← sign
FRT$_{1:11}$ ← exp + 1023
FRT$_{12:63}$ ← frac$_{1:52}$
If sign = 0 then FPSCR$_{FPRF}$ ← "+ normal number"
If sign = 1 then FPSCR$_{FPRF}$ ← "– normal number"
Done
```

Disabled Exponent Overflow:

```
FPSCR$_{OX}$ ← 1
If FPSCR$_{RN}$ = 0b00 then                /* Round to Nearest */
    Do
        If (FRB)$_0$ = 0 then FRT ← 0x7FF0_0000_0000_0000
        If (FRB)$_0$ = 1 then FRT ← 0xFFF0_0000_0000_0000
        If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+ infinity"
        If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "– infinity"
    End
If FPSCR$_{RN}$ = 0b01 then                /* Round toward Zero */
    Do
        If (FRB)$_0$ = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)$_0$ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+ normal number"
        If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "– normal number"
    End
If FPSCR$_{RN}$ = 0b10 then                /* Round toward +Infinity */
    Do
        If (FRB)$_0$ = 0 then FRT ← 0x7FF0_0000_0000_0000
        If (FRB)$_0$ = 1 then FRT ← 0xC7EF_FFFF_E000_0000
        If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+ infinity"
        If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "– normal number"
    End
If FPSCR$_{RN}$ = 0b11 then                /* Round toward – Infinity */
    Do
        If (FRB)$_0$ = 0 then FRT ← 0x47EF_FFFF_E000_0000
        If (FRB)$_0$ = 1 then FRT ← 0xFFF0_0000_0000_0000
        If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+ normal number"
        If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "– infinity"
    End
FPSCR$_{FR}$ ← undefined
FPSCR$_{FI}$ ← 1
FPSCR$_{XX}$ ← 1
Done
```

Enabled Exponent Overflow:

```
    sign ← (FRB)$_0$
    exp ← (FRB)$_{1:11}$ − 1023
    frac$_{0:52}$ ← 0b1 || (FRB)$_{12:63}$
    Round Single(sign,exp,frac$_{0:52}$,0,0,0)
    FPSCR$_{XX}$ ← FPSCR$_{XX}$ | FPSCR$_{FI}$
Enabled Overflow:
    FPSCR$_{OX}$ ← 1
    exp ← exp − 192
    FRT$_0$ ← sign
    FRT$_{1:11}$ ← exp + 1023
    FRT$_{12:63}$ ← frac$_{1:52}$
    If sign = 0 then FPSCR$_{FPRF}$ ← "+normal number"
    If sign = 1 then FPSCR$_{FPRF}$ ← "−normal number"
    Done
```

Zero Operand:

```
    FRT ← (FRB)
    If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+zero"
    If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "−zero"
    FPSCR$_{FR\ FI}$ ← 0b00
    Done
```

Infinity Operand:

```
    FRT ← (FRB)
    If (FRB)$_0$ = 0 then FPSCR$_{FPRF}$ ← "+infinity"
    If (FRB)$_0$ = 1 then FPSCR$_{FPRF}$ ← "−infinity"
    FPSCR$_{FR\ FI}$ ← 0b00
    Done
```

QNaN Operand:

```
    FRT ← (FRB)$_{0:34}$ || $^{29}$0
    FPSCR$_{FPRF}$ ← "QNaN"
    FPSCR$_{FR\ FI}$ ← 0b00
    Done
```

SNaN Operand:

```
    FPSCR$_{VXSNAN}$ ← 1
    If FPSCR$_{VE}$ = 0 then
        Do
            FRT$_{0:11}$ ← (FRB)$_{0:11}$
            FRT$_{12}$ ← 1
            FRT$_{13:63}$ ← (FRB)$_{13:34}$ || $^{29}$0
            FPSCR$_{FPRF}$ ← "QNaN"
        End
    FPSCR$_{FR\ FI}$ ← 0b00
    Done
```

Normal Operand:

```
sign ← (FRB)$_0$
exp ← (FRB)$_{1:11}$ – 1023
frac$_{0:52}$ ← 0b1 || (FRB)$_{12:63}$
Round Single(sign,exp,frac$_{0:52}$,0,0,0)
FPSCR$_{XX}$ ← FPSCR$_{XX}$ | FPSCR$_{FI}$
If exp > 127 and FPSCR$_{OE}$ = 0 then go to Disabled Exponent Overflow
If exp > 127 and FPSCR$_{OE}$ = 1 then go to Enabled Overflow
FRT$_0$ ← sign
FRT$_{1:11}$ ← exp + 1023
FRT$_{12:63}$ ← frac$_{1:52}$
If sign = 0 then FPSCR$_{FPRF}$ ← “+normal number”
If sign = 1 then FPSCR$_{FPRF}$ ← “–normal number”
Done
```

Round Single*(sign,exp,frac$_{0:52}$,G,R,X)*:

```
inc ← 0
lsb ← frac$_{23}$
gbit ← frac$_{24}$
rbit ← frac$_{25}$
xbit ← (frac$_{26:52}$||G||R||X)≠ 0
If FPSCR$_{RN}$ = 0b00 then                /* Round to Nearest */
    Do                  /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If FPSCR$_{RN}$ = 0b10 then                /* Round toward +Infinity */
    Do                  /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If FPSCR$_{RN}$ = 0b11 then                /* Round toward – Infinity */
    Do                  /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac$_{0:23}$ ← frac$_{0:23}$ + inc
If carry_out = 1 then
    Do
        frac$_{0:23}$ ← 0b1 || frac$_{0:22}$
        exp ← exp + 1
    End
frac$_{24:52}$ ← $^{29}$0
FPSCR$_{FR}$ ← inc
FPSCR$_{FI}$ ← gbit | rbit | xbit
Return
```

A.2 Floating-Point Convert to Integer Model

The following describes algorithmically the operation of the *Floating Convert To Integer* instructions.

```
If Floating Convert To Integer Word then
    Do
        round_mode ← $FPSCR_{RN}$
        tgt_precision ← “32-bit integer”
    End

If Floating Convert To Integer Word with round toward Zero then
    Do
        round_mode ← 0b01
        tgt_precision ← “32-bit integer”
    End

If Floating Convert To Integer Doubleword then
    Do
        round_mode ← $FPSCR_{RN}$
        tgt_precision ← “64-bit integer”
    End

If Floating Convert To Integer Doubleword with round toward Zero then
    Do
        round_mode ← 0b01
        tgt_precision ← “64-bit integer”
    End

sign ← $(FRB)_0$
If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12:63}$ = 0 then goto Infinity Operand
If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12}$ = 0 then goto SNaN Operand
If $(FRB)_{1:11}$ = 2047 and $(FRB)_{12}$ = 1 then goto QNaN Operand
If $(FRB)_{1:11}$ > 1086 then goto Large Operand

If $(FRB)_{1:11}$ > 0 then exp ← $(FRB)_{1:11}$ − 1023   /* exp − bias */
If $(FRB)_{1:11}$ = 0 then exp ← −1022
If $(FRB)_{1:11}$ > 0 then $frac_{0:64}$ ← 0b01 || $(FRB)_{12:63}$ || $^{11}0$   /* normal; need leading 0 for later complement */
If $(FRB)_{1:11}$ = 0 then $frac_{0:64}$ ← 0b00 || $(FRB)_{12:63}$ || $^{11}0$   /* denormal */

gbit || rbit || xbit ← 0b000
Do i=1,63−exp   /* do the loop 0 times if exp = 63 */
    $frac_{0:64}$ || gbit || rbit || xbit ← 0b0 || $frac_{0:64}$ || gbit || (rbit | xbit)
End

Round Integer(sign,$frac_{0:64}$,gbit,rbit,xbit,round_mode)

If sign = 1 then $frac_{0:64}$ ← ¬ $frac_{0:64}$ + 1   /* needed leading 0 for $-2^{64} <$ (FRB) $< -2^{63}$ */

If tgt_precision = “32-bit integer” and $frac_{0:64} > 2^{31}-1$ then goto Large Operand
If tgt_precision = “64-bit integer” and $frac_{0:64} > 2^{63}-1$ then goto Large Operand
If tgt_precision = “32-bit integer” and $frac_{0:64} < -2^{31}$ then goto Large Operand
If tgt_precision = “64-bit integer” and $frac_{0:64} < -2^{63}$ then goto Large Operand

$FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$

If tgt_precision = “32-bit integer” then FRT ← 0xuuuu_uuuu || $frac_{33:64}$   /* u is undefined hex digit */
If tgt_precision = “64-bit integer” then FRT ← $frac_{1:64}$
$FPSCR_{FPRF}$ ← undefined
Done
```

Round Integer(*sign,frac$_{0:64}$,gbit,rbit,xbit,round_mode****)***:

```
inc ← 0
If round_mode = 0b00 then              /* Round to Nearest */
    Do                 /* comparisons ignore u bits */
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0bu11uu then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0bu011u then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If round_mode = 0b10 then              /* Round toward +Infinity */
    Do                 /* comparisons ignore u bits */
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If round_mode = 0b11 then              /* Round toward – Infinity */
    Do                 /* comparisons ignore u bits */
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || frac$_{64}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac$_{0:64}$ ← frac$_{0:64}$ + inc
FPSCR$_{FR}$ ← inc
FPSCR$_{FI}$ ← gbit | rbit | xbit
Return
```

Infinity Operand:

```
FPSCR$_{FR\ FI\ VXCVI}$ ← 0b001
If FPSCR$_{VE}$ = 0 then Do
    If tgt_precision = "32-bit integer" then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCR$_{FPRF}$ ← undefined
    End
Done
```

SNaN Operand:

```
FPSCR$_{FR\ FI\ VXSNAN\ VXCVI}$ ← 0b0011
If FPSCR$_{VE}$ = 0 then
    Do
        If tgt_precision = "32-bit integer" then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = "64-bit integer" then FRT ← 0x8000_0000_0000_0000
        FPSCR$_{FPRF}$ ← undefined
    End
Done
```

QNaN Operand:

```
FPSCR$_{FR\ FI\ VXCVI}$ ← 0b001
If FPSCR$_{VE}$ = 0 then
    Do
        If tgt_precision = “32-bit integer” then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        If tgt_precision = “64-bit integer” then FRT ← 0x8000_0000_0000_0000
        FPSCR$_{FPRF}$ ← undefined
    End
Done
```

Large Operand:

```
FPSCR$_{FR\ FI\ VXCVI}$ ← 0b001
If FPSCR$_{VE}$ = 0 then Do
    If tgt_precision = “32-bit integer” then
        Do
            If sign = 0 then FRT ← 0xuuuu_uuuu_7FFF_FFFF   /* u is undefined hex digit */
            If sign = 1 then FRT ← 0xuuuu_uuuu_8000_0000   /* u is undefined hex digit */
        End
    Else
        Do
            If sign = 0 then FRT ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then FRT ← 0x8000_0000_0000_0000
        End
    FPSCR$_{FPRF}$ ← undefined
    End
Done
```

A.3 Floating-Point Convert from Integer Model

The following describes algorithmically the operation of the *Floating Convert From Integer Doubleword* instruction.

```
sign ← $(FRB)_0$
exp ← 63
$frac_{0:63}$ ← (FRB)

If $frac_{0:63}$ = 0 then go to Zero Operand

If sign = 1 then $frac_{0:63}$ ← ¬ $frac_{0:63}$ + 1

Do while $frac_0$ = 0 /* do the loop 0 times if (FRB) = maximum negative integer */
    $frac_{0:63}$ ← $frac_{1:63}$ || 0b0
    exp ← exp – 1
End

Round Float(sign,exp,$frac_{0:63}$,$FPSCR_{RN}$)

If sign = 0 then $FPSCR_{FPRF}$ ← "+normal number"
If sign = 1 then $FPSCR_{FPRF}$ ← "–normal number"
$FRT_0$ ← sign
$FRT_{1:11}$ ← exp + 1023   /* exp + bias */
$FRT_{12:63}$ ← $frac_{1:52}$
Done
```

Zero Operand:

```
$FPSCR_{FR\ FI}$ ← 0b00
$FPSCR_{FPRF}$ ← "+zero"
FRT ← 0x0000_0000_0000_0000
Done
```

Round Float*(sign,exp,frac*$_{0:63}$*,round_mode)*:

```
inc ← 0
lsb ← $frac_{52}$
gbit ← $frac_{53}$
rbit ← $frac_{54}$
xbit ← $frac_{55:63} > 0$
If round_mode = 0b00 then            /* Round to Nearest */
    Do              /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If round_mode = 0b10 then            /* Round toward +Infinity */
    Do              /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If round_mode = 0b11 then            /* Round toward – Infinity */
    Do              /* comparisons ignore u bits */
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
$frac_{0:52}$ ← $frac_{0:52}$ + inc
If carry_out = 1 then exp ← exp + 1
$FPSCR_{FR}$ ← inc
$FPSCR_{FI}$ ← gbit | rbit | xbit
$FPSCR_{XX}$ ← $FPSCR_{XX}$ | $FPSCR_{FI}$
Return
```

Appendix B. 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 that defines simple shorthand for the most frequently used forms of *Branch Conditional, Compare, Trap, Select, Rotate and Shift,* and certain other instructions.

† Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

B.1 Symbols

The following symbols are defined for use in instructions (basic or extended mnemonics) that specify a Condition Register field or a Condition Register bit. The first five (lt, ..., un) identify a bit number within a CR field. The remainder (cr0, ..., cr7) identify a CR field. An expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used to identify a CR bit.

Symbol	Value	Meaning
lt	0	Less than
gt	1	Greater than
eq	2	Equal
so	3	Summary overflow
ic	3	Incomparable (after ***cmpla***)
un	3	Unordered (after floating-point comparison)
cr0	0	CR Field 0
cr1	1	CR Field 1
cr2	2	CR Field 2
cr3	3	CR Field 3
cr4	4	CR Field 4
cr5	5	CR Field 5
cr6	6	CR Field 6
cr7	7	CR Field 7

The extended mnemonics in Sections B.2.2 and B.3 require identification of a CR bit: if one of the CR field symbols is used, it must be multiplied by 4 and added to a bit-number-within-CR-field (value in the range 0-3, explicit or symbolic). The extended mnemonics in Sections B.2.3 and B.5 require identification of a CR field: if one of the CR field symbols is used, it must *not* be multiplied by 4. (For the extended mnemonics in Section B.2.3, the bit number within the CR field is part of the extended mnemonic. The programmer identifies the CR field, and the Assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

B.2 Branch Mnemonics

The mnemonics discussed in this section are variations of the *Branch Conditional* instructions.

Note: ***bclr, bclrl, bcctr***, and ***bcctrl*** each serve as both a basic and an extended mnemonic. The Assembler will recognize a ***bclr, bclrl, bcctr***, or ***bcctrl*** mnemonic with three operands as the basic form, and a ***bclr, bclrl, bcctr***, or ***bcctrl*** mnemonic with two operands as the extended form. In the extended form the BH operand is omitted and assumed to be 0b00. Similarly, for all the extended mnemonics described in Sections B.2.2 - B.2.4 that devolve to any of these four basic mnemonics the BH operand can either be coded or omitted. If it is omitted it is assumed to be 0b00.

B.2.1 BO and BI Fields

The 5-bit BO and BI fields control whether the branch is taken. Providing an extended mnemonic for every possible combination of these fields would be neither useful nor practical. The mnemonics described in Sections B.2.2 - B.2.4 include the most useful cases. Other cases can be coded using a basic *Branch Conditional* mnemonic (***bc***, ***bclr***, ***bcctr***) with the appropriate operands.

B.2.2 Simple Branch Mnemonics

Instructions using one of the mnemonics in Table 3 that tests a Condition Register bit specify the corresponding bit as the first operand. The symbols defined in Section B.1 can be used in this operand.

Notice that there are no extended mnemonics for relative and absolute unconditional branches. For these the basic mnemonics ***b, ba, bl,*** and ***bla*** should be used.

Table 3. Simple branch mnemonics

Branch Semantics	**LR not Set**				**LR Set**			
	bc **Relative**	*bca* **Absolute**	*bclr* **To LR**	*bcctr* **To CTR**	*bcl* **Relative**	*bcla* **Absolute**	*bclrl* **To LR**	*bcctrl* **To CTR**
Branch unconditionally	–	–	blr	bctr	–	–	blrl	bctrl
Branch if $CR_{BI}=1$	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if $CR_{BI}=0$	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR nonzero	bdnz	bdnza	bdnzlr	–	bdnzl	bdnzla	bdnzlrl	–
Decrement CTR, branch if CTR nonzero and $CR_{BI}=1$	bdnzt	bdnzta	bdnztlr	–	bdnztl	bdnztla	bdnztlrl	–
Decrement CTR, branch if CTR nonzero and $CR_{BI}=0$	bdnzf	bdnzfa	bdnzflr	–	bdnzfl	bdnzfla	bdnzflrl	–
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	–	bdzl	bdzla	bdzlrl	–
Decrement CTR, branch if CTR zero and $CR_{BI}=1$	bdzt	bdzta	bdztlr	–	bdztl	bdztla	bdztlrl	–
Decrement CTR, branch if CTR zero and $CR_{BI}=0$	bdzf	bdzfa	bdzflr	–	bdzfl	bdzfla	bdzflrl	–

Examples

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR).

```
bdnz   target                        (equivalent to:   bc    16,0,target)
```

2. Same as (1) but branch only if CTR is nonzero and condition in CR0 is “equal”.

```
bdnzt  eq,target                     (equivalent to:   bc    8,2,target)
```

3. Same as (2), but “equal” condition is in CR5.

```
bdnzt  4*cr5+eq,target               (equivalent to:   bc    8,22,target)
```

† 4. Branch if bit 27 of CR is 0.

```
bf     27,target                     (equivalent to:   bc    4,27,target)
```

5. Same as (4), but set the Link Register. This is a form of conditional “call”.

```
bfl    27,target                     (equivalent to:   bcl   4,27,target)
```

B.2.3 Branch Mnemonics Incorporating Conditions

† In the mnemonics defined in Table 4 on page 164, the test of a bit in a Condition Register field is encoded in the
† mnemonic.

† Instructions using the mnemonics in Table 4 specify the Condition Register field as an optional first operand. One
† of the CR field symbols defined in Section B.1 can be used for this operand. If the CR field being tested is CR
| Field 0, this operand need not be specified unless the resulting basic mnemonic is ***bclr***[***l***] or ***bcctr***[***l***] and the BH
| operand is specified.

A standard set of codes has been adopted for the most common combinations of branch conditions.

Code	Meaning
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow
ic	Incomparable (after ***cmpla***)
ni	Not incomparable (after ***cmpla***)
un	Unordered (after floating-point comparison)
nu	Not unordered (after floating-point comparison)

These codes are reflected in the mnemonics shown in Table 4.

Table 4. Branch mnemonics incorporating conditions

Branch Semantics	**LR not Set**				**LR Set**			
	bc **Relative**	*bca* **Absolute**	*bclr* **To LR**	*bcctr* **To CTR**	*bcl* **Relative**	*bcla* **Absolute**	*bclrl* **To LR**	*bcctrl* **To CTR**
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than or equal	bge	bgea	bgelr	bgectr	bgel	bgela	bgelrl	bgectrl
Branch if greater than	bgt	bgta	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bnl	bnla	bnllr	bnlctr	bnll	bnlla	bnllrl	bnlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bnsl	bnsla	bnslrl	bnsctrl
Branch if incomparable	bic	bica	biclr	bicctr	bicl	bicla	biclrl	bicctrl
Branch if not incomparable	bni	bnia	bnilr	bnictr	bnil	bnila	bnilrl	bnictrl
Branch if unordered	bun	buna	bunlr	bunctr	bunl	bunla	bunlrl	bunctrl
Branch if not unordered	bnu	bnua	bnulr	bnuctr	bnul	bnula	bnulrl	bnuctrl

†

Examples

1. Branch if CR0 reflects condition “not equal”.

```
bne     target                        (equivalent to:    bc      4,2,target)
```

2. Same as (1), but condition is in CR3.

```
bne     cr3,target                    (equivalent to:    bc      4,14,target)
```

3. Branch to an absolute target if CR4 specifies “greater than”, setting the Link Register. This is a form of conditional “call”.

```
bgtla   cr4,target                    (equivalent to:    bcla    12,17,target)
```

4. Same as (3), but target address is in the Count Register.

```
bgtctrl cr4                           (equivalent to:    bcctrl  12,17,0)
```

B.2.4 Branch Prediction

Software can use the “at” bits of *Branch Conditional* instructions to provide a hint to the processor about the behavior of the branch. If, for a given such instruction, the branch is almost always taken or almost always not taken, a suffix can be added to the mnemonic indicating the value to be used for the “at” bits.

+ Predict branch to be taken (at=0b11)

– Predict branch not to be taken (at=0b10)

Such a suffix can be added to any *Branch Conditional* mnemonic, either basic or extended, that tests either the Count Register or a CR bit (but not both). Assemblers should use 0b00 as the default value for the “at” bits, indicating that software has offered no prediction.

Examples

1. Branch if CR0 reflects condition “less than”, specifying that the branch should be predicted to be taken.

   ```
   blt+   target
   ```

2. Same as (1), but target address is in the Link Register and the branch should be predicted not to be taken.

   ```
   bltlr–
   ```

B.3 Condition Register Logical Mnemonics

The *Condition Register Logical* instructions can be used to set (to 1), clear (to 0), copy, or invert a given Condition Register bit. Extended mnemonics are provided that allow these operations to be coded easily.

Table 5. Condition Register logical mnemonics

Operation	Extended Mnemonic	Equivalent to
Condition Register set	crset bx	creqv bx,bx,bx
Condition Register clear	crclr bx	crxor bx,bx,bx
Condition Register move	crmove bx,by	cror bx,by,by
Condition Register not	crnot bx,by	crnor bx,by,by

The symbols defined in Section B.1 can be used to identify the Condition Register bits.

Examples

1. Set CR bit 25.

   ```
   crset   25                       (equivalent to:   creqv   25,25,25)
   ```

2. Clear the SO bit of CR0.

   ```
   crclr   so                       (equivalent to:   crxor   3,3,3)
   ```

3. Same as (2), but SO bit to be cleared is in CR3.

   ```
   crclr   4*cr3+so                 (equivalent to:   crxor   15,15,15)
   ```

4. Invert the EQ bit.

   ```
   crnot   eq,eq                    (equivalent to:   crnor   2,2,2)
   ```

5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.

   ```
   crnot   4*cr5+eq,4*cr4+eq        (equivalent to:   crnor   22,18,18)
   ```

B.4 Subtract Mnemonics

B.4.1 Subtract Immediate

Although there is no “Subtract Immediate” instruction, its effect can be achieved by using an *Add Immediate* instruction with the immediate operand negated. Extended mnemonics are provided that include this negation, making the intent of the computation clearer.

```
subi    Rx,Ry,value              (equivalent to:   addi    Rx,Ry,– value)
subis   Rx,Ry,value              (equivalent to:   addis   Rx,Ry,– value)
subic   Rx,Ry,value              (equivalent to:   addic   Rx,Ry,– value)
subic.  Rx,Ry,value              (equivalent to:   addic.  Rx,Ry,– value)
```

B.4.2 Subtract

The *Subtract From* instructions subtract the second operand (RA) from the third (RB). Extended mnemonics are provided that use the more “normal” order, in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final “o” and/or “.” to cause the OE and/or Rc bit to be set in the underlying instruction.

```
sub    Rx,Ry,Rz          (equivalent to:    subf   Rx,Rz,Ry)
subc   Rx,Ry,Rz          (equivalent to:    subfc  Rx,Rz,Ry)
```

B.5 Compare Mnemonics

The L field in the fixed-point *Compare* instructions controls whether the operands are treated as 64-bit quantities
† or as 32-bit quantities. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

The BF field can be omitted if the result of the comparison is to be placed into CR Field 0. Otherwise the target CR field must be specified as the first operand. One of the CR field symbols defined in Section B.1 can be used for this operand.

Note: The basic *Compare* mnemonics of PowerPC AS are the same as those of POWER, but the POWER instructions have three operands while the PowerPC AS instructions have four. The Assembler will recognize a basic *Compare* mnemonic with three operands as the POWER form, and will generate the instruction with L=0. (Thus the Assembler must require that the BF field, which normally can be omitted when CR Field 0 is the target, be specified explicitly if L is.)

B.5.1 Doubleword Comparisons

Table 6. Doubleword compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare doubleword immediate	cmpdi bf,ra,si	cmpi bf,1,ra,si
Compare doubleword	cmpd bf,ra,rb	cmp bf,1,ra,rb
Compare logical doubleword immediate	cmpldi bf,ra,ui	cmpli bf,1,ra,ui
Compare logical doubleword	cmpld bf,ra,rb	cmpl bf,1,ra,rb

Examples

1. Compare register Rx and immediate value 100 as unsigned 64-bit integers and place result into CR0.
```
   cmpldi  Rx,100                    (equivalent to:     cmpli   0,1,Rx,100)
```
2. Same as (1), but place result into CR4.
```
   cmpldi  cr4,Rx,100                (equivalent to:     cmpli   4,1,Rx,100)
```
3. Compare registers Rx and Ry as signed 64-bit integers and place result into CR0.
```
   cmpd    Rx,Ry                     (equivalent to:     cmp     0,1,Rx,Ry)
```

B.5.2 Word Comparisons

Table 7. Word compare mnemonics

Operation	Extended Mnemonic	Equivalent to
Compare word immediate	cmpwi bf,ra,si	cmpi bf,0,ra,si
Compare word	cmpw bf,ra,rb	cmp bf,0,ra,rb
Compare logical word immediate	cmplwi bf,ra,ui	cmpli bf,0,ra,ui
Compare logical word	cmplw bf,ra,rb	cmpl bf,0,ra,rb

Examples

1. Compare bits 32:63 of register Rx and immediate value 100 as signed 32-bit integers and place result into CR0.
```
   cmpwi   Rx,100                    (equivalent to:     cmpi    0,0,Rx,100)
```
2. Same as (1), but place result into CR4.
```
   cmpwi   cr4,Rx,100                (equivalent to:     cmpi    4,0,Rx,100)
```
3. Compare bits 32:63 of registers Rx and Ry as unsigned 32-bit integers and place result into CR0.
```
   cmplw   Rx,Ry                     (equivalent to:     cmpl    0,0,Rx,Ry)
```

B.6 Trap Mnemonics

The mnemonics defined in Table 8 are variations of the *Trap* instructions, with the most useful values of TO represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the most common combinations of trap conditions.

Code	**Meaning**	**TO encoding**	**<**	**>**	**=**	$\overset{u}{<}$	$\overset{u}{>}$
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
(none)	Unconditional	31	1	1	1	1	1

These codes are reflected in the mnemonics shown in Table 8.

Table 8. Trap mnemonics

Trap Semantics	**64-bit Comparison**		**32-bit Comparison**	
	tdi **Immediate**	*td* **Register**	*twi* **Immediate**	*tw* **Register**
Trap unconditionally	–	–	–	trap
Trap unconditionally with parameters	tdui	tdu	twui	twu
Trap if less than	tdlti	tdlt	twlti	twlt
Trap if less than or equal	tdlei	tdle	twlei	twle
Trap if equal	tdeqi	tdeq	tweqi	tweq
Trap if greater than or equal	tdgei	tdge	twgei	twge
Trap if greater than	tdgti	tdgt	twgti	twgt
Trap if not less than	tdnli	tdnl	twnli	twnl
Trap if not equal	tdnei	tdne	twnei	twne
Trap if not greater than	tdngi	tdng	twngi	twng
Trap if logically less than	tdllti	tdllt	twllti	twllt
Trap if logically less than or equal	tdllei	tdlle	twllei	twlle
Trap if logically greater than or equal	tdlgei	tdlge	twlgei	twlge
Trap if logically greater than	tdlgti	tdlgt	twlgti	twlgt
Trap if logically not less than	tdlnli	tdlnl	twlnli	twlnl
Trap if logically not greater than	tdlngi	tdlng	twlngi	twlng

Examples

1. Trap if register Rx is not 0.

```
tdnei   Rx,0                    (equivalent to:   tdi   24,Rx,0)
```

2. Same as (1), but comparison is to register Ry.

```
tdne    Rx,Ry                   (equivalent to:   td    24,Rx,Ry)
```

3. Trap if bits 32:63 of register Rx, considered as a 32-bit quantity, are logically greater than 0x7FF.

```
twlgti  Rx,0x7FF                (equivalent to:   twi   1,Rx,0x7FF)
```

4. Trap unconditionally.

```
trap                            (equivalent to:   tw    31,0,0)
```

5. Trap unconditionally with immediate parameters Rx and Ry

```
tdu     Rx,Ry                   (equivalent to:   td    31,Rx,Ry)
```

B.7 Trap on XER mnemonics

The mnemonics defined in Table 9 on page 170 are variations of the *Trap on XER* instruction, with the most useful values of XBI represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the XER bits that can be tested (and are not reserved). The code identifies the condition under which the system trap handler is invoked.

Code	**Meaning**	**TO$_4$**
lt	Less than	1
eq	Equal	1
gt	Greater than	1
nl	Not less than	0
ne	Not equal	0
ng	Not greater than	0
ic	Incomparable	1
so	Summary overflow	1
ov	Overflow	1
ca	Carry	1
oc	Offset carry	1
no	Not offset carry	0
nt02	Not T02 (type field bits 0:2 mismatch or no tag)	0
nt07	Not T07 (type field bits 0:7 mismatch or no tag)	0
ntag	Not XER TAG	0
ds	Decimal summary	1

These codes are reflected in the mnemonics shown in Table 9 on page 170.

Table 9. Trap on XER mnemonics

Trap on XER semantics	*txer*
Trap if less than	txerlt
Trap if equal	txereq
Trap if greater than	txergt
Trap if not less than	txernl
Trap if not equal	txerne
Trap if not greater than	txerng
Trap if incomparable	txeric
Trap if summary overflow	txerso
Trap if overflow	txerov
Trap if carry	txerca
Trap if offset carry	txeroc
Trap if not T02 (trap if type field bits 0:2 mismatch or no tag)	txernt02
Trap if not T07 (trap if type field bits 0:7 mismatch or no tag)	txernt07
Trap if not TAG	txerntag
Trap if not decimal summary	txerds

Examples

1. Trap if the EQ bit is set in the XER.

```
txereq     256                    (equivalent to:    txer   1,256,38)
```

2. Trap if the IC bit is set in the XER.

```
txeric                            (equivalent to:    txer   1,0,39)
```

3. Trap if the IC bit is set in the XER.

```
txerntag   5                      (equivalent to:    txer   0,5,43)
```

B.8 Select mnemonics

The mnemonics defined in Table 10 on page 171 are variations of the *Select* instructions, with the most useful values of XBI represented in the mnemonic rather than specified as a numeric operand.

A standard set of codes has been adopted for the XER bits that can be tested (and are not reserved). The code identifies the condition under which the first source operand is selected: if the bit tested is 1 then the first source operand is placed into the target register, otherwise the second source operand is placed into the target register.

Code	**Meaning**
lt	Less than
eq	Equal
gt	Greater than
ic	Incomparable
so	Summary overflow
ov	Overflow
ca	Carry
oc	Offset carry
t02	T02 (i.e., type field bits 0:2 match) 0
t07	T07 (i.e., type field bits 0:7 match) 0
tag	XER TAG
ds	Decimal summary

These codes are reflected in the mnemonics shown in Table 10.

Table 10. Select mnemonics

Select semantics	*selii* **Immediate-Immediate**	*selir* **Immediate-Register**	*selri* **Register-Immediate**	*selrr* **Register-Register**
Select if less than	selltii	selltir	selltri	selltrr
Select if equal	seleqii	seleqir	seleqri	seleqrr
Select if greater than	selgtii	selgtir	selgtri	selgtrr
Select if incomparable	selicii	selicir	selicri	selicrr
Select if summary overflow	selsoii	selsoir	selsori	selsorr
Select if overflow	selovii	selovir	selovri	selovrr
Select if carry	selcaii	selcair	selcari	selcarr
Select if offset carry	selocii	selocir	selocri	selocrr
Select if T02	selt02ii	selt02ir	selt02ri	selt02rr
Select if T07	selt07ii	selt07ir	selt07ri	selt07rr
Select if XER TAG	seltagii	seltagir	seltagri	seltagrr
Select if decimal summary	seldsii	seldsir	seldsri	seldsrr

Examples

1. Set register Rx to 1 if the EQ bit is set in the XER, and to 0 otherwise.

```
seleqii  Rx,1,0                          (equivalent to:     selii   Rx,1,0,38)
```

2. Same as (1) but use the value in Ry if the EQ bit is set.

```
seleqri  Rx,Ry,0                         (equivalent to:     selri   Rx,Ry,0,38)
```

3. Set Rx to the value in Ry if the OV bit is set in the XER, and leave Rx unchanged otherwise.

```
selovrr  Rx,Ry,Rx                        (equivalent to:     selrr   Rx,Ry,Rx,33)
```

4. Set Rx to the absolute value of Ry.

```
neg.     Rx,Ry
selgtrr  Rx,Rx,Ry                        (equivalent to:     selrr   Rx,Rx,Ry,37)
```

5. Set Rx to the minimum of Ry and Rz, regarded as signed 64-bit numbers.

```
cmpd     Ry,Rz                           (equivalent to:     cmp     0,1,Ry,Rz)
selltrr  Rx,Ry,Rz                        (equivalent to:     selrr   Rx,Ry,Rz,36)
```

B.9 Rotate and Shift Mnemonics

The *Rotate and Shift* instructions provide powerful and general ways to manipulate register contents, but can be difficult to understand. Extended mnemonics are provided that allow some of the simpler operations to be coded easily.

Mnemonics are provided for the following types of operation.

Extract Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register to 0.

Insert Select a left-justified or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged. (No extended mnemonic is provided for insertion of a left-justified field when operating on doublewords, because such an insertion requires more than one instruction.)

Rotate Rotate the contents of a register right or left *n* bits without masking.

Shift Shift the contents of a register right or left *n* bits, clearing vacated bits to 0 (logical shift).

Clear Clear the leftmost or rightmost *n* bits of a register to 0.

Clear left and shift left
Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

B.9.1 Operations on Doublewords

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction.

Table 11. Doubleword rotate and shift mnemonics

Operation	Extended Mnemonic	Equivalent to
Extract and left justify immediate	extldi ra,rs,*n*,*b* ($n > 0$)	rldicr ra,rs,*b*,*n*– 1
Extract and right justify immediate	extrdi ra,rs,*n*,*b* ($n > 0$)	rldicl ra,rs,*b*+ *n*,64– *n*
Insert from right immediate	insrdi ra,rs,*n*,*b* ($n > 0$)	rldimi ra,rs,64– (*b*+ *n*),*b*
Rotate left immediate	rotldi ra,rs,*n*	rldicl ra,rs,*n*,0
Rotate right immediate	rotrdi ra,rs,*n*	rldicl ra,rs,64– *n*,0
Rotate left	rotld ra,rs,rb	rldcl ra,rs,rb,0
Shift left immediate	sldi ra,rs,*n* ($n < 64$)	rldicr ra,rs,*n*,63– *n*
Shift right immediate	srdi ra,rs,*n* ($n < 64$)	rldicl ra,rs,64– *n*,*n*
Clear left immediate	clrldi ra,rs,*n* ($n < 64$)	rldicl ra,rs,0,*n*
Clear right immediate	clrrdi ra,rs,*n* ($n < 64$)	rldicr ra,rs,0,63– *n*
Clear left and shift left immediate	clrlsldi ra,rs,*b*,*n* ($n \leq b < 64$)	rldic ra,rs,*n*,*b*– *n*

Examples

1. Extract the sign bit (bit 0) of register Ry and place the result right-justified into register Rx.
```
extrdi   Rx,Ry,1,0                    (equivalent to:    rldicl  Rx,Ry,1,63)
```
2. Insert the bit extracted in (1) into the sign bit (bit 0) of register Rz.
```
insrdi   Rz,Rx,1,0                    (equivalent to:    rldimi  Rz,Rx,63,0)
```
3. Shift the contents of register Rx left 8 bits.
```
sldi     Rx,Rx,8                      (equivalent to:    rldicr  Rx,Rx,8,55)
```
4. Clear the high-order 32 bits of register Ry and place the result into register Rx.

```
clrldi    Rx,Ry,32                          (equivalent to:     rldicl  Rx,Ry,0,32)
```

B.9.2 Operations on Words

All these mnemonics can be coded with a final "." to cause the Rc bit to be set in the underlying instruction. The operations as described above apply to the low-order 32 bits of the registers, as if the registers were 32-bit registers. The Insert operations either preserve the high-order 32 bits of the target register or place rotated data there; the other operations clear these bits.

Table 12. Word rotate and shift mnemonics

Operation	Extended Mnemonic	Equivalent to
Extract and left justify immediate	extlwi ra,rs,*n*,*b* (*n* > 0)	rlwinm ra,rs,*b*,0,*n*– 1
Extract and right justify immediate	extrwi ra,rs,*n*,*b* (*n* > 0)	rlwinm ra,rs,*b*+ *n*,32– *n*,31
Insert from left immediate	inslwi ra,rs,*n*,*b* (*n* > 0)	rlwimi ra,rs,32– *b*,*b*,(*b*+ *n*)– 1
Insert from right immediate	insrwi ra,rs,*n*,*b* (*n* > 0)	rlwimi ra,rs,32– (*b*+ *n*),*b*,(*b*+ *n*)– 1
Rotate left immediate	rotlwi ra,rs,*n*	rlwinm ra,rs,*n*,0,31
Rotate right immediate	rotrwi ra,rs,*n*	rlwinm ra,rs,32– *n*,0,31
Rotate left	rotlw ra,rs,rb	rlwnm ra,rs,rb,0,31
Shift left immediate	slwi ra,rs,*n* (*n* < 32)	rlwinm ra,rs,*n*,0,31– *n*
Shift right immediate	srwi ra,rs,*n* (*n* < 32)	rlwinm ra,rs,32– *n*,*n*,31
Clear left immediate	clrlwi ra,rs,*n* (*n* < 32)	rlwinm ra,rs,0,*n*,31
Clear right immediate	clrrwi ra,rs,*n* (*n* < 32)	rlwinm ra,rs,0,0,31– *n*
Clear left and shift left immediate	clrlslwi ra,rs,*b*,*n* (*n* ≤ *b* < 32)	rlwinm ra,rs,*n*,*b*– *n*,31– *n*

Examples

1. Extract the sign bit (bit 32) of register Ry and place the result right-justified into register Rx.

```
extrwi    Rx,Ry,1,0                       (equivalent to:     rlwinm  Rx,Ry,1,31,31)
```

2. Insert the bit extracted in (1) into the sign bit (bit 32) of register Rz.

```
insrwi    Rz,Rx,1,0                       (equivalent to:     rlwimi  Rz,Rx,31,0,0)
```

3. Shift the contents of register Rx left 8 bits, clearing the high-order 32 bits.

```
slwi      Rx,Rx,8                         (equivalent to:     rlwinm  Rx,Rx,8,0,23)
```

4. Clear the high-order 16 bits of the low-order 32 bits of register Ry and place the result into register Rx, clearing the high-order 32 bits of register Rx.

```
clrlwi    Rx,Ry,16                        (equivalent to:     rlwinm  Rx,Ry,0,16,31)
```

B.10 Move To/From Special Purpose Register Mnemonics

The ***mtspr*** and ***mfspr*** instructions specify a Special Purpose Register (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.

Table 13. Extended mnemonics for moving to/from an SPR

Special Purpose Register	Move To SPR		Move From SPR	
	Extended	**Equivalent to**	**Extended**	**Equivalent to**
Fixed-Point Exception Register (XER)	mtxer Rx	mtspr 1,Rx	mfxer Rx	mfspr Rx,1
Link Register (LR)	mtlr Rx	mtspr 8,Rx	mflr Rx	mfspr Rx,8
Count Register (CTR)	mtctr Rx	mtspr 9,Rx	mfctr Rx	mfspr Rx,9

Examples

1. Copy the contents of Rx to the XER.
   ```
   mtxer   Rx                              (equivalent to:      mtspr  1,Rx)
   ```
2. Copy the contents of the LR to register Rx.
   ```
   mflr    Rx                              (equivalent to:      mfspr  Rx,8)
   ```
3. Copy the contents of register Rx to the CTR.
   ```
   mtctr   Rx                              (equivalent to:      mtspr  9,Rx)
   ```

B.11 Miscellaneous Mnemonics

No-op

Many PowerPC AS instructions can be coded in a way such that, effectively, no operation is performed. An extended mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the no-op that will trigger this.

```
nop                                        (equivalent to:      ori    0,0,0)
```

Load Immediate

The ***addi*** and ***addis*** instructions can be used to load an immediate value into a register. Extended mnemonics are provided to convey the idea that no addition is being performed but merely data movement (from the immediate field of the instruction to a register).

Load a 16-bit signed immediate value into register Rx.

```
li      Rx,value                           (equivalent to:      addi   Rx,0,value)
```

Load a 16-bit signed immediate value, shifted left by 16 bits, into register Rx.

```
lis     Rx,value                           (equivalent to:      addis  Rx,0,value)
```

Load Address

This mnemonic permits computing the value of a base-displacement operand, using the ***addi*** instruction which normally requires separate register and immediate operands.

```
la    Rx,D(Ry)                      (equivalent to:    addi  Rx,Ry,D)
```

The ***la*** mnemonic is useful for obtaining the address of a variable specified by name, allowing the Assembler to supply the base register number and compute the displacement. If the variable v is located at offset Dv bytes from the address in register Rv, and the Assembler has been told to use register Rv as a base for references to the data structure containing v, then the following line causes the address of v to be loaded into register Rx.

```
la    Rx,v                          (equivalent to:    addi  Rx,Rv,Dv)
```

Programming Note

† Unlike the $+_{tea}$ computation for a *Load* or *Store* instruction, the ***la*** computation cannot cause an Effective
† Address Overflow exception, and the result of the ***la*** computation may be different from that of the corre-
† sponding $+_{tea}$ computation if the latter would have produced an Effective Address Overflow exception.

† In an earlier AS/400 architecture called “IMPI”, ***la*** performed boundary checking and could cause an Effective
† Address Overflow exception.

Move Register

Several PowerPC AS instructions can be coded in a way such that they simply copy the contents of one register to another. An extended mnemonic is provided to convey the idea that no computation is being performed but merely data movement (from one register to another).

The following instruction copies the contents of register Ry to register Rx. This mnemonic can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction.

```
mr    Rx,Ry                         (equivalent to:    or    Rx,Ry,Ry)
```

Complement Register

Several PowerPC AS instructions can be coded in a way such that they complement the contents of one register and place the result into another register. An extended mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of register Ry and places the result into register Rx. This mnemonic can be coded with a final “.” to cause the Rc bit to be set in the underlying instruction.

```
not   Rx,Ry                         (equivalent to:    nor   Rx,Ry,Ry)
```

Move To Condition Register

This mnemonic permits copying the contents of the low-order 32 bits of a GPR to the Condition Register, using the same style as the ***mfcr*** instruction.

```
mtcr  Rx                            (equivalent to:    mtcrf 0xFF,Rx)
```

Appendix C. Programming Examples

†

C.1 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is initially defined to be a shift of an N-doubleword quantity (64-bit mode) or an N-word quantity (32-bit mode), where $N>1$. (This definition is relaxed somewhat for 32-bit mode, below.) The quantity to be shifted is contained in N registers (in the low-order 32 bits in 32-bit mode). The shift amount is specified either by an immediate value in the instruction, or by bits 57:63 (64-bit mode) or 58:63 (32-bit mode) of a register.

The examples shown below distinguish between the cases $N=2$ and $N>2$. If $N=2$, the shift amount may be in the range 0 through 127 (64-bit mode) or 0 through 63 (32-bit mode), which are the maximum ranges supported by the *Shift* instructions used. However if $N>2$, the shift amount must be in the range 0 through 63 (64-bit mode) or 0 through 31 (32-bit mode), in order for the examples to yield the desired result. The specific instance shown for $N>2$ is $N=3$: extending those code sequences to larger N is straightforward, as is reducing them to the case $N=2$ when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case $N=3$ is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit mode for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR 6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

For $N>2$, the number of instructions required is $2N-1$ (immediate shifts) or $3N-1$ (non-immediate shifts).

Multiple-precision shifts in 64-bit mode

Shift Left Immediate, N = 3 (shift amnt < 64)

```
        rldicr      r5,r4,sh,63-sh
        rldimi      r4,r3,0,sh
        rldicl      r4,r4,sh,0
        rldimi      r3,r2,0,sh
        rldicl      r3,r3,sh,0
```

Shift Left, N = 2 (shift amnt < 128)

```
        subfic      r31,r6,64
        sld         r2,r2,r6
        srd         r0,r3,r31
        or          r2,r2,r0
        addi        r31,r6,-64
        sld         r0,r3,r31
        or          r2,r2,r0
        sld         r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 64)

```
        subfic      r31,r6,64
        sld         r2,r2,r6
        srd         r0,r3,r31
        or          r2,r2,r0
        sld         r3,r3,r6
        srd         r0,r4,r31
        or          r3,r3,r0
        sld         r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 64)

```
        rldimi      r4,r3,0,64-sh
        rldicl      r4,r4,64-sh,0
        rldimi      r3,r2,0,64-sh
        rldicl      r3,r3,64-sh,0
        rldicl      r2,r2,64-sh,sh
```

Shift Right, N = 2 (shift amnt < 128)

```
        subfic      r31,r6,64
        srd         r3,r3,r6
        sld         r0,r2,r31
        or          r3,r3,r0
        addi        r31,r6,-64
        srd         r0,r2,r31
        or          r3,r3,r0
        srd         r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 64)

```
        subfic      r31,r6,64
        srd         r4,r4,r6
        sld         r0,r3,r31
        or          r4,r4,r0
        srd         r3,r3,r6
        sld         r0,r2,r31
        or          r3,r3,r0
        srd         r2,r2,r6
```

Multiple-precision shifts in 32-bit mode

Shift Left Immediate, N = 3 (shift amnt < 32)

```
        rlwinm      r2,r2,sh,0,31-sh
        rlwimi      r2,r3,sh,32-sh,31
        rlwinm      r3,r3,sh,0,31-sh
        rlwimi      r3,r4,sh,32-sh,31
        rlwinm      r4,r4,sh,0,31-sh
```

Shift Left, N = 2 (shift amnt < 64)

```
        subfic      r31,r6,32
        slw         r2,r2,r6
        srw         r0,r3,r31
        or          r2,r2,r0
        addi        r31,r6,-32
        slw         r0,r3,r31
        or          r2,r2,r0
        slw         r3,r3,r6
```

Shift Left, N = 3 (shift amnt < 32)

```
        subfic      r31,r6,32
        slw         r2,r2,r6
        srw         r0,r3,r31
        or          r2,r2,r0
        slw         r3,r3,r6
        srw         r0,r4,r31
        or          r3,r3,r0
        slw         r4,r4,r6
```

Shift Right Immediate, N = 3 (shift amnt < 32)

```
        rlwinm      r4,r4,32-sh,sh,31
        rlwimi      r4,r3,32-sh,0,sh-1
        rlwinm      r3,r3,32-sh,sh,31
        rlwimi      r3,r2,32-sh,0,sh-1
        rlwinm      r2,r2,32-sh,sh,31
```

Shift Right, N = 2 (shift amnt < 64)

```
        subfic      r31,r6,32
        srw         r3,r3,r6
        slw         r0,r2,r31
        or          r3,r3,r0
        addi        r31,r6,-32
        srw         r0,r2,r31
        or          r3,r3,r0
        srw         r2,r2,r6
```

Shift Right, N = 3 (shift amnt < 32)

```
        subfic      r31,r6,32
        srw         r4,r4,r6
        slw         r0,r3,r31
        or          r4,r4,r0
        srw         r3,r3,r6
        slw         r0,r2,r31
        or          r3,r3,r0
        srw         r2,r2,r6
```

Multiple-precision shifts in 64-bit mode, continued

Shift Right Algebraic Immediate, N = 3 (shift amnt < 64)

```
        rldimi    r4,r3,0,64-sh
        rldicl    r4,r4,64-sh,0
        rldimi    r3,r2,0,64-sh
        rldicl    r3,r3,64-sh,0
        sradi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 128)

```
        subfic    r31,r6,64
        srd       r3,r3,r6
        sld       r0,r2,r31
        or        r3,r3,r0
        addic.    r31,r6,-64
        srad      r0,r2,r31
        selrr     r3,r0,r3,gt
        srad      r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 64)

```
        subfic    r31,r6,64
        srd       r4,r4,r6
        sld       r0,r3,r31
        or        r4,r4,r0
        srd       r3,r3,r6
        sld       r0,r2,r31
        or        r3,r3,r0
        srad      r2,r2,r6
```

Multiple-precision shifts in 32-bit mode, continued

Shift Right Algebraic Immediate, N = 3 (shift amnt < 32)

```
        rlwinm    r4,r4,32-sh,sh,31
        rlwimi    r4,r3,32-sh,0,sh-1
        rlwinm    r3,r3,32-sh,sh,31
        rlwimi    r3,r2,32-sh,0,sh-1
        srawi     r2,r2,sh
```

Shift Right Algebraic, N = 2 (shift amnt < 64)

```
        subfic    r31,r6,32
        srw       r3,r3,r6
        slw       r0,r2,r31
        or        r3,r3,r0
        addic.    r31,r6,-32
        sraw      r0,r2,r31
        selrr     r3,r0,r3,gt
        sraw      r2,r2,r6
```

Shift Right Algebraic, N = 3 (shift amnt < 32)

```
        subfic    r31,r6,32
        srw       r4,r4,r6
        slw       r0,r3,r31
        or        r4,r4,r0
        srw       r3,r3,r6
        slw       r0,r2,r31
        or        r3,r3,r0
        sraw      r2,r2,r6
```

C.2 Floating-Point Conversions

This section gives examples of how the *Floating-Point Conversion* instructions can be used to perform various conversions.

Warning: Some of the examples use the optional ***fsel*** instruction. Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities: see Section C.3.4, “Notes” on page 182.

C.2.1 Conversion from Floating-Point Number to Floating-Point Integer

The full *convert to floating-point integer* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1 and the result is returned in FPR 3.

```
mtfsb0   23            #clear VXCVI
fctid[z] f3,f1         #convert to fx int
fcfid    f3,f3         #convert back again
mcrfs    7,5           #VXCVI to CR
bf       31,$+8        #skip if VXCVI was 0
fmr      f3,f1         #input was fp int
```

C.2.2 Conversion from Floating-Point Number to Signed Fixed-Point Integer Doubleword

The full *convert to signed fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctid[z] f2,f1         #convert to dword int
stfd     f2,disp(r1)   #store float
ld       r3,disp(r1)   #load dword
```

C.2.3 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Doubleword

The full *convert to unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value 2^{64}– 2048 is in FPR 3, the value 2^{63} is in FPR 4 and GPR 4, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f5,f3,f1      #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4      #subtract 2**63
fcmpu    cr2,f2,f4     #use diff if ≥ 2**63
fsel     f2,f5,f5,f2
fctid[z] f2,f2         #convert to fx int
stfd     f2,disp(r1)   #store float
ld       r3,disp(r1)   #load dword
blt      cr2,$+8       #add 2**63 if input
add      r3,r3,r4      #  was ≥ 2**63
```

C.2.4 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full *convert to signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fctiw[z] f2,f1         #convert to fx int
stfd     f2,disp(r1)   #store float
lwa      r3,disp+4(r1) #load word algebraic
```

C.2.5 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full *convert to unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR 1, the value 0 is in FPR 0, the value $2^{32}-1$ is in FPR 3, the result is returned in GPR 3, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
fsel     f2,f1,f1,f0   #use 0 if < 0
fsub     f4,f3,f1      #use max if > max
fsel     f2,f4,f2,f3
fctid[z] f2,f2         #convert to fx int
stfd     f2,disp(r1)   #store float
lwz      r3,disp+4(r1) #load word and zero
```

C.2.6 Conversion from Signed Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from signed fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
std      r3,disp(r1)   #store dword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp int
```

C.2.7 Conversion from Unsigned Fixed-Point Integer Doubleword to Floating-Point Number

The full *convert from unsigned fixed-point integer doubleword* function, using the rounding mode specified by $FPSCR_{RN}$, can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the value 2^{32} is in FPR 4, the result is returned in FPR 1, and two doublewords at displacement “disp” from the address in GPR 1 can be used as scratch space.

```
rldicl   r2,r3,32,32   #isolate high half
rldicl   r0,r3,0,32    #isolate low half
std      r2,disp(r1)   #store dword both
std      r0,disp+8(r1)
lfd      f2,disp(r1)   #load float both
lfd      f1,disp+8(r1)
fcfid    f2,f2         #convert each half to
fcfid    f1,f1         #  fp int (exact result)
fmadd    f1,f4,f2,f1   #(2**32)*high + low
```

An alternative, shorter, sequence can be used if rounding according to $FSCPR_{RN}$ is desired and $FPSCR_{RN}$ specifies *Round toward + Infinity* or *Round toward – Infinity*, or if it is acceptable for the rounded answer to be either of the two representable floating-point integers nearest to the given fixed-point integer. In this case the full *convert from unsigned fixed-point integer doubleword* function can be implemented with the sequence shown below, assuming the value 2^{64} is in FPR 2.

```
std      r3,disp(r1)   #store dword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp int
fadd     f4,f1,f2      #add 2**64
fsel     f1,f1,f1,f4   #  if r3 < 0
```

C.2.8 Conversion from Signed Fixed-Point Integer Word to Floating-Point Number

The full *convert from signed fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
extsw    r3,r3         #extend sign
std      r3,disp(r1)   #store dword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp int
```

C.2.9 Conversion from Unsigned Fixed-Point Integer Word to Floating-Point Number

The full *convert from unsigned fixed-point integer word* function can be implemented with the sequence shown below, assuming the fixed-point value to be converted is in GPR 3, the result is returned in FPR 1, and a doubleword at displacement “disp” from the address in GPR 1 can be used as scratch space. (The result is exact.)

```
rldicl   r0,r3,0,32    #zero-extend
std      r0,disp(r1)   #store dword
lfd      f1,disp(r1)   #load float
fcfid    f1,f1         #convert to fp int
```

C.3 Floating-Point Selection

This section gives examples of how the optional *Floating Select* instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using ***fsel*** and other PowerPC AS instructions. In the examples, *a, b, x, y,* and *z* are floating-point variables, which are assumed to be in FPRs *fa, fb, fx, fy*, and *fz*. FPR *fs* is assumed to be available for scratch space.

Additional examples can be found in Section C.2, “Floating-Point Conversions” on page 180.

Warning: Care must be taken in using ***fsel*** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities: see Section C.3.4, “Notes”.

C.3.1 Comparison to Zero

```
High-level language:      PowerPC AS:         Notes

if a ≥ 0.0 then x ← y     fsel  fx,fa,fy,fz    (1)
           else x ← z

if a > 0.0 then x ← y     fneg  fs,fa          (1,2)
           else x ← z     fsel  fx,fs,fz,fy

if a = 0.0 then x ← y     fsel  fx,fa,fy,fz    (1)
           else x ← z     fneg  fs,fa
                          fsel  fx,fs,fx,fz
```

C.3.2 Minimum and Maximum

```
High-level language:      PowerPC AS:         Notes

x ← min(a,b)              fsub  fs,fa,fb      (3,4,5)
                          fsel  fx,fs,fb,fa

x ← max(a,b)              fsub  fs,fa,fb      (3,4,5)
                          fsel  fx,fs,fa,fb
```

C.3.3 Simple if-then-else Constructions

```
High-level language:      PowerPC AS:         Notes

if a ≥ b then x ← y       fsub  fs,fa,fb       (4,5)
         else x ← z       fsel  fx,fs,fy,fz

if a > b then x ← y       fsub  fs,fb,fa      (3,4,5)
         else x ← z       fsel  fx,fs,fz,fy

if a = b then x ← y       fsub  fs,fa,fb       (4,5)
         else x ← z       fsel  fx,fs,fy,fz
                          fneg  fs,fs
                          fsel  fx,fs,fx,fz
```

C.3.4 Notes

The following Notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of ***fsel*** is contemplated.

In these Notes, the “optimized program” is the PowerPC AS program shown, and the “unoptimized program” (not shown) is the corresponding PowerPC AS program that uses ***fcmpu*** and *Branch Conditional* instructions instead of ***fsel***.

1. The unoptimized program affects the VXSNAN bit of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exception is enabled, while the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if *a* is a NaN.
3. The optimized program gives the incorrect result if *a* and/or *b* is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if *a* and *b* are infinities of the same sign. (Here it is assumed that Invalid Operation Exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if Invalid Operation Exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects the OX, UX, XX, and VXISI bits of the FPSCR, and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

Appendix D. Cross-Reference for Changed POWER Mnemonics

The following table lists the POWER instruction mnemonics that have been changed in the PowerPC AS 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 or Book in which the instruction is described, as well as the instruction names. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC AS User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*). If an instruction is defined in more than one of these Books, the lowest-numbered Book is used.

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 / Bk	POWER		PowerPC AS	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
61	a[o][.]	Add	addc[o][.]	Add Carrying
62	ae[o][.]	Add Extended	adde[o][.]	
60	ai	Add Immediate	addic	Add Immediate Carrying
60	ai.	Add Immediate and Record	addic.	Add Immediate Carrying and Record
62	ame[o][.]	Add To Minus One Extended	addme[o][.]	Add to Minus One Extended
78	andil.	AND Immediate Lower	andi.	AND Immediate
78	andiu.	AND Immediate Upper	andis.	AND Immediate Shifted
63	aze[o][.]	Add To Zero Extended	addze[o][.]	Add to Zero Extended
28	bcc[l]	Branch Conditional to Count Register	bcctr[l]	
28	bcr[l]	Branch Conditional to Link Register	bclr[l]	
59	cal	Compute Address Lower	addi	Add Immediate
59	cau	Compute Address Upper	addis	Add Immediate Shifted
60	cax[o][.]	Compute Address	add[o][.]	Add
83	cntlz[.]	Count Leading Zeros	cntlzw[.]	Count Leading Zeros Word
II	dclz	Data Cache Line Set to Zero	dcbz	Data Cache Block set to Zero
† II	dcs	Data Cache Synchronize	sync	Synchronize
82	exts[.]	Extend Sign	extsh[.]	Extend Sign Halfword
125	fa[.]	Floating Add	fadd[.]	
126	fd[.]	Floating Divide	fdiv[.]	
126	fm[.]	Floating Multiply	fmul[.]	
127	fma[.]	Floating Multiply-Add	fmadd[.]	
127	fms[.]	Floating Multiply-Subtract	fmsub[.]	
128	fnma[.]	Floating Negative Multiply-Add	fnmadd[.]	
128	fnms[.]	Floating Negative Multiply-Subtract	fnmsub[.]	
125	fs[.]	Floating Subtract	fsub[.]	
II	ics	Instruction Cache Synchronize	isync	Instruction Synchronize
40	l	Load	lwz	Load Word and Zero
49	lbrx	Load Byte-Reverse Indexed	lwbrx	Load Word Byte-Reverse Indexed
51	lm	Load Multiple	lmw	Load Multiple Word
54	lsi	Load String Immediate	lswi	Load String Word Immediate
55	lsx	Load String Indexed	lswx	Load String Word Indexed
40	lu	Load with Update	lwzu	Load Word and Zero with Update

Page / Bk	POWER		PowerPC AS	
	Mnemonic	Instruction	Mnemonic	Instruction
40	lux	Load with Update Indexed	lwzux	Load Word and Zero with Update Indexed
40	lx	Load Indexed	lwzx	Load Word and Zero Indexed
III	mtsri	Move To Segment Register Indirect	mtsrin	
64	muli	Multiply Immediate	mulli	Multiply Low Immediate
64	muls[o][.]	Multiply Short	mullw[o][.]	Multiply Low Word
79	oril	OR Immediate Lower	ori	OR Immediate
79	oriu	OR Immediate Upper	oris	OR Immediate Shifted
III	rfsvc	Return From SVC	rfscv	Return From System Call Vectored
89	rlimi[.]	Rotate Left Immediate Then Mask Insert	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
86	rlinm[.]	Rotate Left Immediate Then AND With Mask	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
88	rlnm[.]	Rotate Left Then AND With Mask	rlwnm[.]	Rotate Left Word then AND with Mask
61	sf[o][.]	Subtract From	subfc[o][.]	Subtract From Carrying
62	sfe[o][.]	Subtract From Extended	subfe[o][.]	
61	sfi	Subtract From Immediate	subfic	Subtract From Immediate Carrying
62	sfme[o][.]	Subtract From Minus One Extended	subfme[o][.]	
63	sfze[o][.]	Subtract From Zero Extended	subfze[o][.]	
90	sl[.]	Shift Left	slw[.]	Shift Left Word
91	sr[.]	Shift Right	srw[.]	Shift Right Word
93	sra[.]	Shift Right Algebraic	sraw[.]	Shift Right Algebraic Word
92	srai[.]	Shift Right Algebraic Immediate	srawi[.]	Shift Right Algebraic Word Imme-diate
46	st	Store	stw	Store Word
50	stbrx	Store Byte-Reverse Indexed	stwbrx	Store Word Byte-Reverse Indexed
52	stm	Store Multiple	stmw	Store Multiple Word
56	stsi	Store String Immediate	stswi	Store String Word Immediate
57	stsx	Store String Indexed	stswx	Store String Word Indexed
46	stu	Store with Update	stwu	Store Word with Update
46	stux	Store with Update Indexed	stwux	Store Word with Update Indexed
46	stx	Store Indexed	stwx	Store Word Indexed
29	svca	Supervisor Call	sc	System Call
29	svcl	Supervisor Call	scv	System Call Vectored
73	t	Trap	tw	Trap Word
72	ti	Trap Immediate	twi	Trap Word Immediate
III	tlbi	TLB Invalidate Entry	tlbie	
79	xoril	XOR Immediate Lower	xori	XOR Immediate
79	xoriu	XOR Immediate Upper	xoris	XOR Immediate Shifted

Appendix E. Incompatibilities with the POWER Architecture

This appendix identifies the known incompatibilities that must be managed in the migration from the POWER Architecture to the PowerPC AS Architecture. Some of the incompatibilities can, at least in principle, be detected by the processor, which could trap and let software simulate the POWER operation. Others cannot be detected by the processor even in principle.

In general, the incompatibilities identified here are those that affect a POWER application program; incompatibilities for instructions that can be used only by POWER system programs are not necessarily discussed.

References to instructions and facilities that are not defined in Book I, *PowerPC AS User Instruction Set Architecture*, apply to an implementation that conforms to Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*.

E.1 New Instructions, Formerly Privileged Instructions

Instructions new to PowerPC AS typically use opcode values (including extended opcode) that are illegal in POWER. A few instructions that are privileged in POWER (e.g., ***dclz***, called ***dcbz*** in PowerPC AS) have been made nonprivileged in PowerPC AS. Any POWER program that executes one of these now-valid or now-nonprivileged instructions, expecting to cause the system illegal instruction error handler or the system privileged instruction error handler to be invoked, will not execute correctly on PowerPC AS.

E.2 Newly Privileged Instructions

The following instructions are nonprivileged in POWER but privileged in PowerPC AS.

mfmsr
mfsr

E.3 Reserved Bits in Instructions

These are shown with "/"s in the instruction layouts. In POWER such bits are ignored by the processor. In PowerPC AS they must be 0 or the instruction form is invalid.

In several cases the PowerPC AS Architecture assumes that such bits in POWER instructions are indeed 0. The cases include the following.

- ***bclr***[***l***] and ***bcctr***[***l***] assume that bits 19:20 in the POWER instructions are 0.
- ***cmpi, cmp, cmpli***, and ***cmpl*** assume that bit 10 in the POWER instructions is 0.
- ***mtspr*** and ***mfspr*** assume that bits 16:20 in the POWER instructions are 0.
- ***mtcrf*** and ***mfcr*** assume that bit 11 in the POWER instructions is 0.
- ***sync*** assumes that bit 10 in the POWER instruction (***dcs***) is 0.

E.4 Reserved Bits in Registers

Both POWER and PowerPC AS permit software to write any value to these bits. However in POWER reading such a bit always returns 0, while in PowerPC AS reading it may return either 0 or the value that was last written to it.

E.5 Alignment Check

The POWER MSR AL bit (bit 24) is no longer supported; the corresponding PowerPC AS MSR bit, bit 56, is the US bit in *tags active* mode and is treated as reserved in *tags inactive* mode. The low-order bits of the EA are always used. (Notice that the value 0 — the normal value for a reserved bit — means "ignore the low-order EA bits" in POWER, and the value 1 means "use the low-order EA bits".) POWER-compatible operating system code will probably write the value 1 to this bit.

E.6 Condition Register

The following instructions specify a field in the CR explicitly (via the BF field) and also, in POWER, use bit 31 as the Record bit. In PowerPC AS, if bit 31 = 1 for these instructions the instruction form is invalid. In POWER, if Rc=1 the instructions execute normally except as follows:

cmp	CR0 is undefined if Rc=1 and BF≠0
cmpl	CR0 is undefined if Rc=1 and BF≠0
mcrxr	CR0 is undefined if Rc=1 and BF≠0
fcmpu	CR1 is undefined if Rc=1
fcmpo	CR1 is undefined if Rc=1
mcrfs	CR1 is undefined if Rc=1 and BF≠1

E.7 Inappropriate Use of LK and Rc Bits

For the instructions listed below, if bit 31 (LK or Rc bit in POWER) is set to 1, POWER executes the instruction normally with the exception of setting the Link Register (if LK=1) or Condition Register Field 0 or 1 (if Rc=1) to an undefined value. In PowerPC AS such instruction forms are invalid.

PowerPC AS instructions that are invalid form if bit 31 = 1 (LK bit in POWER):

sc (***svc*** in POWER)
the *Condition Register Logical* instructions
mcrf
isync (***ics*** in POWER)

PowerPC AS instructions that are invalid form if bit 31 = 1 (Rc bit in POWER):

fixed-point X-form *Load* and *Store* instructions
fixed-point X-form *Compare* instructions
the X-form *Trap* instruction
mtspr, mfspr, mtcrf, mcrxr, mfcr
floating-point X-form *Load* and *Store* instructions
floating-point *Compare* instructions
mcrfs
dcbz (***dclz*** in POWER)

E.8 BO Field

POWER shows certain bits in the BO field — used by *Branch Conditional* instructions — as “x”. Although the POWER Architecture does not say how these bits are to be interpreted, they are in fact ignored by the processor.

PowerPC AS shows these bits as “z”, “a”, or “t”. The “z” bits are ignored, as in POWER. However, the “a” and “t” bits can be used by software to provide a hint about how the branch is likely to behave. If a POWER program has the “wrong” value for these bits, the program will produce the same results as on POWER but performance may be affected.

E.9 BH Field

Bits 19:20 of the *Branch Conditional to Link Register* and *Branch Conditional to Count Register* instructions are reserved in POWER but are defined as a branch hint (BH) field in PowerPC AS. Because these bits are hints, they may affect performance but do not affect the results of executing the instruction.

E.10 Branch Conditional to Count Register

For the case in which the Count Register is decremented and tested (i.e., the case in which BO_2=0), POWER specifies only that the branch target address is undefined, with the implication that the Count Register, and the Link Register if LK=1, are updated in the normal way. PowerPC AS specifies that this instruction form is invalid.

E.11 System Call

There are several respects in which PowerPC AS is incompatible with POWER for *System Call* instructions — which in POWER are called *Supervisor Call* instructions.

- POWER provides a version of the *Supervisor Call* instruction (bits 30:31 = 0b00) that allows instruction fetching to continue at any one of 128 locations without altering the Link Register. PowerPC AS provides no such version: if bits 30:31 of the instruction are 0b00 the instruction form is invalid.
- POWER provides a version of the *Supervisor Call* instruction (bits 30:31 = 0b11) that resumes instruction fetching at one location and sets the Link Register to the address of the next instruction. PowerPC AS provides no such version: if bits 30:31 of the instruction are 0b11 the instruction form is invalid.
- For POWER, information from the MSR is saved in the Count Register. For PowerPC AS this information is saved in SRR1 for the *System Call* instruction (the *System Call Vectored* instruction is compatible with POWER in this regard).

- POWER permits bits 16:19 and 27:29 of the instruction to be nonzero, while in PowerPC AS such an instruction form is invalid.

Architecture and Engineering Note

Bits 16:19 and 27:29 should be regarded as reserved for POWER. As long as POWER compatibility is required for this instruction, bits 16:19 and 27:29 should be ignored by the processor.

- POWER saves the low-order 16 bits of the instruction, in the Count Register. PowerPC AS does not save them.
- The settings of MSR bits by the associated interrupt differ between POWER and PowerPC AS; see *POWER Processor Architecture* and Book III, *PowerPC AS Operating Environment Architecture*.

E.12 Fixed-Point Exception Register (XER)

Bits 48:55 of the XER are reserved in PowerPC AS, while in POWER the corresponding bits (16:23) are defined and contain the comparison byte for the ***lscbx*** instruction (which PowerPC AS lacks).

Engineering Note

For reasons of compatibility with the POWER Architecture, early implementations must set XER bits 48:55 from the source value on write, and return the value last written to them on read.

E.13 Update Forms of Storage Access Instructions

PowerPC AS requires that RA not be equal to either RT (fixed-point *Load* only) or 0. If the restriction is violated the instruction form is invalid. POWER permits these cases, and simply avoids saving the EA.

E.14 Multiple Register Loads

PowerPC AS requires that RA, and RB if present in the instruction format, not be in the range of registers to be loaded, while POWER permits this and does not alter RA or RB in this case. (The PowerPC AS restriction applies even if RA=0, although there is no obvious benefit to the restriction in this case since RA is not used to compute the effective address if RA=0.) If the PowerPC AS restriction is violated, either the system illegal instruction error handler is invoked or the results are boundedly undefined. The instructions affected are:

lmw (***lm*** in POWER)
lswi (***lsi*** in POWER)
lswx (***lsx*** in POWER)

For example, an ***lmw*** instruction that loads all 32 registers is valid in POWER but is an invalid form in PowerPC AS.

E.15 Load/Store Multiple Instructions

There are several respects in which PowerPC AS is incompatible with POWER for *Load Multiple* and *Store Multiple* instructions.

- If the EA is not word-aligned, in PowerPC AS either an Alignment interrupt occurs or the results are boundedly undefined, while in POWER an Alignment interrupt occurs if $MSR_{AL}=1$ (the low-order two bits of the EA are ignored if $MSR_{AL}=0$).

Engineering Note

If attempt is made to execute an ***lmw*** or ***stmw*** instruction having an incorrectly aligned effective address, early implementations must either correctly transfer the addressed bytes or cause an Alignment interrupt, for reasons of compatibility with the POWER Architecture.

- In PowerPC AS the instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

E.16 Move Assist Instructions

There are several respects in which PowerPC AS is incompatible with POWER for *Move Assist* instructions.

- In PowerPC AS an ***lswx*** instruction with zero length leaves the contents of RT undefined (if $RT\neq RA$ and $RT\neq RB$) or is an invalid instruction form (if RT=RA or RT=RB), while in POWER the corresponding instruction (***lsx***) is a no-op in these cases.
- In PowerPC AS an ***lswx*** instruction with zero length may alter the Reference bit, and a ***stswx*** instruction with zero length may alter the Reference and Change bits, while in POWER the corresponding instructions (***lsx*** and ***stsx***) do not alter the Reference and Change bits in this case.

| ■ In PowerPC AS a *Move Assist* instruction may be interrupted by a system-caused interrupt, while in POWER the instruction cannot be thus interrupted.

E.17 Move To/From SPR

There are several respects in which PowerPC AS is incompatible with POWER for *Move To/From Special Purpose Register* instructions.

- The SPR field is ten bits long in PowerPC AS, but only five in POWER (see also Section E.3, “Reserved Bits in Instructions” on page 185).
- ***mfspr*** can be used to read the Decrementer in problem state in POWER, but only in privileged state in PowerPC AS.
- If the SPR value specified in the instruction is not one of the defined values, POWER behaves as follows.
  - — If the instruction is executed in problem state and $SPR_0=1$, a Privileged Instruction type Program interrupt occurs. No architected registers are altered except those set by the interrupt.
  - † — Otherwise no architected registers are altered.

  In this same case, PowerPC AS behaves as follows.
  - — If the instruction is executed in problem state and $spr_0=1$, either an Illegal Instruction type Program interrupt or a Privileged Instruction type Program interrupt occurs. No architected registers are altered except those set by the interrupt.
  - † — Otherwise either an Illegal Instruction type Program interrupt occurs (in which case no architected registers are altered except those set by the interrupt) or the results are | boundedly undefined (or possibly undefined, | for ***mtspr***; see Book III).

Engineering Note

For reasons of compatibility with the POWER Architecture, early implementations must cause an Illegal Instruction type Program interrupt for an attempt to execute an ***mtspr*** or ***mfspr*** instruction with SPR=0 ($spr_{0:4}=0$ denotes the POWER MQ register).

Similarly, early implementations must cause an Illegal Instruction type Program interrupt for an attempt to execute an ***mfspr*** instruction with SPR=4 ($spr_{0:4}=4$ denotes reading the Real-Time Clock Upper in POWER), SPR=5 ($spr_{0:4}=5$ denotes reading the Real-Time Clock Lower in POWER), or SPR=6 ($spr_{0:4}=6$ denotes reading the Decrementer in POWER).

Essentially all POWER programs are expected to have bits 16:20 of ***mtspr*** and ***mfspr*** instructions set to 0. These bits correspond to PowerPC AS's $spr_{5:9}$, and are reserved bits in POWER. The requirements described in this Note provide compatibility only for POWER programs that have these bits set to 0.

E.18 Effects of Exceptions on FPSCR Bits FR and FI

For the following cases, POWER does not specify how FR and FI are set, while PowerPC AS preserves them for Invalid Operation Exception caused by a *Compare* instruction, sets FI to 1 and FR to an undefined value for disabled Overflow Exception, and clears them otherwise.

- Invalid Operation Exception (enabled or disabled)
- Zero Divide Exception (enabled or disabled)
- Disabled Overflow Exception

E.19 Store Floating-Point Single Instructions

There are several respects in which PowerPC AS is incompatible with POWER for *Store Floating-Point Single* instructions.

- POWER uses $FPSCR_{UE}$ to help determine whether denormalization should be done, while PowerPC AS does not. Using $FPSCR_{UE}$ is in fact incorrect: if $FPSCR_{UE}=1$ and a denormalized single-precision number is copied from one storage location to another by means of ***lfs*** followed by ***stfs***, the two “copies” may not be the same.
- For an operand having an exponent that is less than 874 (unbiased exponent less than -149),

POWER stores a zero (if $FPSCR_{UE}=0$) while PowerPC AS stores an undefined value.

E.20 Move From FPSCR

POWER defines the high-order 32 bits of the result of ***mffs*** to be 0xFFFF_FFFF, while PowerPC AS specifies that they are undefined.

E.21 Zeroing Bytes in the Data Cache

The ***dclz*** instruction of POWER and the ***dcbz*** instruction of PowerPC AS have the same opcode. However, the functions differ in the following respects.

- ***dclz*** clears a line while ***dcbz*** clears a block.
- ***dclz*** saves the EA in RA (if RA≠0) while ***dcbz*** does not.
- ***dclz*** is privileged while ***dcbz*** is not.

E.22 Synchronization

The ***sync*** instruction (called ***dcs*** in POWER) and the ***isync*** instruction (called ***ics*** in POWER) cause more pervasive synchronization in PowerPC AS than in POWER. However, unlike ***dcs***, ***sync*** does not wait until data cache block writes caused by preceding instructions have been performed in main storage. Also, ***sync*** has an L field while ***dcs*** does not.

E.23 Direct-Store Segments

POWER's direct-store segments are not supported in PowerPC AS.

E.24 Segment Register Manipulation Instructions

The definitions of the four *Segment Register Manipulation* instructions ***mtsr, mtsrin, mfsr,*** and ***mfsrin*** differ in two respects between POWER and PowerPC AS. Instructions similar to ***mtsrin*** and ***mfsrin*** are called ***mtsri*** and ***mfsri*** in POWER.

privilege:	***mfsr*** and ***mfsri*** are problem state instructions in POWER, while ***mfsr*** and ***mfsrin*** are privileged in PowerPC AS.
function:	the “indirect” instructions (***mtsri*** and ***mfsri***) in POWER use an RA register in computing the Segment Register number, and the computed EA is stored into RA (if RA≠0 and RA≠RT), while in PowerPC AS ***mtsrin*** and ***mfsrin*** have no RA field and the EA is not stored.

mtsr, mtsrin (mtsri), and ***mfsr*** have the same opcodes in PowerPC AS as in POWER. ***mfsri*** (POWER) and ***mfsrin*** (PowerPC AS) have different opcodes.

Also, the *Segment Register Manipulation* instructions are required in POWER whereas they are optional in PowerPC AS.

E.25 TLB Entry Invalidation

The ***tlbi*** instruction of POWER and the ***tlbie*** instruction of PowerPC AS have the same opcode. However, the functions differ in the following respects.

- ***tlbi*** computes the EA as (RA|0) + (RB), while ***tlbie*** lacks an RA field and computes the EA and related information as (RB).
- ***tlbi*** saves the EA in RA (if RA≠0), while ***tlbie*** lacks an RA field and does not save the EA.
- For ***tlbi*** the high-order 36 bits of RB are used in computing the EA, while for ***tlbie*** these bits contain additional information that is not directly related to the EA.
- ***tlbie*** has an L field, while ***tlbi*** does not.

Also, ***tlbi*** is required in POWER whereas ***tlbie*** is optional in PowerPC AS.

E.26 Alignment Interrupts

Placing information about the interrupting instruction into the DSISR and the DAR when an Alignment interrupt occurs is optional in PowerPC AS but required in POWER.

E.27 Floating-Point Interrupts

Both architectures use MSR bit 20 to control the generation of interrupts for floating-point enabled exceptions. However, in PowerPC AS this bit is part of a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER this bit is used independently to control the occurrence of the interrupt (in POWER all floating-point interrupts are precise).

E.28 Timing Facilities

E.28.1 Real-Time Clock

The POWER Real-Time Clock is not supported in PowerPC AS. Instead, PowerPC AS provides a Time Base. Both the RTC and the TB are 64-bit Special Purpose Registers, but they differ in the following respects.

- The RTC counts seconds and nanoseconds, while the TB counts “ticks”. The ticking rate of the TB is implementation-dependent.
- The RTC increments discontinuously: 1 is added to RTCU when the value in RTCL passes 999_999_999. The TB increments continuously: 1 is added to TBU when the value in TBL passes 0xFFFF_FFFF.
- The RTC is written and read by the ***mtspr*** and ***mfspr*** instructions, using SPR numbers that denote the RTCU and RTCL. The TB is written by the ***mtspr*** instruction (using new SPR numbers), and read by the new ***mftb*** instruction.
- The SPR numbers that denote POWER's RTCL and RTCU are invalid in PowerPC AS.
- The RTC is guaranteed to increment at least once in the time required to execute ten *Add Immediate* instructions. No analogous guarantee is made for the TB.
- Not all bits of RTCL need be implemented, while all bits of the TB must be implemented.

E.28.2 Decrementer

The PowerPC AS Decrementer differs from the POWER Decrementer in the following respects.

- The PowerPC AS DEC decrements at the same rate that the TB increments, while the POWER DEC decrements every nanosecond (which is the same rate that the RTC increments).
- Not all bits of the POWER DEC need be implemented, while all bits of the PowerPC AS DEC must be implemented.
- The interrupt caused by the DEC has its own interrupt vector location in PowerPC AS, but is considered an External interrupt in POWER.

E.29 Deleted Instructions

The following instructions are part of the POWER Architecture but have been dropped from the PowerPC AS Architecture.

abs	Absolute
clcs	Cache Line Compute Size
clf	Cache Line Flush
cli (*)	Cache Line Invalidate
dclst	Data Cache Line Store
div	Divide
divs	Divide Short
doz	Difference Or Zero
dozi	Difference Or Zero Immediate
lscbx	Load String And Compare Byte Indexed
maskg	Mask Generate
maskir	Mask Insert From Register
mfsri	Move From Segment Register Indirect
mul	Multiply
nabs	Negative Absolute
rac (*)	Real Address Compute
rfi (*)	Return From Interrupt
rlmi	Rotate Left Then Mask Insert
rrib	Rotate Right And Insert Bit
sle	Shift Left Extended
sleq	Shift Left Extended With MQ
sliq	Shift Left Immediate With MQ
slliq	Shift Left Long Immediate With MQ
sllq	Shift Left Long With MQ
slq	Shift Left With MQ
sraiq	Shift Right Algebraic Immediate With MQ
sraq	Shift Right Algebraic With MQ
sre	Shift Right Extended
srea	Shift Right Extended Algebraic
sreq	Shift Right Extended With MQ
sriq	Shift Right Immediate With MQ
srliq	Shift Right Long Immediate With MQ
srlq	Shift Right Long With MQ
srq	Shift Right With MQ

(*) This instruction is privileged.

Note: Many of these instructions use the MQ register. The MQ is not defined in the PowerPC AS Architecture.

E.30 Discontinued Opcodes

The opcodes listed below are defined in the POWER Architecture but have been dropped from the PowerPC AS Architecture. The list contains the POWER mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are reserved in PowerPC AS.

MNEM	PRI	XOP
abs	31	360
clcs	31	531
clf	31	118
cli (*)	31	502
dclst	31	630
div	31	331
divs	31	363
doz	31	264
dozi	09	–
lscbx	31	277
maskg	31	29
maskir	31	541
mfsri	31	627
mul	31	107
nabs	31	488
rac (*)	31	818
rfi (*)	19	50
rlmi	22	–
rrib	31	537
sle	31	153
sleq	31	217
sliq	31	184
slliq	31	248
sllq	31	216
slq	31	152
sraiq	31	952
sraq	31	920
sre	31	665
srea	31	921
sreq	31	729
sriq	31	696
srliq	31	760
srlq	31	728
srq	31	664

(*) This instruction is privileged.

Assembler Note

It might be helpful to current software writers for the Assembler to flag the discontinued POWER instructions.

Engineering Note

The instructions listed above are reserved in the PowerPC AS Architecture. For reasons of compatibility with the POWER Architecture, early implementations must cause an Illegal Instruction type Program interrupt for an attempt to execute any of these instructions that are not privileged.

E.31 POWER2 Compatibility

The POWER2 instruction set is a superset of the POWER instruction set. Some of the instructions added for POWER2 are included in the PowerPC AS Architecture. Those that have been renamed in the PowerPC AS Architecture are listed in this section, as are the new POWER2 instructions that are not included in the PowerPC AS Architecture.

Other incompatibilities are also listed.

E.31.1 Cross-Reference for Changed POWER2 Mnemonics

The following table lists the new POWER2 instruction mnemonics that have been changed in the PowerPC AS User Instruction Set Architecture, sorted by POWER2 mnemonic.

To determine the PowerPC AS mnemonic for one of these POWER2 mnemonics, find the POWER2 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.

POWER2 mnemonics that have not changed are not listed.

Page	POWER2		PowerPC AS	
	Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
131	fcir[.]	Floating Convert Double to Integer with Round	fctiw[.]	Floating Convert To Integer Word
131	fcirz[.]	Floating Convert Double to Integer with Round to Zero	fctiwz[.]	Floating Convert To Integer Word with round toward Zero

E.31.2 Floating-Point Conversion to Integer

The ***fcir*** and ***fcirz*** instructions of POWER2 have the same opcodes as do the ***fctiw*** and ***fctiwz*** instructions, respectively, of PowerPC AS. However, the functions differ in the following respects.

- ***fcir*** and ***fcirz*** set the high-order 32 bits of the target FPR to 0xFFFF_FFFF, while ***fctiw*** and ***fctiwz*** set them to an undefined value.
- Except for enabled Invalid Operation Exceptions, ***fcir*** and ***fcirz*** set the FPRF field of the FPSCR based on the result, while ***fctiw*** and ***fctiwz*** set it to an undefined value.
- ***fcir*** and ***fcirz*** do not affect the VXSNAN bit of the FPSCR, while ***fctiw*** and ***fctiwz*** do.
- ***fcir*** and ***fcirz*** set $FPSCR_{XX}$ to 1 for certain cases of "Large Operands" (i.e., operands that are too large to be represented as a 32-bit signed fixed-point integer), while ***fctiw*** and ***fctiwz*** do not alter it for any case of "Large Operand". (The IEEE standard requires not altering it for "Large Operands".)

E.31.3 Storage Access Ordering

| POWER2 uses MSR bit 28 to control storage access
| ordering. This bit is reserved in PowerPC AS, and no
| corresponding control is provided.

E.31.4 Floating-Point Interrupts

Both architectures use MSR bits 20 and 23 to control the generation of interrupts for floating-point enabled exceptions. However, in PowerPC AS these bits comprise a two-bit value that controls the occurrence, precision, and recoverability of the interrupt, while in POWER2 these bits are used independently to control the occurrence (bit 20) and the precision (bit 23) of the interrupt. Moreover, in PowerPC AS all floating-point interrupts are considered Program interrupts, while in POWER2 imprecise floating-point interrupts have their own interrupt vector location.

E.31.5 Trace

The Trace interrupt vector location differs between the two architectures, and there are many other differences. Also, the Trace facility is optional in PowerPC AS but required in POWER2.

E.31.6 Deleted Instructions

The following instructions are new in POWER2 implementations of the POWER Architecture but have been dropped from the PowerPC AS Architecture.

lfq	Load Floating-Point Quad
lfqu	Load Floating-Point Quad with Update
lfqux	Load Floating-Point Quad with Update Indexed
lfqx	Load Floating-Point Quad Indexed
stfq	Store Floating-Point Quad
stfqu	Store Floating-Point Quad with Update
stfqux	Store Floating-Point Quad with Update Indexed
stfqx	Store Floating-Point Quad Indexed

E.31.7 Discontinued Opcodes

The opcodes listed below are new in POWER2 implementations of the POWER Architecture but have been dropped from the PowerPC AS Architecture. The list contains the POWER2 mnemonic (MNEM), the primary opcode (PRI), and the extended opcode (XOP) if appropriate. The corresponding instructions are reserved in PowerPC AS.

MNEM	PRI	XOP
lfq	56	–
lfqu	57	–
lfqux	31	823
lfqx	31	791
stfq	60	–
stfqu	61	–
stfqux	31	951
stfqx	31	919

Engineering Note

The instructions listed above are reserved in the PowerPC AS Architecture. For reasons of compatibility with POWER2 implementations of the POWER Architecture, early implementations must cause an Illegal Instruction type Program interrupt for an attempt to execute any of these instructions.

†

Appendix F. New Instructions

The following instructions in the PowerPC AS User Instruction Set Architecture are new; they are not in the POWER Architecture.

The following instructions are optional: ***fres, frsqrte, fsel, fsqrt*[*s*]**.

cmpla	Compare Logical Addresses
cntlzd	Count Leading Zeros Doubleword
divd	Divide Doubleword
divdu	Divide Doubleword Unsigned
divw	Divide Word
divwu	Divide Word Unsigned
dsixes	Decimal Sixes
dtcs	Decimal Test and Clear Sign
extsb	Extend Sign Byte
extsw	Extend Sign Word
fadds	Floating Add Single
fcfid	Floating Convert From Integer Doubleword
fctid	Floating Convert To Integer Doubleword
fctidz	Floating Convert To Integer Doubleword with round toward Zero
fctiw	Floating Convert To Integer Word
fctiwz	Floating Convert To Integer Word with round toward Zero
fdivs	Floating Divide Single
fmadds	Floating Multiply-Add Single
fmsubs	Floating Multiply-Subtract Single
fmuls	Floating Multiply Single
fnmadds	Floating Negative Multiply-Add Single
fnmsubs	Floating Negative Multiply-Subtract Single
fres	Floating Reciprocal Estimate Single
frsqrte	Floating Reciprocal Square Root Estimate
fsel	Floating Select
***fsqrt*[*s*]**	Floating Square Root [Single]
fsubs	Floating Subtract Single
ld	Load Doubleword
†	
ldu	Load Doubleword with Update
ldux	Load Doubleword with Update Indexed
ldx	Load Doubleword Indexed
lmd	Load Multiple Doubleword
lq	Load Quadword
lsdi	Load String Doubleword Immediate
lsdx	Load String Doubleword Indexed
lwa	Load Word Algebraic
†	
lwaux	Load Word Algebraic with Update Indexed
lwax	Load Word Algebraic Indexed
mcrxrt	Move to Condition Register from XER TGCC
mulhd	Multiply High Doubleword
mulhdu	Multiply High Doubleword Unsigned
mulhw	Multiply High Word
mulhwu	Multiply High Word Unsigned
mulld	Multiply Low Doubleword
rldcl	Rotate Left Doubleword then Clear Left
rldcr	Rotate Left Doubleword then Clear Right
rldic	Rotate Left Doubleword Immediate then Clear
rldicl	Rotate Left Doubleword Immediate then Clear Left
rldicr	Rotate Left Doubleword Immediate then Clear Right
rldimi	Rotate Left Doubleword Immediate then Mask Insert
selii	Select Immediate-Immediate
selir	Select Immediate-Register
selri	Select Register-Immediate
selrr	Select Register-Register
settag	Set XER TAG
sld	Shift Left Doubleword
srad	Shift Right Algebraic Doubleword
sradi	Shift Right Algebraic Doubleword Immediate
srd	Shift Right Doubleword
std	Store Doubleword
†	
stdu	Store Doubleword with Update
stdux	Store Doubleword with Update Indexed
stdx	Store Doubleword Indexed
stfiwx	Store Floating-Point as Integer Word Indexed
stmd	Store Multiple Doubleword
stq	Store Quadword
stsdi	Store String Doubleword Immediate
stsdx	Store String Doubleword Indexed
†	
subf	Subtract From
td	Trap Doubleword
tdi	Trap Doubleword Immediate
txer	Trap on XER

Appendix G. Illegal Instructions

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the PowerPC AS Architecture; that is, some future version of the PowerPC AS Architecture may define any of these instructions to perform new functions.

The following primary opcodes are illegal.

1, 4, 5, 6

The following primary opcodes are illegal in *tags inactive* mode.

56

The following primary opcodes have unused extended opcodes. Their unused extended opcodes can be determined from the opcode maps in Appendix I. All unused extended opcodes are illegal.

19, 30, 31, 56, 57, 59, 60, 61, 62, 63

An instruction consisting entirely of binary 0s is illegal, and is guaranteed to be illegal in all future versions of this architecture. |

The following instructions are illegal in *tags inactive* mode:

- ***cmpla***
- ***dsixes***
- ***dtcs.***
- ***lmd***
- ***lq***
- ***lsdi***
- ***lsdx***
- ***mcrxrt***
- ***rfscv***
- ***scv***
- ***selii***[**.**]
- ***selir***[**.**]
- ***selri***[**.**]
- ***selrr***[**.**]
- ***settag***
- ***stmd***
- ***stq***
- ***stsdi***
- ***stsdx***
- ***txer***

Appendix H. Reserved Instructions

The instructions in this class are allocated to specific purposes that are outside the scope of the PowerPC AS User Instruction Set Architecture, PowerPC AS Virtual Environment Architecture, and PowerPC AS Operating Environment Architecture.

The following types of instruction are included in this class.

1. The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction: see Section 1.8.2, “Illegal Instruction Class” on page 13) and the extended opcodes shown below.

   256 Service Processor “Attention” (PowerPC AS only)

   257 ***bccbr*** (PowerPC AS only)

2. Instructions for the POWER Architecture that have not been included in the PowerPC AS Architecture. These are listed in Section E.30, “Discontinued Opcodes” on page 191 and Section E.31.7, “Discontinued Opcodes” on page 193.
3. Implementation-specific instructions used to conform to the PowerPC AS Architecture specification.
4. Any other instructions contained in Book IV, *PowerPC AS Implementation Features* for any implementation, that are not defined in the PowerPC AS User Instruction Set Architecture, PowerPC AS Virtual Environment Architecture, or PowerPC AS Operating Environment Architecture.

Appendix I. Opcode Maps

This section contains tables showing the opcodes and extended opcodes in all members of the POWER architecture family.

For the primary opcode table (Table 14 on page 203), each cell is in the following format.

Opcode in Decimal		Opcode in Hexadecimal
	Instruction Mnemonic	
Applicable Machines		Instruction Format

"Applicable Machines" identifies the POWER architecture family members that recognize the opcode, encoded as follows:

A	PowerPC AS
At	PowerPC AS in *tags active* mode only
Api	PowerPC AS in PowerPC-incompatible mode
P	PowerPC
2	POWER2
O	Original POWER (RS/6000)
All	All of the above

The extended opcode tables show the extended opcode in decimal, the instruction mnemonic, the applicable machines, and the instruction format. These tables appear in order of primary opcode within two groups. The first group consists of the primary opcodes that have small extended opcode fields (2-4 bits), namely 30, 56, 57, 58, 60, 61, and 62. The second group consists of primary opcodes that have 10-bit extended opcode fields. The tables for the second group are rotated.

In the extended opcode tables several special markings are used.

- A prime (′) following an instruction mnemonic denotes an additional cell, after the lowest-numbered one, used by the instruction. For example, ***subfc*** occupies cells 8 and 520 of primary opcode 31, with the former corresponding to OE=0 and the latter to OE=1. Similarly, ***sradi*** occupies cells 826 and 827, with the former corresponding to sh_5=0 and the latter to sh_5=1 (the 9-bit extended opcode 413, shown on page 92, excludes the sh_5 bit).
- Two vertical bars (||) are used instead of primed mnemonics when an instruction occupies an entire column of a table. The instruction mnemonic is repeated in the last cell of the column.
- For primary opcode 31, an asterisk (*) in a cell that would otherwise be empty means that the cell is reserved because it is "overlaid", by a fixed-point or *Storage Access* instruction having only a primary opcode, by an instruction having an extended opcode in primary opcode 30, 58, or 62, or by a potential instruction in any of the categories just mentioned. The overlaying instruction, if any, is also shown. A cell thus reserved should not be assigned to an instruction having primary opcode 31. (The overlaying is a consequence of opcode decoding for fixed-point instructions: the primary opcode, and the extended opcode if any, are mapped internally to a 10-bit "compressed opcode" for ease of subsequent decoding.)
- | Parentheses around the opcode or extended opcode mean that the instruction was defined in earlier versions of the PowerPC AS Architecture but is no longer defined in the PowerPC AS Architecture.

| An empty cell, a cell containing only an asterisk, or a cell in which the opcode or extended opcode is parenthesized, corresponds to an illegal instruction.

When instruction names and/or mnemonics differ among the family members, the PowerPC AS/PowerPC terminology is used.

The instruction consisting entirely of binary 0's causes the system illegal instruction error handler to be invoked for all members of the POWER family, and this is likely to remain true in future models (it is guaranteed in the PowerPC AS Architecture). An instruction having primary opcode 0 but not consisting entirely of binary 0's is reserved except for the following extended opcodes (instruction bits 21:30).

256	Service Processor "Attention" (PowerPC AS only)
257	***bccbr*** (PowerPC AS only)

Engineering Note

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

Architecture Note

The following opcodes are reserved because they are used in some implementations.

- Primary opcode 19: extended opcode 51
- Primary opcode 31: extended opcodes 131, 163, 262, 274, 308, 323, 451, 454, 486, 914, 946, 966, 978, 998, 1010

These opcodes will not be assigned a meaning in the PowerPC AS Architecture except after careful consideration of the effect of such assignment on existing implementations. The same applies to opcodes that are parenthesized in the opcode maps.

Table 14. Primary opcodes

0 00 Illegal, Reserved All	1 01	2 02 tdi AP D	3 03 twi All D	See primary opcode 0 extensions on page 201. Trap Doubleword Immediate Trap Word Immediate
4 04	5 05	6 06	7 07 mulli All D	Multiply Low Immediate
8 08 subfic All D	9 09 dozi 2O D	10 0A cmpli All D	11 0B cmpi All D	Subtract From Immediate Carrying Difference or Zero Immediate Compare Logical Immediate Compare Immediate
12 0C addic All D	13 0D addic. All D	14 0E addi All D	15 0F addis All D	Add Immediate Carrying Add Immediate Carrying and Record Add Immediate Add Immediate Shifted
16 10 bc All B	17 11 sc, scv All SC	18 12 b All I	19 13 CR ops, etc. All XL	Branch Conditional System Call (All), System Call Vectored (A2O) Branch See Table 22 on page 206
20 14 rlwimi All M	21 15 rlwinm All M	22 16 rlmi 2O M	23 17 rlwnm All M	Rotate Left Word Imm. then Mask Insert Rotate Left Word Imm. then AND with Mask Rotate Left then Mask Insert Rotate Left Word then AND with Mask
24 18 ori All D	25 19 oris All D	26 1A xori All D	27 1B xoris All D	OR Immediate OR Immediate Shifted XOR Immediate XOR Immediate Shifted
28 1C andi. All D	29 1D andis. All D	30 1E FX Dwd Rot & Select AP MD[S]	31 1F FX Extended Ops All	AND Immediate AND Immediate Shifted See Table 15 on page 204 See Table 23 on page 208
32 20 lwz All D	33 21 lwzu All D	34 22 lbz All D	35 23 lbzu All D	Load Word and Zero Load Word and Zero with Update Load Byte and Zero Load Byte and Zero with Update
36 24 stw All D	37 25 stwu All D	38 26 stb All D	39 27 stbu All D	Store Word Store Word with Update Store Byte Store Byte with Update
40 28 lhz All D	41 29 lhzu All D	42 2A lha All D	43 2B lhau All D	Load Half and Zero Load Half and Zero with Update Load Half Algebraic Load Half Algebraic with Update
44 2C sth All D	45 2D sthu All D	46 2E lmw All D	47 2F stmw All D	Store Half Store Half with Update Load Multiple Word Store Multiple Word
48 30 lfs All D	49 31 lfsu All D	50 32 lfd All D	51 33 lfdu All D	Load Floating-Point Single Load Floating-Point Single with Update Load Floating-Point Double Load Floating-Point Double with Update
52 34 stfs All D	53 35 stfsu All D	54 36 stfd All D	55 37 stfdu All D	Store Floating-Point Single Store Floating-Point Single with Update Store Floating-Point Double Store Floating-Point Double with Update
† 56 38 † lq/lfq † /3 illegal † At/2 DQ/DS	57 39 lfqu, 3 illegal 2 DS	58 3A FX DS-form Loads AP DS	59 3B FP Single Extended Ops AP A	See Table 16 on page 205 See Table 17 on page 205 See Table 18 on page 205 See Table 24 on page 210
60 3C stfq, 3 illegal 2 DS	61 3D stfqu, 3 illegal 2 DS	62 3E FX DS-Form Stores AP DS	63 3F FP Double Extended Ops All	See Table 19 on page 205 See Table 20 on page 205 See Table 21 on page 205 See Table 25 on page 212

Table 15. Extended opcodes for primary opcode 30 (instruction bits 27:30)

	00	**01**	**10**	**11**
00	0 ***rldicl*** AP MD	1 ***rldicl′*** AP MD	2 ***rldicr*** AP MD	3 ***rldicr′*** AP MD
01	4 ***rldic*** AP MD	5 ***rldic′*** AP MD	6 ***rldimi*** AP MD	7 ***rldimi′*** AP MD
10	8 ***rldcl*** AP MDS	9 ***rldcr*** AP MDS		
11	12 ***selii*** At MDS	13 ***selir*** At MDS	14 ***selri*** At MDS	15 ***selrr*** At MDS

Table 16. Extended opcodes for primary opcode 56 (instruction bits 30:31)

	0	**1**
† † † † **0**	0 ***lq / lfq*** At/2 DQ/DS	1 ***lq*** At DQ
1	2 ***lq*** At DQ	3 ***lq*** At DQ

Table 17. Extended opcodes for primary opcode 57 (instruction bits 30:31)

	0	**1**
0	0 ***lfqu*** 2 DS	
1		

Table 18. Extended opcodes for primary opcode 58 (instruction bits 30:31)

	0	**1**
0	0 ***ld*** AP DS	1 ***ldu*** AP DS
1	2 ***lwa*** AP DS	3 ***lmd*** At DS

Table 19. Extended opcodes for primary opcode 60 (instruction bits 30:31)

	0	**1**
0	0 ***stfq*** 2 DS	
1		

Table 20. Extended opcodes for primary opcode 61 (instruction bits 30:31)

	0	**1**
0	0 ***stfqu*** 2 DS	
1		

Table 21. Extended opcodes for primary opcode 62 (instruction bits 30:31)

	0	**1**
0	0 ***std*** AP DS	1 ***stdu*** AP DS
1	2 ***stq*** At DS	3 ***stmd*** At DS

Table 22 (Page 1 of 2). Extended opcodes for primary opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 ***mcrf*** All XL																16 ***bclr*** All XL		18 ***rfid*** AP XL													
00001		33 ***crnor*** All XL																	(50) ***rfi*** All XL	51 Res'd AP												
00010																			82 ***rfscv*** At2O XL													
00011																																
00100		129 ***crandc*** All XL																					150 ***isync*** All XL									
00101																																
00110		193 ***crxor*** All XL																														
00111		225 ***crnand*** All XL																														
01000		257 ***crand*** All XL																														
01001		289 ***creqv*** All XL																														
01010																																
01011																																
01100																																
01101		417 ***crorc*** All XL																														
01110		449 ***cror*** All XL																														
01111																																

Table 22 (Page 2 of 2). Extended opcodes for primary opcode 19 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																	528 ***bcctr*** All XL															
10001																																
10010																																
10011																																
10100																																
10101																																
10110																																
10111																																
11000																																
11001																																
11010																																
11011																																
11100																																
11101																																
11110																																
11111																																

Table 23 (Page 1 of 2). Extended opcodes for primary opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 cmp All X				4 tw All X				8 subfc All XO	9 mulhdu AP XO	10 addc All XO	11 mulhwu AP XO				15 Res0* All				19 mfcr All X	20 lwarx AP X	21 ldx AP X		23 lwzx All X	24 slw All X		26 cntlzw All X	27 sld AP X	28 and All X	29 maskg 2O X	30 rldicl* AP MD	
00001	32 cmpl All X				36 txer At TX				40 subf AP XO							47 *						53 ldux AP X	54 dcbst AP X	55 lwzux All X			58 cntlzd AP X		60 andc All X	61 dsixes At X	62 rldicl* AP MD	
00010	64 cmpla At X				68 td AP X					73 mulhd AP XO		75 mulhw AP XO				79 tdi* AP D			(82) mtsrd AP X	83 mfmsr All X	84 ldarx AP X		86 dcbf AP X	87 lbzx All X						93 dtcs At X	94 rldicr* AP MD	
00011					100 txer′ At TX				104 neg All XO			107 mul 2O XO				111 twi* All D			(114) mtsrdin AP X				118 clf 2O X	119 lbzux All X					124 nor All X		126 rldicr* AP MD	
00100				131 Res'd AP					136 subfe All XO		138 adde All XO					143 *	144 mtcrf All XFX		146 mtmsr All X			149 stdx AP X	150 stwcx. AP X	151 stwx All X	152 slq 2O X	153 sle 2O X					158 rldic* AP MD	159 rlwimi* All M
00101				163 Res'd AP	164 txer′ At TX											175 *			178 mtmsrd X AP			181 stdux AP X		183 stwux All X	184 sliq 2O X						190 rldic* AP MD	191 rlwinm* All M
00110									200 subfze All XO		202 addze All XO					207 *			210 mtsr All X				214 stdcx. AP X	215 stbx All X	216 sllq 2O X	217 sleq 2O X					222 rldimi* AP MD	223 rlmi* 2O M
00111					228 txer′ At TX				232 subfme All XO	233 mulld AP XO	234 addme All XO	235 mullw All XO				239 mulli* All D			242 mtsrin All X				246 dcbtst AP X	247 stbux All X	248 slliq 2O X						254 rldimi* AP MD	255 rlwnm* All M
01000							262 Res'd AP		264 doz 2O XO		266 add All XO					271 subfic* All D			274 Res'd A			277 lscbx 2O X	278 dcbt AP X	279 lhzx All X					284 eqv All X		286 rldcl* AP MDS	287 ori* All D
01001					292 txer′ At TX											303 dozi* 2O D			306 tlbie All X		308 Res'd AP		310 eciwx AP X	311 lhzux All X					316 xor All X		318 rldcl* AP MDS	319 oris* All D
01010				323 Res'd AP								331 div 2O XO				335 cmpli* All D				339 mfspr All XFX		341 lwax AP X		343 lhax All X							350 *	351 xori* All D
01011					356 txer′ At TX				360 abs 2O XO			363 divs 2O XO				367 cmpi* All D			370 tlbia AP X	371 mftb AP XFX		373 lwaux AP X		375 lhaux All X							382 *	383 xoris* All D
01100																399 addic* All D			402 slbmte A X					407 sthx All X					412 orc All X		414 selii* A MDS	415 andi.* All D
01101					420 txer′ At TX											431 addic.* All D			434 slbie AP X				438 ecowx AP X	439 sthux All X					444 or All X		446 selir* A MDS	447 andis.* All D
01110				451 Res'd AP			454 Res'd AP			457 divdu AP XO		459 divwu AP XO				463 addi* All D				467 mtspr All XFX		469 lmd* At DS	(470) dcbi AP X	471 lmw* All D					476 nand All X		478 selri* A MDS	
01111					484 txer′ At TX		486 Res'd AP		488 nabs 2O XO	489 divd AP XO		491 divw AP XO				495 addis* All D			498 slbia AP X	499 settag At XFX		501 stmd* At DS	502 cli 2O X	503 stmw* All D							510 selrr* A MDS	

‡

Table 23 (Page 2 of 2). Extended opcodes for primary opcode 31 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000	512 ***mcrxr*** All X								520 ***subfc′*** All XO	521 ***mulhdu′*** AP XO	522 ***addc′*** All XO	523 ***mulhwu′*** AP XO								531 ***clcs*** 2O X		533 ***lswx*** All X	534 ***lwbrx*** All X	535 ***lfsx*** All X	536 ***srw*** All X	537 ***rrib*** 2O X		539 ***srd*** AP X		541 ***maskir*** 2O X		
10001	544 ***mcrxrt*** At X				548 ***txer′*** At TX				552 ***subf′*** AP XO													565 ***lsdx*** At X	566 ***tlbsync*** AP X	567 ***lfsux*** All X								
10010										585 ***mulhd′*** AP XO		587 ***mulhw′*** AP XO								595 ***mfsr*** All X		597 ***lswi*** All X	598 ***sync*** All X	599 ***lfdx*** All X								
10011					612 ***txer′*** At TX				616 ***neg′*** All XO			619 ***mul′*** 2O XO								627 ***mfsri*** 2O X		629 ***lsdi*** At X	630 ***dclst*** 2O X	631 ***lfdux*** All X								
10100									648 ***subfe′*** All XO		650 ***adde′*** All XO									659 ***mfsrin*** AP X		661 ***stswx*** All X	662 ***stwbrx*** All X	663 ***stfsx*** All X	664 ***srq*** 2O X	665 ***sre*** 2O X						
10101					676 ***txer′*** At TX																	693 ***stsdx*** At X		695 ***stfsux*** All X	696 ***sriq*** 2O X							
10110									712 ***subfze′*** All XO		714 ***addze′*** All XO											725 ***stswi*** All X		727 ***stfdx*** All X	728 ***srlq*** 2O X	729 ***sreq*** 2O X						
10111					740 ***txer′*** At TX				744 ***subfme′*** All XO	745 ***mulld′*** AP XO	746 ***addme′*** All XO	747 ***mullw′*** All XO										757 ***stsdi*** At X	(758) ***dcba*** AP X	759 ***stfdux*** All X	760 ***srliq*** 2O X							
11000									776 ***doz′*** 2O XO		778 ***add′*** All XO												790 ***lhbrx*** All X	791 ***lfqx*** 2 X	792 ***sraw*** All X		794 ***srad*** AP X					
11001					804 ***txer′*** At TX														818 ***rac*** 2O X					823 ***lfqux*** 2 X	824 ***srawi*** All X		826 ***sradi*** AP XS	827 ***sradi′*** AP XS				
11010												843 ***div′*** 2O XO								851 ***slbmfev*** A X			854 ***eieio*** AP X									
11011					868 ***txer′*** At TX				872 ***abs′*** 2O XO			875 ***divs′*** 2O XO											(886) ***vsync*** At X									
11100																			914 Res'd AP	915 ***slbmfee*** A X			918 ***sthbrx*** All X	919 ***stfqx*** 2 X	920 ***sraq*** 2O X	921 ***srea*** 2O X	922 ***extsh*** All X					
11101					932 ***txer′*** At TX														946 Res'd AP					951 ***stfqux*** 2 X	952 ***sraiq*** 2O X		954 ***extsb*** AP X					
11110							966 Res'd AP			969 ***divdu′*** AP XO		971 ***divwu′*** AP XO							978 Res'd AP				982 ***icbi*** AP X	983 ***stfiwx*** AP X			986 ***extsw*** AP X					
11111					996 ***txer′*** At TX		998 Res'd AP		1000 ***nabs′*** 2O XO	1001 ***divd′*** AP XO		1003 ***divw′*** AP XO							1010 Res'd AP				1014 ***dcbz*** All X									

Table 24 (Page 1 of 2). Extended opcodes for primary opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000																			18 *fdivs* AP A		20 *fsubs* AP A	21 *fadds* AP A	22 *fsqrts* AP A		24 *fres* AP A	25 *fmuls* AP A			28 *fmsubs* AP A	29 *fmadds* AP A	30 *fnmsubs* AP A	31 *fnmadds* AP A
00001																			||		||	||	||		||	||			||	||	||	||
00010																			||		||	||	||		||	||			||	||	||	||
00011																			||		||	||	||		||	||			||	||	||	||
00100																			||		||	||	||		||	||			||	||	||	||
00101																			||		||	||	||		||	||			||	||	||	||
00110																			||		||	||	||		||	||			||	||	||	||
00111																			||		||	||	||		||	||			||	||	||	||
01000																			||		||	||	||		||	||			||	||	||	||
01001																			||		||	||	||		||	||			||	||	||	||
01010																			||		||	||	||		||	||			||	||	||	||
01011																			||		||	||	||		||	||			||	||	||	||
01100																			||		||	||	||		||	||			||	||	||	||
01101																			||		||	||	||		||	||			||	||	||	||
01110																			||		||	||	||		||	||			||	||	||	||
01111																			||		||	||	||		||	||			||	||	||	||

Table 24 (Page 2 of 2). Extended opcodes for primary opcode 59 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																			||		||	||	||		||	||			||	||	||	||
10001																			||		||	||	||		||	||			||	||	||	||
10010																			||		||	||	||		||	||			||	||	||	||
10011																			||		||	||	||		||	||			||	||	||	||
10100																			||		||	||	||		||	||			||	||	||	||
10101																			||		||	||	||		||	||			||	||	||	||
10110																			||		||	||	||		||	||			||	||	||	||
10111																			||		||	||	||		||	||			||	||	||	||
11000																			||		||	||	||		||	||			||	||	||	||
11001																			||		||	||	||		||	||			||	||	||	||
11010																			||		||	||	||		||	||			||	||	||	||
11011																			||		||	||	||		||	||			||	||	||	||
11100																			||		||	||	||		||	||			||	||	||	||
11101																			||		||	||	||		||	||			||	||	||	||
11110																			||		||	||	||		||	||			||	||	||	||
11111																			|| *fdivs*		|| *fsubs*	|| *fadds*	|| *fsqrts*		|| *fres*	|| *fmuls*			|| *fmsubs*	|| *fmadds*	|| *fnmsubs*	|| *fnmadds*

Table 25 (Page 1 of 2). Extended opcodes for primary opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
00000	0 ***fcmpu*** All X												12 ***frsp*** All X		14 ***fctiw*** AP2 X	15 ***fctiwz*** AP2 X			18 ***fdiv*** All A		20 ***fsub*** All A	21 ***fadd*** All A	22 ***fsqrt*** AP2 A	23 ***fsel*** AP A		25 ***fmul*** All A	26 ***frsqrte*** AP A		28 ***fmsub*** All A	29 ***fmadd*** All A	30 ***fnmsub*** All A	31 ***fnmadd*** All A
00001	32 ***fcmpo*** All X						38 ***mtfsb1*** All X		40 ***fneg*** All X										||		||	||	||	||		||	||		||	||	||	||
00010	64 ***mcrfs*** All X						70 ***mtfsb0*** All X		72 ***fmr*** All X										||		||	||	||	||		||	||		||	||	||	||
00011																			||		||	||	||	||		||	||		||	||	||	||
00100							134 ***mtfsfi*** All X		136 ***fnabs*** All X										||		||	||	||	||		||	||		||	||	||	||
00101																			||		||	||	||	||		||	||		||	||	||	||
00110																			||		||	||	||	||		||	||		||	||	||	||
00111																			||		||	||	||	||		||	||		||	||	||	||
01000									264 ***fabs*** All X										||		||	||	||	||		||	||		||	||	||	||
01001																			||		||	||	||	||		||	||		||	||	||	||
01010																			||		||	||	||	||		||	||		||	||	||	||
01011																			||		||	||	||	||		||	||		||	||	||	||
01100																			||		||	||	||	||		||	||		||	||	||	||
01101																			||		||	||	||	||		||	||		||	||	||	||
01110																			||		||	||	||	||		||	||		||	||	||	||
01111																			||		||	||	||	||		||	||		||	||	||	||

Table 25 (Page 2 of 2). Extended opcodes for primary opcode 63 (instruction bits 21:30)

	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
10000																			||		||	||	||	||		||	||		||	||	||	||
10001																			||		||	||	||	||		||	||		||	||	||	||
10010								583 ***mffs*** All X											||		||	||	||	||		||	||		||	||	||	||
10011								(615) ***mffpspr*** Api X											||		||	||	||	||		||	||		||	||	||	||
10100																			||		||	||	||	||		||	||		||	||	||	||
10101																			||		||	||	||	||		||	||		||	||	||	||
10110								711 ***mtfsf*** All XFL											||		||	||	||	||		||	||		||	||	||	||
10111								(743) ***mtfpspr*** Api X											||		||	||	||	||		||	||		||	||	||	||
11000																			||		||	||	||	||		||	||		||	||	||	||
11001															814 ***fctid*** AP X	815 ***fctidz*** AP X			||		||	||	||	||		||	||		||	||	||	||
11010															846 ***fcfid*** AP X				||		||	||	||	||		||	||		||	||	||	||
11011																			||		||	||	||	||		||	||		||	||	||	||
11100																			||		||	||	||	||		||	||		||	||	||	||
11101																			||		||	||	||	||		||	||		||	||	||	||
11110																			||		||	||	||	||		||	||		||	||	||	||
11111																			|| ***fdiv***		|| ***fsub***	|| ***fadd***	|| ***fsqrt***	|| ***fsel***		|| ***fmul***	|| ***frsqrte***		|| ***fmsub***	|| ***fmadd***	|| ***fnmsub***	|| ***fnmadd***

Appendix J. PowerPC AS Instruction Set Sorted by Opcode

This appendix lists all the instructions in the PowerPC AS Architecture, in order by opcode. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC AS User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*). If an instruction is defined in more than one of these Books, the lowest-numbered Book is used.

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
D	2			72	tdi	Trap Doubleword Immediate
D	3			72	twi	Trap Word Immediate
D	7			64	mulli	Multiply Low Immediate
D	8		SR	61	subfic	Subtract From Immediate Carrying
D	10			69	cmpli	Compare Logical Immediate
D	11			68	cmpi	Compare Immediate
D	12		SR	60	addic	Add Immediate Carrying
D	13		SR	60	addic.	Add Immediate Carrying and Record
D	14			59	addi	Add Immediate
D	15			59	addis	Add Immediate Shifted
B	16		CT	27	bc[l][a]	Branch Conditional
SC	17	0	TA	29	scv	System Call Vectored
SC	17	1		29	sc	System Call
I	18			27	b[l][a]	Branch
XL	19	0		32	mcrf	Move Condition Register Field
XL	19	16	CT	28	bclr[l]	Branch Conditional to Link Register
XL	19	18		III	rfid	Return from Interrupt Doubleword
XL	19	33		31	crnor	Condition Register NOR
XL	19	82	TA	III	rfscv	Return From System Call Vectored
XL	19	129		31	crandc	Condition Register AND with Complement
XL	19	150		II	isync	Instruction Synchronize
XL	19	193		30	crxor	Condition Register XOR
XL	19	225		30	crnand	Condition Register NAND
XL	19	257		30	crand	Condition Register AND
XL	19	289		31	creqv	Condition Register Equivalent
XL	19	417		31	crorc	Condition Register OR with Complement
XL	19	449		30	cror	Condition Register OR
XL	19	528	CT	28	bcctr[l]	Branch Conditional to Count Register
M	20		SR	89	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR	86	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR	88	rlwnm[.]	Rotate Left Word then AND with Mask
D	24			79	ori	OR Immediate
D	25			79	oris	OR Immediate Shifted
D	26			79	xori	XOR Immediate
D	27			79	xoris	XOR Immediate Shifted
D	28		SR	78	andi.	AND Immediate
D	29		SR	78	andis.	AND Immediate Shifted
MD	30	0	SR	85	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	SR	85	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	2	SR	86	rldic[.]	Rotate Left Doubleword Immediate then Clear

	Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
		Primary	Extend				
	MD	30	3	SR	89	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
	MDS	30	8	SR	87	rldcl[.]	Rotate Left Doubleword then Clear Left
	MDS	30	9	SR	88	rldcr[.]	Rotate Left Doubleword then Clear Right
	MDS	30	12	TA	76	selii[.]	Select Immediate-Immediate
	MDS	30	13	TA	76	selir[.]	Select Immediate-Register
	MDS	30	14	TA	77	selri[.]	Select Register-Immediate
	MDS	30	15	TA	77	selrr[.]	Select Register-Register
	X	31	0		68	cmp	Compare
	X	31	4		73	tw	Trap Word
	XO	31	8	SR	61	subfc[o][.]	Subtract From Carrying
	XO	31	9	SR	65	mulhdu[.]	Multiply High Doubleword Unsigned
	XO	31	10	SR	61	addc[o][.]	Add Carrying
	XO	31	11	SR	65	mulhwu[.]	Multiply High Word Unsigned
|	XFX	31	19		97	mfcr	Move From Condition Register
|	XFX	31	19		138	mfcr	Move From Condition Register (optional version)
†	X	31	20		II	lwarx	Load Word And Reserve Indexed
	X	31	21		42	ldx	Load Doubleword Indexed
	X	31	23		40	lwzx	Load Word and Zero Indexed
	X	31	24	SR	90	slw[.]	Shift Left Word
	X	31	26	SR	83	cntlzw[.]	Count Leading Zeros Word
	X	31	27	SR	90	sld[.]	Shift Left Doubleword
	X	31	28	SR	80	and[.]	AND
	X	31	32		69	cmpl	Compare Logical
	TX	31	36	TA	74	txer	Trap on XER
	XO	31	40	SR	60	subf[o][.]	Subtract From
	X	31	53		42	ldux	Load Doubleword with Update Indexed
	X	31	54		II	dcbst	Data Cache Block Store
	X	31	55		40	lwzux	Load Word and Zero with Update Indexed
	X	31	58	SR	83	cntlzd[.]	Count Leading Zeros Doubleword
	X	31	60	SR	81	andc[.]	AND with Complement
	X	31	61	TA	94	dsixes	Decimal Sixes
	X	31	64	TA	70	cmpla	Compare Logical Addresses
	X	31	68		73	td	Trap Doubleword
	XO	31	73	SR	65	mulhd[.]	Multiply High Doubleword
	XO	31	75	SR	65	mulhw[.]	Multiply High Word
|							
	X	31	83		III	mfmsr	Move From Machine State Register
†	X	31	84		II	ldarx	Load Doubleword And Reserve Indexed
	X	31	86		II	dcbf	Data Cache Block Flush
	X	31	87		37	lbzx	Load Byte and Zero Indexed
	X	31	93	TA	94	dtcs.	Decimal Test and Clear Sign
	XO	31	104	SR	63	neg[o][.]	Negate
|							
	X	31	119		37	lbzux	Load Byte and Zero with Update Indexed
	X	31	124	SR	81	nor[.]	NOR
	XO	31	136	SR	62	subfe[o][.]	Subtract From Extended
	XO	31	138	SR	62	adde[o][.]	Add Extended
	XFX	31	144		97	mtcrf	Move To Condition Register Fields
|	XFX	31	144		138	mtcrf	Move To Condition Register Field (optional version)
	X	31	146		III	mtmsr	Move To Machine State Register
	X	31	149		47	stdx	Store Doubleword Indexed
†	X	31	150		II	stwcx.	Store Word Conditional Indexed
	X	31	151		46	stwx	Store Word Indexed
	X	31	178		III	mtmsrd	Move To Machine State Register Doubleword
	X	31	181		47	stdux	Store Doubleword with Update Indexed
	X	31	183		46	stwux	Store Word with Update Indexed
	XO	31	200	SR	63	subfze[o][.]	Subtract From Zero Extended
	XO	31	202	SR	63	addze[o][.]	Add to Zero Extended

	Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
		Primary	Extend				
	X	31	210	32	III	mtsr	Move To Segment Register
†	X	31	214		II	stdcx.	Store Doubleword Conditional Indexed
	X	31	215		44	stbx	Store Byte Indexed
	XO	31	232	SR	62	subfme[o][.]	Subtract From Minus One Extended
	XO	31	233	SR	64	mulld[o][.]	Multiply Low Doubleword
	XO	31	234	SR	62	addme[o][.]	Add to Minus One Extended
	XO	31	235	SR	64	mullw[o][.]	Multiply Low Word
	X	31	242	32	III	mtsrin	Move To Segment Register Indirect
	X	31	246		II	dcbtst	Data Cache Block Touch for Store
	X	31	247		44	stbux	Store Byte with Update Indexed
	XO	31	266	SR	60	add[o][.]	Add
	X	31	278		II	dcbt	Data Cache Block Touch
	X	31	279		38	lhzx	Load Halfword and Zero Indexed
	X	31	284	SR	81	eqv[.]	Equivalent
	X	31	306	64	III	tlbie	TLB Invalidate Entry
	X	31	310		II	eciwx	External Control In Word Indexed
	X	31	311		38	lhzux	Load Halfword and Zero with Update Indexed
	X	31	316	SR	80	xor[.]	XOR
	XFX	31	339		96	mfspr	Move From Special Purpose Register
	X	31	341		41	lwax	Load Word Algebraic Indexed
	X	31	343		39	lhax	Load Halfword Algebraic Indexed
	X	31	370		III	tlbia	TLB Invalidate All
	XFX	31	371		II	mftb	Move From Time Base
	X	31	373		41	lwaux	Load Word Algebraic with Update Indexed
	X	31	375		39	lhaux	Load Halfword Algebraic with Update Indexed
	X	31	402		III	slbmte	SLB Move To Entry
	X	31	407		45	sthx	Store Halfword Indexed
	X	31	412	SR	81	orc[.]	OR with Complement
	XS	31	413	SR	92	sradi[.]	Shift Right Algebraic Doubleword Immediate
	X	31	434		III	slbie	SLB Invalidate Entry
	X	31	438		II	ecowx	External Control Out Word Indexed
	X	31	439		45	sthux	Store Halfword with Update Indexed
	X	31	444	SR	80	or[.]	OR
	XO	31	457	SR	67	divdu[o][.]	Divide Doubleword Unsigned
	XO	31	459	SR	67	divwu[o][.]	Divide Word Unsigned
	XFX	31	467		95	mtspr	Move To Special Purpose Register
	X	31	476	SR	80	nand[.]	NAND
	XO	31	489	SR	66	divd[o][.]	Divide Doubleword
	XO	31	491	SR	66	divw[o][.]	Divide Word
	X	31	498		III	slbia	SLB Invalidate All
	XFX	31	499	TA	96	settag	Set XER Tag
	X	31	512		97	mcrxr	Move to Condition Register from XER
	X	31	533		55	lswx	Load String Word Indexed
	X	31	534		49	lwbrx	Load Word Byte-Reverse Indexed
	X	31	535		118	lfsx	Load Floating-Point Single Indexed
	X	31	536	SR	91	srw[.]	Shift Right Word
	X	31	539	SR	91	srd[.]	Shift Right Doubleword
	X	31	544	TA	97	mcrxrt	Move to Condition Register from XER TGCC
	X	31	565	TA	55	lsdx	Load String Doubleword Indexed
	X	31	566		III	tlbsync	TLB Synchronize
	X	31	567		118	lfsux	Load Floating-Point Single with Update Indexed
	X	31	595	32	III	mfsr	Move From Segment Register
	X	31	597		54	lswi	Load String Word Immediate
†	X	31	598		II	sync	Synchronize
	X	31	599		119	lfdx	Load Floating-Point Double Indexed
	X	31	629	TA	54	lsdi	Load String Doubleword Immediate
	X	31	631		119	lfdux	Load Floating-Point Double with Update Indexed

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	659	32	III	mfsrin	Move From Segment Register Indirect
X	31	661		57	stswx	Store String Word Indexed
X	31	662		50	stwbrx	Store Word Byte-Reverse Indexed
X	31	663		121	stfsx	Store Floating-Point Single Indexed
X	31	693	TA	57	stsdx	Store String Doubleword Indexed
X	31	695		121	stfsux	Store Floating-Point Single with Update Indexed
X	31	725		56	stswi	Store String Word Immediate
X	31	727		122	stfdx	Store Floating-Point Double Indexed
X	31	757	TA	56	stsdi	Store String Doubleword Immediate
X	31	759		122	stfdux	Store Floating-Point Double with Update Indexed
X	31	790		49	lhbrx	Load Halfword Byte-Reverse Indexed
X	31	792	SR	93	sraw[.]	Shift Right Algebraic Word
X	31	794	SR	93	srad[.]	Shift Right Algebraic Doubleword
X	31	824	SR	92	srawi[.]	Shift Right Algebraic Word Immediate
X	31	851		III	slbmfev	SLB Move From Entry VSID
X	31	854		II	eieio	Enforce In-order Execution of I/O
X	31	915		III	slbmfee	SLB Move From Entry ESID
X	31	918		50	sthbrx	Store Halfword Byte-Reverse Indexed
X	31	922	SR	82	extsh[.]	Extend Sign Halfword
X	31	954	SR	82	extsb[.]	Extend Sign Byte
X	31	982		II	icbi	Instruction Cache Block Invalidate
X	31	983		123	stfiwx	Store Floating-Point as Integer Word Indexed
X	31	986	SR	82	extsw[.]	Extend Sign Word
X	31	1014		II	dcbz	Data Cache Block set to Zero
D	32			40	lwz	Load Word and Zero
D	33			40	lwzu	Load Word and Zero with Update
D	34			37	lbz	Load Byte and Zero
D	35			37	lbzu	Load Byte and Zero with Update
D	36			46	stw	Store Word
D	37			46	stwu	Store Word with Update
D	38			44	stb	Store Byte
D	39			44	stbu	Store Byte with Update
D	40			38	lhz	Load Halfword and Zero
D	41			38	lhzu	Load Halfword and Zero with Update
D	42			39	lha	Load Halfword Algebraic
D	43			39	lhau	Load Halfword Algebraic with Update
D	44			45	sth	Store Halfword
D	45			45	sthu	Store Halfword with Update
D	46			51	lmw	Load Multiple Word
D	47			52	stmw	Store Multiple Word
D	48			118	lfs	Load Floating-Point Single
D	49			118	lfsu	Load Floating-Point Single with Update
D	50			119	lfd	Load Floating-Point Double
D	51			119	lfdu	Load Floating-Point Double with Update
D	52			121	stfs	Store Floating-Point Single
D	53			121	stfsu	Store Floating-Point Single with Update
D	54			122	stfd	Store Floating-Point Double
D	55			122	stfdu	Store Floating-Point Double with Update
DQ	56		TA	43	lq	Load Quadword
DS	58	0		42	ld	Load Doubleword
DS	58	1		42	ldu	Load Doubleword with Update
DS	58	2		41	lwa	Load Word Algebraic
DS	58	3	TA	51	lmd	Load Multiple Doubleword
A	59	18		126	fdivs[.]	Floating Divide Single
A	59	20		125	fsubs[.]	Floating Subtract Single
A	59	21		125	fadds[.]	Floating Add Single

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
A	59	22		140	fsqrts[.]	Floating Square Root Single
A	59	24		140	fres[.]	Floating Reciprocal Estimate Single
A	59	25		126	fmuls[.]	Floating Multiply Single
A	59	28		127	fmsubs[.]	Floating Multiply-Subtract Single
A	59	29		127	fmadds[.]	Floating Multiply-Add Single
A	59	30		128	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	31		128	fnmadds[.]	Floating Negative Multiply-Add Single
DS	62	0		47	std	Store Doubleword
DS	62	1		47	stdu	Store Doubleword with Update
DS	62	2	TA	48	stq	Store Quadword
DS	62	3	TA	52	stmd	Store Multiple Doubleword
X	63	0		133	fcmpu	Floating Compare Unordered
X	63	12		129	frsp[.]	Floating Round to Single-Precision
X	63	14		131	fctiw[.]	Floating Convert To Integer Word
X	63	15		131	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18		126	fdiv[.]	Floating Divide
A	63	20		125	fsub[.]	Floating Subtract
A	63	21		125	fadd[.]	Floating Add
A	63	22		140	fsqrt[.]	Floating Square Root
A	63	23		141	fsel[.]	Floating Select
A	63	25		126	fmul[.]	Floating Multiply
A	63	26		141	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	28		127	fmsub[.]	Floating Multiply-Subtract
A	63	29		127	fmadd[.]	Floating Multiply-Add
A	63	30		128	fnmsub[.]	Floating Negative Multiply-Subtract
A	63	31		128	fnmadd[.]	Floating Negative Multiply-Add
X	63	32		133	fcmpo	Floating Compare Ordered
X	63	38		136	mtfsb1[.]	Move To FPSCR Bit 1
X	63	40		124	fneg[.]	Floating Negate
X	63	64		134	mcrfs	Move to Condition Register from FPSCR
X	63	70		136	mtfsb0[.]	Move To FPSCR Bit 0
X	63	72		124	fmr[.]	Floating Move Register
X	63	134		135	mtfsfi[.]	Move To FPSCR Field Immediate
X	63	136		124	fnabs[.]	Floating Negative Absolute Value
X	63	264		124	fabs[.]	Floating Absolute Value
X	63	583		134	mffs[.]	Move From FPSCR
XFL	63	711		135	mtfsf[.]	Move To FPSCR Fields
X	63	814		130	fctid[.]	Floating Convert To Integer Doubleword
X	63	815		130	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	846		132	fcfid[.]	Floating Convert From Integer Doubleword

[1]See key to mode dependency column, on page 225.

Appendix K. PowerPC AS Instruction Set Sorted by Mnemonic

This appendix lists all the instructions in the PowerPC AS Architecture, in order by mnemonic. A page number is shown for instructions that are defined in this Book (Book I, *PowerPC AS User Instruction Set Architecture*), and the Book number is shown for instructions that are defined in other Books (Book II, *PowerPC AS Virtual Environment Architecture*, and Book III, *PowerPC AS Operating Environment Architecture*). If an instruction is defined in more than one of these Books, the lowest-numbered Book is used.

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
XO	31	266	SR	60	add[o][.]	Add
XO	31	10	SR	61	addc[o][.]	Add Carrying
XO	31	138	SR	62	adde[o][.]	Add Extended
D	14			59	addi	Add Immediate
D	12		SR	60	addic	Add Immediate Carrying
D	13		SR	60	addic.	Add Immediate Carrying and Record
D	15			59	addis	Add Immediate Shifted
XO	31	234	SR	62	addme[o][.]	Add to Minus One Extended
XO	31	202	SR	63	addze[o][.]	Add to Zero Extended
X	31	28	SR	80	and[.]	AND
X	31	60	SR	81	andc[.]	AND with Complement
D	28		SR	78	andi.	AND Immediate
D	29		SR	78	andis.	AND Immediate Shifted
I	18			27	b[l][a]	Branch
B	16		CT	27	bc[l][a]	Branch Conditional
XL	19	528	CT	28	bcctr[l]	Branch Conditional to Count Register
XL	19	16	CT	28	bclr[l]	Branch Conditional to Link Register
X	31	0		68	cmp	Compare
D	11			68	cmpi	Compare Immediate
X	31	32		69	cmpl	Compare Logical
X	31	64	TA	70	cmpla	Compare Logical Addresses
D	10			69	cmpli	Compare Logical Immediate
X	31	58	SR	83	cntlzd[.]	Count Leading Zeros Doubleword
X	31	26	SR	83	cntlzw[.]	Count Leading Zeros Word
XL	19	257		30	crand	Condition Register AND
XL	19	129		31	crandc	Condition Register AND with Complement
XL	19	289		31	creqv	Condition Register Equivalent
XL	19	225		30	crnand	Condition Register NAND
XL	19	33		31	crnor	Condition Register NOR
XL	19	449		30	cror	Condition Register OR
XL	19	417		31	crorc	Condition Register OR with Complement
XL	19	193		30	crxor	Condition Register XOR
X	31	86		II	dcbf	Data Cache Block Flush
X	31	54		II	dcbst	Data Cache Block Store
X	31	278		II	dcbt	Data Cache Block Touch
X	31	246		II	dcbtst	Data Cache Block Touch for Store
X	31	1014		II	dcbz	Data Cache Block set to Zero
XO	31	489	SR	66	divd[o][.]	Divide Doubleword
XO	31	457	SR	67	divdu[o][.]	Divide Doubleword Unsigned

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
XO	31	491	SR	66	divw[o][.]	Divide Word
XO	31	459	SR	67	divwu[o][.]	Divide Word Unsigned
X	31	61	TA	94	dsixes	Decimal Sixes
X	31	93	TA	94	dtcs.	Decimal Test and Clear Sign
X	31	310		II	eciwx	External Control In Word Indexed
X	31	438		II	ecowx	External Control Out Word Indexed
X	31	854		II	eieio	Enforce In-order Execution of I/O
X	31	284	SR	81	eqv[.]	Equivalent
X	31	954	SR	82	extsb[.]	Extend Sign Byte
X	31	922	SR	82	extsh[.]	Extend Sign Halfword
X	31	986	SR	82	extsw[.]	Extend Sign Word
X	63	264		124	fabs[.]	Floating Absolute Value
A	63	21		125	fadd[.]	Floating Add
A	59	21		125	fadds[.]	Floating Add Single
X	63	846		132	fcfid[.]	Floating Convert From Integer Doubleword
X	63	32		133	fcmpo	Floating Compare Ordered
X	63	0		133	fcmpu	Floating Compare Unordered
X	63	814		130	fctid[.]	Floating Convert To Integer Doubleword
X	63	815		130	fctidz[.]	Floating Convert To Integer Doubleword with round toward Zero
X	63	14		131	fctiw[.]	Floating Convert To Integer Word
X	63	15		131	fctiwz[.]	Floating Convert To Integer Word with round toward Zero
A	63	18		126	fdiv[.]	Floating Divide
A	59	18		126	fdivs[.]	Floating Divide Single
A	63	29		127	fmadd[.]	Floating Multiply-Add
A	59	29		127	fmadds[.]	Floating Multiply-Add Single
X	63	72		124	fmr[.]	Floating Move Register
A	63	28		127	fmsub[.]	Floating Multiply-Subtract
A	59	28		127	fmsubs[.]	Floating Multiply-Subtract Single
A	63	25		126	fmul[.]	Floating Multiply
A	59	25		126	fmuls[.]	Floating Multiply Single
X	63	136		124	fnabs[.]	Floating Negative Absolute Value
X	63	40		124	fneg[.]	Floating Negate
A	63	31		128	fnmadd[.]	Floating Negative Multiply-Add
A	59	31		128	fnmadds[.]	Floating Negative Multiply-Add Single
A	63	30		128	fnmsub[.]	Floating Negative Multiply-Subtract
A	59	30		128	fnmsubs[.]	Floating Negative Multiply-Subtract Single
A	59	24		140	fres[.]	Floating Reciprocal Estimate Single
X	63	12		129	frsp[.]	Floating Round to Single-Precision
A	63	26		141	frsqrte[.]	Floating Reciprocal Square Root Estimate
A	63	23		141	fsel[.]	Floating Select
A	63	22		140	fsqrt[.]	Floating Square Root
A	59	22		140	fsqrts[.]	Floating Square Root Single
A	63	20		125	fsub[.]	Floating Subtract
A	59	20		125	fsubs[.]	Floating Subtract Single
X	31	982		II	icbi	Instruction Cache Block Invalidate
XL	19	150		II	isync	Instruction Synchronize
D	34			37	lbz	Load Byte and Zero
D	35			37	lbzu	Load Byte and Zero with Update
X	31	119		37	lbzux	Load Byte and Zero with Update Indexed
X	31	87		37	lbzx	Load Byte and Zero Indexed
DS	58	0		42	ld	Load Doubleword
† X	31	84		II	ldarx	Load Doubleword And Reserve Indexed
DS	58	1		42	ldu	Load Doubleword with Update
X	31	53		42	ldux	Load Doubleword with Update Indexed
X	31	21		42	ldx	Load Doubleword Indexed
D	50			119	lfd	Load Floating-Point Double
D	51			119	lfdu	Load Floating-Point Double with Update

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
X	31	631		119	lfdux	Load Floating-Point Double with Update Indexed
X	31	599		119	lfdx	Load Floating-Point Double Indexed
D	48			118	lfs	Load Floating-Point Single
D	49			118	lfsu	Load Floating-Point Single with Update
X	31	567		118	lfsux	Load Floating-Point Single with Update Indexed
X	31	535		118	lfsx	Load Floating-Point Single Indexed
D	42			39	lha	Load Halfword Algebraic
D	43			39	lhau	Load Halfword Algebraic with Update
X	31	375		39	lhaux	Load Halfword Algebraic with Update Indexed
X	31	343		39	lhax	Load Halfword Algebraic Indexed
X	31	790		49	lhbrx	Load Halfword Byte-Reverse Indexed
D	40			38	lhz	Load Halfword and Zero
D	41			38	lhzu	Load Halfword and Zero with Update
X	31	311		38	lhzux	Load Halfword and Zero with Update Indexed
X	31	279		38	lhzx	Load Halfword and Zero Indexed
DS	58	3	TA	51	lmd	Load Multiple Doubleword
D	46			51	lmw	Load Multiple Word
DQ	56		TA	43	lq	Load Quadword
X	31	629	TA	54	lsdi	Load String Doubleword Immediate
X	31	565	TA	55	lsdx	Load String Doubleword Indexed
X	31	597		54	lswi	Load String Word Immediate
X	31	533		55	lswx	Load String Word Indexed
DS	58	2		41	lwa	Load Word Algebraic
† X	31	20		II	lwarx	Load Word And Reserve Indexed
X	31	373		41	lwaux	Load Word Algebraic with Update Indexed
X	31	341		41	lwax	Load Word Algebraic Indexed
X	31	534		49	lwbrx	Load Word Byte-Reverse Indexed
D	32			40	lwz	Load Word and Zero
D	33			40	lwzu	Load Word and Zero with Update
X	31	55		40	lwzux	Load Word and Zero with Update Indexed
X	31	23		40	lwzx	Load Word and Zero Indexed
XL	19	0		32	mcrf	Move Condition Register Field
X	63	64		134	mcrfs	Move to Condition Register from FPSCR
X	31	512		97	mcrxr	Move to Condition Register from XER
X	31	544	TA	97	mcrxrt	Move to Condition Register from XER TGCC
XFX	31	19		97	mfcr	Move From Condition Register
XFX	31	19		138	mfcr	Move From Condition Register (optional version)
X	63	583		134	mffs[.]	Move From FPSCR
X	31	83		III	mfmsr	Move From Machine State Register
XFX	31	339		96	mfspr	Move From Special Purpose Register
X	31	595	32	III	mfsr	Move From Segment Register
X	31	659	32	III	mfsrin	Move From Segment Register Indirect
XFX	31	371		II	mftb	Move From Time Base
XFX	31	144		97	mtcrf	Move To Condition Register Fields
XFX	31	144		138	mtcrf	Move To Condition Register Field (optional version)
X	63	70		136	mtfsb0[.]	Move To FPSCR Bit 0
X	63	38		136	mtfsb1[.]	Move To FPSCR Bit 1
XFL	63	711		135	mtfsf[.]	Move To FPSCR Fields
X	63	134		135	mtfsfi[.]	Move To FPSCR Field Immediate
X	31	146		III	mtmsr	Move To Machine State Register
X	31	178		III	mtmsrd	Move To Machine State Register Doubleword
XFX	31	467		95	mtspr	Move To Special Purpose Register
X	31	210	32	III	mtsr	Move To Segment Register
X	31	242	32	III	mtsrin	Move To Segment Register Indirect
XO	31	73	SR	65	mulhd[.]	Multiply High Doubleword
XO	31	9	SR	65	mulhdu[.]	Multiply High Doubleword Unsigned
XO	31	75	SR	65	mulhw[.]	Multiply High Word

Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
	Primary	Extend				
XO	31	11	SR	65	mulhwu[.]	Multiply High Word Unsigned
XO	31	233	SR	64	mulld[o][.]	Multiply Low Doubleword
D	7			64	mulli	Multiply Low Immediate
XO	31	235	SR	64	mullw[o][.]	Multiply Low Word
X	31	476	SR	80	nand[.]	NAND
XO	31	104	SR	63	neg[o][.]	Negate
X	31	124	SR	81	nor[.]	NOR
X	31	444	SR	80	or[.]	OR
X	31	412	SR	81	orc[.]	OR with Complement
D	24			79	ori	OR Immediate
D	25			79	oris	OR Immediate Shifted
XL	19	18		III	rfid	Return from Interrupt Doubleword
XL	19	82	TA	III	rfscv	Return From System Call Vectored
MDS	30	8	SR	87	rldcl[.]	Rotate Left Doubleword then Clear Left
MDS	30	9	SR	88	rldcr[.]	Rotate Left Doubleword then Clear Right
MD	30	2	SR	86	rldic[.]	Rotate Left Doubleword Immediate then Clear
MD	30	0	SR	85	rldicl[.]	Rotate Left Doubleword Immediate then Clear Left
MD	30	1	SR	85	rldicr[.]	Rotate Left Doubleword Immediate then Clear Right
MD	30	3	SR	89	rldimi[.]	Rotate Left Doubleword Immediate then Mask Insert
M	20		SR	89	rlwimi[.]	Rotate Left Word Immediate then Mask Insert
M	21		SR	86	rlwinm[.]	Rotate Left Word Immediate then AND with Mask
M	23		SR	88	rlwnm[.]	Rotate Left Word then AND with Mask
SC	17	1		29	sc	System Call
SC	17	0	TA	29	scv	System Call Vectored
MDS	30	12	TA	76	selii[.]	Select Immediate-Immediate
MDS	30	13	TA	76	selir[.]	Select Immediate-Register
MDS	30	14	TA	77	selri[.]	Select Register-Immediate
MDS	30	15	TA	77	selrr[.]	Select Register-Register
XFX	31	499	TA	96	settag	Set XER Tag
X	31	498		III	slbia	SLB Invalidate All
X	31	434		III	slbie	SLB Invalidate Entry
X	31	915		III	slbmfee	SLB Move From Entry ESID
X	31	851		III	slbmfev	SLB Move From Entry VSID
X	31	402		III	slbmte	SLB Move To Entry
X	31	27	SR	90	sld[.]	Shift Left Doubleword
X	31	24	SR	90	slw[.]	Shift Left Word
X	31	794	SR	93	srad[.]	Shift Right Algebraic Doubleword
XS	31	413	SR	92	sradi[.]	Shift Right Algebraic Doubleword Immediate
X	31	792	SR	93	sraw[.]	Shift Right Algebraic Word
X	31	824	SR	92	srawi[.]	Shift Right Algebraic Word Immediate
X	31	539	SR	91	srd[.]	Shift Right Doubleword
X	31	536	SR	91	srw[.]	Shift Right Word
D	38			44	stb	Store Byte
D	39			44	stbu	Store Byte with Update
X	31	247		44	stbux	Store Byte with Update Indexed
X	31	215		44	stbx	Store Byte Indexed
DS	62	0		47	std	Store Doubleword
† X	31	214		II	stdcx.	Store Doubleword Conditional Indexed
DS	62	1		47	stdu	Store Doubleword with Update
X	31	181		47	stdux	Store Doubleword with Update Indexed
X	31	149		47	stdx	Store Doubleword Indexed
D	54			122	stfd	Store Floating-Point Double
D	55			122	stfdu	Store Floating-Point Double with Update
X	31	759		122	stfdux	Store Floating-Point Double with Update Indexed
X	31	727		122	stfdx	Store Floating-Point Double Indexed
X	31	983		123	stfiwx	Store Floating-Point as Integer Word Indexed
D	52			121	stfs	Store Floating-Point Single

	Form	Opcode		Mode Dep.[1]	Page / Bk	Mnemonic	Instruction
		Primary	Extend				
	D	53			121	stfsu	Store Floating-Point Single with Update
	X	31	695		121	stfsux	Store Floating-Point Single with Update Indexed
	X	31	663		121	stfsx	Store Floating-Point Single Indexed
	D	44			45	sth	Store Halfword
	X	31	918		50	sthbrx	Store Halfword Byte-Reverse Indexed
	D	45			45	sthu	Store Halfword with Update
	X	31	439		45	sthux	Store Halfword with Update Indexed
	X	31	407		45	sthx	Store Halfword Indexed
	DS	62	3	TA	52	stmd	Store Multiple Doubleword
	D	47			52	stmw	Store Multiple Word
	DS	62	2	TA	48	stq	Store Quadword
	X	31	757	TA	56	stsdi	Store String Doubleword Immediate
	X	31	693	TA	57	stsdx	Store String Doubleword Indexed
	X	31	725		56	stswi	Store String Word Immediate
	X	31	661		57	stswx	Store String Word Indexed
	D	36			46	stw	Store Word
	X	31	662		50	stwbrx	Store Word Byte-Reverse Indexed
†	X	31	150		II	stwcx.	Store Word Conditional Indexed
	D	37			46	stwu	Store Word with Update
	X	31	183		46	stwux	Store Word with Update Indexed
	X	31	151		46	stwx	Store Word Indexed
	XO	31	40	SR	60	subf[o][.]	Subtract From
	XO	31	8	SR	61	subfc[o][.]	Subtract From Carrying
	XO	31	136	SR	62	subfe[o][.]	Subtract From Extended
	D	8		SR	61	subfic	Subtract From Immediate Carrying
	XO	31	232	SR	62	subfme[o][.]	Subtract From Minus One Extended
	XO	31	200	SR	63	subfze[o][.]	Subtract From Zero Extended
†	X	31	598		II	sync	Synchronize
	X	31	68		73	td	Trap Doubleword
	D	2			72	tdi	Trap Doubleword Immediate
	X	31	370		III	tlbia	TLB Invalidate All
|	X	31	306	64	III	tlbie	TLB Invalidate Entry
	X	31	566		III	tlbsync	TLB Synchronize
	X	31	4		73	tw	Trap Word
	D	3			72	twi	Trap Word Immediate
	TX	31	36	TA	74	txer	Trap on XER
|							
	X	31	316	SR	80	xor[.]	XOR
	D	26			79	xori	XOR Immediate
	D	27			79	xoris	XOR Immediate Shifted

[1]Key to Mode Dependency Column

† Except as described below and in Section 1.12.3, "Effective Address Calculation" on page 17, all instructions 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.

†	CT	If the instruction tests the Count Register, it tests the low-order 32 bits in 32-bit mode and all 64 bits in 64-bit mode.
†	SR	The setting of status registers (such as XER and CR0) is mode-dependent.
†	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.

Index

Last Page - End of Document