PowerPC AS Virtual Environment Architecture

Book II

Version 2.00

Feb. 24, 1999

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

Technical Content:

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

IBM Confidential - Feb. 24, 1999

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

NOTES

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

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC AS User Instruction Set Architecture, that are provided by the PowerPC AS Virtual Environment Architecture. It covers the storage model and related instructions and facilities available to the application programmer, and the Time Base as seen by the application programmer.

Other related documents define the PowerPC AS User Instruction Set Architecture, the PowerPC AS Operating Environment Architecture, and PowerPC AS Implementation Features. Book I, *PowerPC AS User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book 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.

User Responsibilities

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- Destroy the entire document when it is superseded, obsolete, or no longer needed.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

Next Scheduled Review

There is no scheduled review.

Approval Process

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

Approvals

This version has been approved by the PARB.

Version Verification for those with access to KISS64

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

Version Verification for those without access to KISS64

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

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

Table of Contents

Figures

Changes as of 1999/02/24 Version 2.00

<table>
<tr><th>change</th><th>reason</th><th>page</th></tr>
<tr><td>Make several changes related to Data Cache instructions.
■ Add an optional version of the dcbt instruction with a Touch Hint field.
■ State that the hint provided by dcbt and dcbtst is ignored if the specified block is in Guarded storage.
■ Weaken the description of the hint provided by dcbt and dcbtst, making it merely a statement that the program is likely soon to access the block.
■ Change dcbt and dcbtst so that the actions caused by these instructions cannot be synchronized by software.
■ Modify dcbt and dcbtst so that they never invoke the system data storage error handler.
■ Make minor wording changes for consistency in the dcbt, dcbtst, mftb, eciwx, and ecowx descriptions.</td><td>RFC02000 and Correspondence of 3 Nov. '98. In addition the following changes were made.
■ An Engineering Note was added to the description of dcbt in Section 3.2.2, pointing the designers to the description of the optional version of this instruction.
■ The beginning of the first sentence of the Engineering Note for dcbt[st] and for the data stream variant of dcbt regarding implications for cache design was corrected to avoid implying that programs are likely to contain these instructions.
■ The table showing the TBR encodings for mftb was reformatted to match that for mfspr in Book I. In particular, the “Privileged” column was deleted (nothing in Book II is privileged).
■ The title of Section 5.2.1.1 was made singular.</td><td>4, 18, 30, 34-36</td></tr>
<tr><td>Remove references to “direct-store segments” and “ordinary segments”.</td><td>RFC02001.</td><td>2, 17</td></tr>
<tr><td>■ Add “lightweight” variant of the sync instruction (lwsync).
■ Remove the vsync instruction from the architecture.
■ State that the architecture is likely to be changed in the future to permit a dcbf instruction to clear the reservation on the processor executing the instruction.
■ Move the Book I material about storage synchronization to Book II.</td><td>RFC02002 and Correspondence of 26 Dec. '98. Material that was moved from Book I to Book II (Book I's five Storage Synchronization instructions and Section D.1) has change bars only where where it is modified. In addition the following changes were made.
■ In Section 1.5, isync was removed from the bulleted list of Cache Management instructions because this RFC moves isync out of the “Cache Management Instructions” section.
■ In Section 3.3.3 in the first bullet of the second paragraph of the second Programming Note of the sync instruction description, “reference and change recording” was changed to “reference, change, and tag set recording”.
■ For clarity, “in the PowerPC AS Virtual Environment Architecture” was inserted in the new paragraph for Appendix E.</td><td>6-12, 16-17, 21-27, 39-51</td></tr>
<tr><td>Remove references to address compare from Book II. Also, remove the statement that eciwx and ecowx can invoke the system data storage error handler only if E=0.</td><td>RFC02003.</td><td>16, 34</td></tr>
<tr><td>Define a second virtual page size that is implementation-dependent and remove a reference to “BAT”.</td><td>RFC02005. In addition, for completeness “or instructions” was added near the end of the fourth paragraph of Section 1.3.</td><td>2, 15, 37</td></tr>
<tr><td>Eliminate Firm Consistency.</td><td>RFC02009.</td><td>2, 6-7</td></tr>
</table>

change	reason	page
Describe ***dcbz*** in terms of architecture rather than implementation. Eliminate ***dcba*** and ***dcbi***.	RFC02010. In addition the following changes were made. ■ ***dcbz*** instruction description: — RTL was added. — Mention of tag bits was removed from the second paragraph. (The RTL covers it, as for *Store* instructions.) — The paragraph about Memory Coherence was deleted, because it applies to all instructions that alter storage. — The last paragraph of the Engineering Note was corrected. — Minor wording changes were made, for consistency with wording elsewhere in the Books. ■ The fact that ***stwcx.*** and ***stdcx.*** clear the associated tag bits was added to their RTL.	3, 10, 15-16, 19, 36, 49-51
Make the following changes. ■ Clarify that ***lq***, ***stq***, ***ldarx***, and ***stdcx.***, as well as ***lwarx*** and ***stwcx.***, may cause the system data storage error handler to be invoked if they specify a location that is Write Through Required or Caching Inhibited. ■ State that performance is likely to be poor for non-atomic accesses to storage that is Write Through Required or Caching Inhibited. ■ Rewrite Section 2.1 and allow instruction restart in more cases.	RFC02011 and Correspondence of 18 Dec. '98. In addition the following changes were made. ■ A new paragraph was added at the beginning of Section 2.1, similar to the paragraph that the RFC adds to the section entitled "Performing Operations Out-of-Order" in Book III. ■ Mention of the fact that *Load And Reserve* and *Store Conditional* can cause the system alignment error handler to be invoked if they specify a location that is Write Through Required or Caching Inhibited was added to the introduction to Section 3.3.2.	4, 13-16, 22, 37
Make the following changes. ■ Make Little-Endian optional. ■ Make Write Through Required Optional. ■ Do not allow mismatched Write Through Required attributes for a page. ■ Remove the Architecture Note in Section 1.7 (about aliasing) and in the ***icbi*** instruction description in Section 3.2.1 (about a COBRA 4 deviation).	RFC02014. In addition the following changes were made. ■ In Section 1.4 an inappropriate bullet concerning a 32-bit implementation was deleted. ■ "Byte Ordering" was removed from the title of Section 5.3, and the first paragraph of the section was reworded for consistency with wording elsewhere in the Book. ■ "Amazon" was changed to "PowerPC AS" throughout the Book, without change bars.	3-6, 13, 17, 37

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
Clarify that ***dcbz*** and ***dcba*** need not be QW atomic but must set tag bit(s) to zero atomically with each access that modifies byte(s) within the tag block.	Amazon RFC 373	3

change	reason	page
Correct statement of operand placement on performance for 2 byte integers in Little-Endian to be consistent with PowerPC RFC00236.	Amazon RFC 368	37
Make minor changes including: ■ Eliminate the definition of “mechanism” in Section Chapter 1. ■ Add ***dcba*** in the list of new instructions in Appendix D.	compatibility with PowerPC and obvious omissions	1, 49
Start phasing Block Address Translation (BAT) out of the architecture.	RFC00249.	13-14
Provide support for 4 KB pages and one larger page size.	RFC00248 as rewritten by Correspondence of 9 Dec. '97.	15

Changes as of 1998/03/27 Version 1.06

change	reason	page
Make support of *Load And Reserve* and *Store Conditional* to Caching Inhibited storage optional, in the “being phased out of the architecture” category of optionality.	RFC00245 as amended at June PAWG meeting.	4, 9
Remove the Engineering Note in ***vsync*** description that points out a difference between ***sync*** and ***vsync***. Add an example of a store-load sequence that can be ordered by ***vsync***.	Amazon RFC 355	old ***vsync*** section, 41, old “Sync' w/o a Lock” section
Change mention of “Rochester Future Systems” to “AS/400”	Amazon RFC 354	6
Add an Engineering Note to warn against giving a cache block to another processor after the block is partially modified by ***dcbz*** or ***dcba***	Amazon RFC 353	19, old ***dcba*** section
Give ***vsync*** the same cumulative property for storage accesses that RFC00233 gives to ***eieio*** and ***sync***	Amazon RFC 352	6
For GP, change atomicity rules to be equivalent to PowerPC except for quadword and tag atomicity. ■ ***lhbrx***, ***lwbrx***, ***sthbrx***, ***stwbrx***, instructions are now atomic in *tags active* mode ■ *Load* and *Store* instructions whose operand is wholly contained within an aligned DW are no longer atomic ■ floating-point *Load* and *Store* instructions with aligned operands are atomic in *tags active* mode	Amazon RFC 351	3, 3
Add an Architecture Note saying that due to TLB array hardware errors and multithreading, Amazon, but not PowerPC, allows instruction restart for accesses to storage that is not Guarded.	Amazon RFC 345	14

change	reason	page
Require E=DS accesses to aligned doublewords	Amazon RFC 325	14
Clarify that “ordinary segments” are used in *tags active* mode and “segments” in *tags inactive* mode.	Amazon RFC 323	2
Move Programming Note in Chapter 2 to Book III, *PowerPC AS Operating Environment Architecture* “Optional Facilities and Instructions”	Amazon RFC 309	14
Explain that “treated as a *Load*(*Store*)” includes address compare.	Amazon RFC 303	16
Add an Architecture Note stating that a COBRA 4 deviation requires that if a ***mtmsrd*** instruction is followed by an ***isync*** and the ***isync*** is one or two instructions in the sequential instruction stream before an ***icbi***, then a ***sync*** instruction must exist between the ***isync*** and ***icbi***.	Amazon RFC 298	17
Reorganize WIMG description, and remove redundant descriptions of *Cache Management* instructions.	RFC00242 and Correspondence of 9 Oct., 29 Oct., and 14 Nov. '96, as amended at 21 Nov. '96 PAWG conference call.	1, 4ff, 9-11, 15-27, 33, old ***dcba*** section

change	reason	page
Redefine ***sync*** to make it a memory barrier, state the extent to which dependencies order storage accesses, and make ***eieio***'s memory barrier cumulative for accesses in its second set.	RFC00233 and Correspondence of 7 Nov. '96. In addition the following changes were made. ■ At the end of the paragraph before the Engineering Note in Section 1.7.2, “accesses by” was changed to “accesses caused by” for consistency with changes made by the RFC. ■ In the Engineering Note in Section 3.2.1, “or of the corresponding implementation-specific sequence” was changed to “and of any corresponding system-specific sequence” for consistency with changes made to this section by the RFC00242 Correspondence dated 14 Nov. '96. ■ A few comments in the code sequences in Sections 3.2.1, B.2.2.1, and B.2.2.2 were abbreviated slightly to make them fit. ■ In the first sentence after the code sequence in Section B.2.1.2, “GPR 5” was used instead of “r5” for consistency with the paragraph before the sequence. ■ Terminology used by the RFC regarding storage control attributes was changed as needed for consistency with RFC00242. The cases that required more than simple substitution or minor rewording are the following. — second paragraph of Section 1.7.1 Because “Memory Coherence Not Required” is no longer considered a storage control attribute, “if any” was inserted near the end of the last sentence (and “Memory Coherence” was changed to “Memory Coherence Required”). — first paragraph of Section B.2.2.1 The parenthesized material in the first sentence was changed, and the second sentence was reworded substantially. The RFC's second sentence is both too strong (the Guarded attribute is irrelevant if the storage is in ***eieio***'s second set) and too weak (***sync*** must be used unless the storage is in ***eieio***'s second set, because cumulative ordering is required). — title of Section B.2.2.2 “using ***eieio***” was used instead of the RFC's parenthesized material. That material is incomplete in omitting the Write Through Required attribute, and mentioning both that and the Caching Inhibited attribute would make the title too long.	1-3, 6-11, 15-27, 33ff, 41ff
Permit aliasing of ***dcbi*** as ***dcbf***, and start phasing ***dcbi*** out of the architecture.	RFC00234.	10, 11, 15
Make Little-Endian performance guidelines equivalent to PowerPC (but not Amazon) Big-Endian.	RFC00236.	13

change	reason	page
Make various clarifications regarding instruction restart.	RFC00237 as amended at Oct. PAWG meeting. In addition the following changes were made, both in Section 2.1. ■ The phrase “in which the instruction may be restarted” was added to the Engineering Note, because the Note does not apply to aligned single-register accesses. ■ The title of Book III was retained in the Programming Note (the RFC wrongly states that the title appears in the preceding paragraph).	13, 14
Clarify that the effective address specified by ***icbi*** is translated as for a *Load* instruction.	RFC00238 as amended at Oct. PAWG meeting.	17
Make miscellaneous clarifications in Book II.	RFC00240 as amended at Oct. PAWG meeting.	18, 34
Say that pages are aligned.	RFC00177 and Correspondence of 23 March '96.	2
Start to phase *tags inactive* direct-store out of the architecture.	RFC00220. In addition, in the Engineering Note on p. 9 “they can” was changed to “the storage subsystem can”, instead of to “it can” as proposed in the RFC, to avoid ambiguity of “it”. (The Engineering Note in Section 2.2 in Version 1.07 was moved by RFC00217 to Section 1.4; the changes proposed for that Note in RFC00220 have been made in the new section.)	2, 3, old Chapter 1 cache op sections, 13, 14, 17-20, 9
Require Caching Inhibited Guarded stores to be ordered.	RFC00217 as amended at March PAWG meeting. In addition, for correctness, a clause covering load/store combining operations was added at the end of the paragraph added to the ***eieio*** Programming Note on p. 27.	3, 6, old Chapter 1 ***eieio*** section, 14, 19, 27, 34
Add new *Cache Management* instruction *Data Cache Block Allocate* (***dcba***).	RFC00228 and Correspondence of 10 May '96. In addition the following changes were made. ■ A sentence was added at the end of the introduction to Chapter 5 to match a corresponding sentence in the introduction to the “Optional” chapters of the other Books. ■ Because the new sentence might suggest that an implementation could provide one of the External Control instructions and not the other, clarification that this is not permitted was added at the end of that sentence and at the end of the first paragraph of Section 5.1. ■ In the second Engineering Note in the ***dcba*** instruction description, the word “all” in “all processors” was italicized for emphasis. ■ ***dcba*** was added at the beginning of the table in Appendix E, instead of before ***dcbz*** as proposed in the RFC, because the table lists instructions alphabetically by mnemonic.	3, old Chapter 1 ***sync*** section, 6+1, 10, 15, 17-20, 27, 33, 35, 49, 51
Eliminate reference to Book III sentence that RFC deletes. Clarify that ***sync*** waits for Reference and Change bit updates to be visible to all processors and mechanisms.	RFC00226.	old Chapter 1 ***sync*** section
Add Programming Note about self-modifying code on MP with combined caches.	RFC00200.	17

change	reason	page
State in their instruction descriptions that ***eciwx*** and ***ecowx*** are optional.	RFC00218. The “Optional Facilities and Instructions” appendix was made a chapter, as agreed at the March PAWG meeting; this necessitated changing “appendix” to “chapter” in several places.	34
Incorporate minor changes from the Morgan Kaufmann book. All such changes that seem desirable have now been made. Very minor changes (e.g., fixing grammatical errors) are not marked with change bars.	Agreed in discussion of RFC00173 at Nov. '94 PAWG meeting.	various
Correct the “Approval Process” description.	Correspondence of 27 Oct. '94.	iii
Clarify conditions under which a cache block is considered “modified”.	RFC00201, correspondence of 24 Oct. '94, and amendment from Nov. PAWG meeting.	old “Coherence [Not] Required” sections, 18
State in Book II that WIMG bits have meaning only when the effective address is translated.	RFC00208.	4
Correct four minor errors.	Error Notice of 27 Oct. '94, Book II items 1-4.	old Chapter 1 ***dcbz*** section, 13, 18, 31
Clarify that ***isync*** does not wait for storage accesses to be performed.	RFC00199 and correspondence of 25 Oct. '94.	21, 27
Use “performed” vs. “executed” consistently for loads and stores.	RFC00205.	18
Clarify “monotonically increasing” in Programming Notes for Time Base.	RFC00206.	29
Clarify paging implications of ***eciwx*** and ***ecowx***.	RFC00180.	33
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. Storage Model

1.1 Definitions and Notation

The following definitions, in addition to those specified in Book I, are used in this Book.

- **processor**
  A hardware component that executes the PowerPC AS instructions specified in a program.
- **system**
  A combination of processors, storage, and associated mechanisms that is capable of executing programs. Sometimes the reference to system includes services provided by the operating system.
- **main storage**
  The level of the storage hierarchy in which all storage state is visible to all processors and mechanisms in the system.
- **instruction storage**
  The view of storage as seen by the mechanism that fetches instructions.
- **data storage**
  The view of storage as seen by a *Storage Access* or *Cache Management* instruction.
- **program order**
  The execution of instructions in the order required by the sequential execution model (see Book I, *PowerPC AS User Instruction Set Architecture*).
- **storage location**
  One or more sequential bytes of storage beginning at the address specified by a *Storage Access* or *Cache Management* instruction or by the instruction fetching mechanism. The number of bytes comprising the location depends on the type of instruction being executed, or is four for instruction fetching.
- **storage access**
  An access to a storage location caused by executing a *Storage Access* or *Cache Management* instruction ("data access") or by fetching an instruction, or an implicit access that occurs as a side effect of such an access (e.g., to translate the effective address).
- **uniprocessor**
  A system that contains one PowerPC AS processor.
- **multiprocessor**
  A system that contains two or more PowerPC AS processors.
- **shared storage multiprocessor**
  A multiprocessor that contains some common storage, which all the PowerPC AS processors in the system can access.
- **performed**
  A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently). An instruction cache block invalidation by P1 is performed with respect to P2 when an instruction fetch by P2 will not be satisfied from the copy of the block that existed in its instruction cache when the instruction causing the invalidation was executed, and similarly for a data cache block invalidation. The preceding

definitions apply regardless of whether P1 and P2 are the same entity.

- **page**
  An aligned unit of storage for which protection and control attributes are independently specifiable and for which reference and change status are independently recorded. Two virtual page sizes are supported simultaneously, 4 KB and a larger size. The larger size is an implementation-dependent power of 2 (bytes). Real pages are always 4 KB.
- **block**
  The aligned unit of storage operated on by each *Cache Management* instruction. The size of a block can vary by instruction and by implementation. The maximum block size is 4 KB.
- **aligned storage access**
  A load or store is aligned if the address of the target storage location is a multiple of the size of the transfer effected by the instruction.

1.2 Introduction

The PowerPC AS User Instruction Set Architecture, discussed in Book I, defines storage as a linear array of bytes indexed from 0 to a maximum of $2^{64} - 1$. Each byte is identified by its index, called its address, and each byte contains a value. This information is sufficient to allow the programming of applications that require no special features of any particular system environment. The PowerPC AS Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The PowerPC AS Virtual Environment Architecture, in conjunction with services based on the PowerPC AS Operating Environment Architecture (see Book III) and provided by the operating system, permits explicit control of this expanded storage model. A simple model for sequential execution allows at most one storage access to be performed at a time and requires that all storage accesses appear to be performed in program order. In contrast to this simple model, the PowerPC AS architecture specifies a relaxed model of storage consistency. In a multiprocessor system that allows multiple copies of a storage location, aggressive implementations of the architecture can permit intervals of time during which different copies of a storage location have different values. This chapter describes features of the PowerPC AS architecture that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The PowerPC AS system implements a virtual storage model for applications. For *tags inactive* mode and for PLS addresses in *tags active* mode (see Book III, *PowerPC AS Operating Environment Architecture*), this means that a combination of hardware and software can present a storage model that allows applications to exist within a “virtual” address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of “effective address” (EA) space, subject to limitations imposed by the operating system. In a typical PowerPC AS system in *tags inactive* mode and for PLS addresses in *tags active* mode, each program's EA space is a subset of a larger “virtual address” (VA) space managed by the operating system.

Each effective address is translated to a real address (i.e., to an address of a byte in real storage or on an I/O device) before being used to access storage. The hardware accomplishes this, using the address translation mechanism described in Book III. The operating system manages the real (physical) storage resources of the system, by setting up the tables and other information used by the hardware address translation mechanism.

Book II deals primarily with effective addresses that are in “segments” translated by the “address translation mechanism” (see Book III). Each such effective address lies in a “virtual page”, which is mapped to a “real page” (4 KB virtual page) or to a contiguous sequence of real pages (large virtual page) before data or instructions in the virtual page are accessed.

In general, real storage may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map a sufficient set of virtual pages of the applications. If a sufficient set is maintained, “paging” activity is minimized. If not, performance degradation is likely.

The operating system can support restricted access to virtual pages (including read/write, read only, and no access: see Book III), based on system standards (e.g., program code might be read only) and application requests.

1.4 Single-Copy Atomicity

An access is *single-copy atomic*, or simply *atomic*, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus *serialized*: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

In PowerPC AS the following single-register accesses are always atomic:

- byte accesses (all bytes are aligned on byte boundaries)
- halfword accesses aligned on halfword boundaries
- word accesses aligned on word boundaries
- doubleword accesses aligned on doubleword boundaries
- quadword accesses aligned on quadword boundaries in *tags active* mode and only for the following instructions:
  - ***lq***
  - ***stq***

  Quadword atomicity applies only to storage that is neither Write Through Required nor Caching Inhibited. The tag bit(s) are part of every quadword atomic access the same as every other bit in the quadword.

No other accesses are guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic.

- any *Load* or *Store* instruction for which the operand is unaligned
- ***lmw, stmw, lswi, lswx, stswi, stswx***
- ***lmd, stmd, lsdi, lsdx, stsdi, stsdx***

†

- any *Cache Management* instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accesses. The number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed.

| For ***dcbz*** and for *Store* instructions other than ***stq***, the tag bit for every tag block that has a byte modified is set to zero as part of the atomic access that modified | the byte(s) within the tag block. If a ***dcbz*** or *Store* instruction causes multiple disjoint atomic accesses within a tag block, the tag bit of the tag block is set to zero as part of each atomic access.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors execute atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
2. When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors execute stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors execute stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.
6. When a load and a store with the same target location can be executed simultaneously, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

Engineering Note

Atomicity of storage accesses is provided by the processor in conjunction with the storage subsystem. The processor must provide a storage subsystem interface that is sufficient to allow a storage subsystem to meet the atomicity requirements specified here.

Multiprocessor implementations with doubleword tags must give the appearance of having quadword tag atomicity.

For initial hardware debug it is often useful to run with cache disabled. In some ways cache disabled mode is similar to Caching Inhibited storage. Although quadword atomicity is not required for storage that is Caching Inhibited, support for quadword atomicity with cache disabled should be considered.

1.5 Cache Model

A cache model in which there is one cache for instructions and another cache for data is called a “Harvard-style” cache. This is the model assumed by the PowerPC AS Architecture, e.g., in the descriptions of the *Cache Management* instructions in Section 3.2, “Cache Management Instructions” on page 16. Alternative cache models may be implemented (e.g., a “combined cache” model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (e.g., modifications caused by *Store* instructions).

| A location in the data cache is considered to be modified in that cache if the location has been modified (e.g., by a *Store* instruction) and the modified data have not been been written to main storage.

Cache Management instructions are provided so that programs can manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that will be executed (i.e., when the program modifies data in storage and then attempts to execute the modified data as instructions). The *Cache Management* instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location (see Section 1.6, “Storage Control Attributes”).

The *Cache Management* instructions allow the program to do the following.

- invalidate the copy of storage in an instruction cache block (***icbi***)

†

- † provide a hint that the program will probably soon access a specified data cache block (***dcbt, dcbtst***)
- set the contents of a data cache block to zeros (***dcbz***)
- copy the contents of a modified data cache block to main storage (***dcbst***)
- copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (***dcbf***)

1.6 Storage Control Attributes

Some operating systems may provide a means to allow programs to specify the storage control attributes described in this section. Because the support provided for these attributes by the operating system may vary between systems, the details of the specific system being used must be known before these attributes can be used.

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location, as described below. The storage control attributes are the following.

- Write Through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded

These attributes have meaning only when an effective address is translated by the processor performing the storage access. All combinations of these attributes † are supported except Write Through Required with Caching Inhibited.

Programming Note

The Write Through Required and Caching Inhibited attributes are mutually exclusive because, as described below, the Write Through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not.

Storage that is Write Through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the † ***lq***, ***stq***, ***lwarx***, ***ldarx***, ***stwcx.***, and ***stdcx.*** instructions may cause the system data storage error handler to be invoked if they specify a location in storage having either of these † attributes.

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

1.6.1 Write Through Required

A store to a Write Through Required storage location is performed in main storage. A *Store* instruction that specifies a location in Write Through Required storage may cause additional locations in main storage to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. The store does not cause the block to be considered to be modified in the data cache.

In general, accesses caused by separate *Store* instructions that specify locations in Write Through Required storage may be combined into one access. Such combining does not occur if the *Store* instructions are separated by a ***sync*** instruction or by an ***eieio*** instruction.

1.6.2 Caching Inhibited

An access to a Caching Inhibited storage location is performed in main storage. A *Load* instruction that specifies a location in Caching Inhibited storage may cause additional locations in main storage to be accessed unless the specified location is also Guarded. An instruction fetch from Caching Inhibited storage may cause additional words in main storage to be accessed. No copy of the accessed locations is placed into the caches.

In general, non-overlapping accesses caused by separate *Load* instructions that specify locations in Caching Inhibited storage may be combined into one access, as may non-overlapping accesses caused by separate *Store* instructions that specify locations in Caching Inhibited storage. Such combining does not occur if the *Load* or *Store* instructions are separated by a ***sync*** instruction, or by an ***eieio*** instruction if the storage is also Guarded.

1.6.3 Memory Coherence Required

An access to a Memory Coherence Required storage location is performed coherently, as follows.

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processor and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called *coherence blocks*. Their size is implementation-dependent (see the Book IV, *PowerPC AS Implementation Features* document for the implementation), but is usually larger than a word and often the size of a cache block.

For storage that is not Memory Coherence Required, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this may be system-dependent.

Because the Memory Coherence Required attribute for a given storage location is of little use unless all processors that access the location do so coherently, in statements about Memory Coherence Required storage elsewhere in Books I – III it is generally assumed that the storage has the Memory Coherence Required attribute for all processors that access it.

Programming Note

Operating systems that allow programs to request that storage not be Memory Coherence Required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.

In most systems the default is that all storage is Memory Coherence Required. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be Memory Coherence Required, and can manage the coherence of that storage by using the ***sync*** instruction, the *Cache Management* instructions, and services provided by the operating system.

Engineering Note

Memory coherence can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to a given location in Memory Coherence Required storage.

A processor observing a storage access initiated by another processor or mechanism must honor the coherence requirements of that access, even if the observing processor last accessed the affected storage location as not Memory Coherence Required.

1.6.4 Guarded

A data access to a Guarded storage location is performed only if either (a) the access is caused by an instruction that is known to be required by the sequential execution model, or (b) the access is a load and the storage location is already in a cache. If the storage is also Caching Inhibited, only the storage location specified by the instruction is accessed; otherwise any storage location in the cache block containing the specified storage location may be accessed.

Instructions are not fetched from virtual storage that is Guarded. If the effective address of the current instruction is in such storage, the system instruction storage error handler is invoked.

Programming Note

In some implementations, instructions may be executed before they are known to be required by the sequential execution model. Because the results of instructions executed in this manner are discarded if it is later determined that those instructions would not have been executed in the sequential execution model, this behavior does not affect most programs.

This behavior does affect programs that access storage locations that are not “well-behaved” (e.g., a storage location that represents a control register on an I/O device that, when accessed, causes the device to perform an operation). To avoid unintended results, programs that access such storage locations should request that the storage be Guarded, and should prevent such storage locations from being in a cache (e.g., by requesting that the storage also be Caching Inhibited).

1.7 Shared Storage

This architecture supports the sharing of storage between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a storage location by one or more programs using different effective addresses. All these cases are considered storage sharing. Storage is shared in blocks that are an integral number of pages.

When the same storage location has different effective addresses, the addresses are said to be *aliases*. Each application can be granted separate access privileges to aliased pages.

Engineering Note

Page level aliasing can be implemented in many ways, for example with real addressed caches, L2 directories, or an external signal to an inverse directory. Each processor implementation will decide on its level of implementation in support of its system requirements.

1.7.1 Storage Access Ordering

The storage model for the ordering of storage accesses is *weakly consistent*. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed when storage is shared by two or more programs.

The order in which the processor performs storage accesses, the order in which those accesses are performed with respect to another processor or mechanism, and the order in which those accesses are performed in main storage may all be different. Several means of enforcing an ordering of storage accesses are provided to allow programs to share storage with other programs, or with mechanisms such as I/O devices. These means are listed below. The phrase “to the extent required by the associated Memory Coherence Required attributes” refers to the Memory Coherence Required attribute, if any, associated with each access.

- If two *Store* instructions specify storage locations that are both Caching Inhibited and Guarded, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.
- If a *Load* instruction depends on the value returned by a preceding *Load* instruction (because the value is used to compute the effective address specified by the second *Load*), the corresponding storage accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated Memory Coherence Required attributes. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first *Load* is ANDed with zero and then added to the effective address specified by the second *Load*).
- When a processor (P1) executes a ***sync***, ***lwsync***, or ***eieio*** instruction a *memory barrier* is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses asso-

| ciated with instructions following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be “cumulative” if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.

† — A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.

† — B includes all applicable storage accesses by any such processor or mechanism that are performed after a *Load* instruction executed by that processor or mechanism has returned † the value stored by a store that is in B.

No ordering should be assumed among the storage accesses caused by a single instruction (i.e, by an instruction for which the access is not atomic), and no means are provided for controlling that order.

Programming Note

Because stores cannot be performed "out-of-order" (see Book III, *PowerPC AS Operating Environment Architecture*), if a *Store* instruction depends on the value returned by a preceding *Load* instruction (because the value returned by the *Load* is used to compute either the effective address specified by the *Store* or the value to be stored), the corresponding storage accesses are performed in program order. The same applies if whether the *Store* instruction is executed depends on a conditional *Branch* instruction that in turn depends on the value returned by a preceding *Load* instruction.

Because an ***isync*** instruction prevents the execution of instructions following the ***isync*** until instructions preceding the ***isync*** have completed, if an ***isync*** follows a conditional *Branch* instruction that depends on the value returned by a preceding *Load* instruction, the load on which the *Branch* depends is performed before any loads caused by instructions following the ***isync***. This applies even if the effects of the "dependency" are independent of the value loaded (e.g., the value is compared to itself and the *Branch* tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.

With the exception of the cases described above and earlier in this section, data dependencies and control dependencies do not order storage accesses. Examples include the following.

- If a *Load* instruction specifies the same storage location as a preceding *Store* instruction and the location is in storage that is not Caching Inhibited, the load may be satisfied from a "store queue" (a buffer into which the processor places stored values before presenting them to the storage subsystem), and not be visible to other processors and mechanisms. A consequence is that if a subsequent *Store* depends on the value returned by the *Load*, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a *Store Conditional* instruction may complete before its store has been performed, a conditional *Branch* instruction that depends on the CR0 value set by a *Store Conditional* instruction does not order the *Store Conditional*'s store with respect to storage accesses caused by instructions that follow the *Branch*.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (e.g., branches) do not order storage accesses except as described above. For example, when a subroutine returns to its caller the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Because processors may implement nonarchitected duplicates of architected resources (e.g., GPRs, CR fields, and the Link Register), resource dependencies (e.g., specification of the same target register for two *Load* instructions) do not order storage accesses.

Examples of correct uses of dependencies, ***sync***, ***lwsync***, and ***eieio*** to order storage accesses can be found in Appendix B, "Programming Examples for Sharing Storage" on page 41.

Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before storage accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same storage location, if the location is in storage that is Memory Coherence Required the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is in storage that is Caching Inhibited.

Because accesses to storage that is Caching Inhibited are performed in main storage, memory barriers and dependencies on *Load* instructions order such accesses with respect to any processor or mechanism even if the storage is not Memory Coherence Required.

Programming Note

The first example below illustrates cumulative ordering of storage accesses preceding a memory barrier, and the second illustrates cumulative ordering of storage accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

Processor A: stores the value 1 to location X

Processor B: loads from location X obtaining the value 1, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor C: loads from location Y obtaining the value 2, executes a ***sync*** instruction, then loads from location X

Example 2:

Processor A: stores the value 1 to location X, executes a ***sync*** instruction, then stores the value 2 to location Y

Processor B: loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z

Processor C: loads from location Z obtaining the value 3, executes a ***sync*** instruction, then loads from location X

In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

Engineering Note

It is permissible to perform a dependent load before the load on which it depends, if software accessing shared storage cannot tell the difference.

It is always permissible to prefetch a data cache block from non-Guarded storage based on predicting the effective address specified by a *Load* or *Store* instruction.

Engineering Note

| The correct operation of ***sync***, ***lwsync***, and ***eieio*** depends on both the processor and the storage subsystem.

The definition of memory barriers is not intended to preclude address pipelining. If two applicable *Storage Access* instructions are separated by
| ***sync***, ***lwsync***, or ***eieio***, it is permissible for the address associated with the second instruction to be presented to a given level of the storage hierarchy before the data access caused by the first instruction has completed at that level. However, if such pipelining is done, the processor must provide sufficient information that the storage subsystem can keep the storage accesses in the correct order, and the storage subsystem must do so.

| Memory barriers need not order the following:

| ■ the prefetching of cache blocks
| ■ the casting out of cache blocks
| ■ consistency operations initiated by other
| processors

†
1.7.2 Atomic Update

The *Load And Reserve* and *Store Conditional* instructions together permit atomic update of a storage location. There are word and doubleword forms of each of these instructions. Described here is the operation of the word forms ***lwarx*** and ***stwcx.***; operation of the doubleword forms ***ldarx*** and ***stdcx.*** is the same except for obvious substitutions.

|

The ***lwarx*** instruction is a load from a word-aligned
† location that has two side effects. Both of these side
† effects occur at the same time that the load is per-
† formed.

1. A reservation for a subsequent ***stwcx.*** instruction is created.

2. The storage coherence mechanism is notified that
† a reservation exists for the storage location spec-
† ified by the ***lwarx***.

The ***stwcx.*** instruction is a store to a word-aligned location that is conditioned on the existence of the reservation created by the ***lwarx*** and on whether the same storage location is specified by both instructions. To emulate an atomic operation with these instructions, it is necessary that both the ***lwarx***
† and the ***stwcx.*** specify the same storage location.

†

A ***stwcx.*** performs a store to the target storage location only if the storage location specified by the ***lwarx*** that established the reservation has not been stored into by another processor or mechanism since the reservation was created. If the storage locations specified by the two instructions differ, the store is not necessarily performed.

A ***stwcx.*** that performs its store is said to "succeed".

Examples of the use of ***lwarx*** and ***stwcx.*** are given in Appendix B, "Programming Examples for Sharing Storage" on page 41.

A successful ***stwcx.*** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or ***lwarx*** from the given location on another processor may return a "stale" value. However, a subsequent ***lwarx*** from the given location on the other processor followed by a successful ***stwcx.*** on that processor is guaranteed to have returned the value stored by the first processor's ***stwcx.*** (in the absence of other stores to the given location).

Programming Note

The store caused by a successful ***stwcx.*** is ordered, by a dependence on the reservation, with respect to the load caused by the ***lwarx*** that established the reservation, such that the two storage accesses are performed in program order with respect to any processor or mechanism.

Engineering Note

Both ***lwarx*** and ***stwcx.*** have a data dependence on the processor reservation resource.

1.7.2.1 Reservations

The ability to emulate an atomic operation using ***lwarx*** and ***stwcx.*** is based on the conditional behavior of ***stwcx.***, the reservation created by ***lwarx***, and the clearing of that reservation if the target location is modified by another processor or mechanism before the ***stwcx.*** performs its store.

A reservation is held on an aligned unit of real storage called a *reservation granule*. The size of the reservation granule is 2^n bytes, where n is implementation-dependent but is always at least 4 (thus the minimum reservation granule size is a quadword). The reservation granule associated with effective address EA contains the real address to which EA maps. ("real_addr(EA)" in the RTL for the *Load And Reserve* and *Store Conditional* instructions stands for "real address to which EA maps".)

A processor has at most one reservation at any time. A reservation is established by executing a ***lwarx*** or ***ldarx*** instruction, and is lost (or may be lost, in the case of the fourth bullet) if any of the following occur.

- The processor holding the reservation executes another ***lwarx*** or ***ldarx***: this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any ***stwcx.*** or ***stdcx.***, regardless of whether the specified address matches the address specified by the ***lwarx*** or ***ldarx*** that established the reservation.
- Some other processor executes a *Store* or ***dcbz*** to the same reservation granule, or modifies a Reference, Change, or Tag Set bit (see Book III, *PowerPC AS Operating Environment Architecture*) in the same reservation granule.
- Some other processor executes a ***dcbtst, dcbst,*** or ***dcbf*** to the same reservation granule: whether the reservation is lost is undefined.
- Some other mechanism modifies a storage location in the same reservation granule.

Interrupts (see Book III, *PowerPC AS Operating Environment Architecture*) do not clear reservations (however, system software invoked by interrupts may clear reservations).

Programming Note

One use of ***lwarx*** and ***stwcx.*** is to emulate a "Compare and Swap" primitive like that provided by the IBM System/370 Compare and Swap instruction; see Section B.1, "Atomic Update Primitives" on page 41. A System/370-style Compare and Swap checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The combination of ***lwarx*** and ***stwcx.*** improves on such a Compare and Swap, because the reservation reliably binds the ***lwarx*** and ***stwcx.*** together. The reservation is always lost if the word is modified by another processor or mechanism between the ***lwarx*** and ***stwcx.***, so the ***stwcx.*** never succeeds unless the word has not been stored into (by another processor or mechanism) since the ***lwarx***.

Programming Note

Warning: The architecture is likely to be changed in the future to permit the reservation to be lost if a ***dcbf*** instruction is executed on the processor holding the reservation. Therefore ***dcbf*** instructions should not be placed between a *Load And Reserve* instruction and the subsequent *Store Conditional* instruction.

Programming Note

In general, programming conventions must ensure that ***lwarx*** and ***stwcx.*** specify addresses that match; a ***stwcx.*** should be paired with a specific ***lwarx*** to the same storage location. Situations in which a ***stwcx.*** may erroneously be issued after some ***lwarx*** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a ***lwarx*** and before the paired ***stwcx.***. The ***stwcx.*** in the new context might succeed, which is not what was intended by the programmer. Such a situation must be prevented by executing a ***stwcx.*** or ***stdcx.*** that specifies a dummy writable aligned location as part of the context switch; see the section entitled “Interrupt Processing” in Book III.

Programming Note

Because the reservation is lost if another processor stores anywhere in the reservation granule, lock words (or doublewords) should be allocated such that few such stores occur, other than perhaps to the lock word itself. (Stores by other processors to the lock word result from contention for the lock, and are an expected consequence of using locks to control access to shared storage; stores to other locations in the reservation granule can cause needless reservation loss.) Such allocation can most easily be accomplished by allocating an entire reservation granule for the lock and wasting all but one word. Because reservation granule size is implementation-dependent, portable code must do such allocation dynamically.

Similar considerations apply to other data that are shared directly using ***lwarx*** and ***stwcx.*** (e.g., pointers in certain linked lists; see Section B.3, “List Insertion” on page 45).

Engineering Note

Reservations must take part in storage coherence. A reservation must be cleared if another processor receives authorization from the coherence mechanism to store to the reservation granule.

If an implementation continues to hold a reservation when the cache block containing the reservation granule (here called the “reserved block”) is evicted, the reservation must continue to participate in the coherence protocol. In a snooping implementation, it must join in snooping. In a directory-based implementation, it must register its interest in the reserved block with the directory (shared-read access).

Note: The implementation technique described in the next paragraph will become possible if and when the architecture is changed to permit ***dcbf*** to clear the reservation on the processor executing the ***dcbf*** instruction.

If an implementation demands that the reserved block be held in the cache, one way to satisfy the architectural requirements is the following. The implementation must be able to protect that block from eviction except by explicit invalidation (e.g., execution of ***dcbf***) by the processor holding the reservation, and by cross-invalidates received from other processors, as long as the reservation persists. Caches in such an implementation must be sufficiently associative that the machine can continue to run with eviction of the reserved block inhibited.

1.7.2.2 Forward Progress

Forward progress in loops that use ***lwarx*** and ***stwcx.*** is achieved by a cooperative effort among hardware, system software, and application software.

The architecture guarantees that when a processor executes a ***lwarx*** to obtain a reservation for location X and then a ***stwcx.*** to store a value to location X, either

1. the ***stwcx.*** succeeds and the value is written to location X, or
2. the ***stwcx.*** fails because some other processor or mechanism modified location X, or
3. the ***stwcx.*** fails because the processor's reservation was lost for some other reason.

In Cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real storage. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in Case 3. While the architecture alone cannot provide such a guarantee, the characteristics listed in Cases 1 and 2 are necessary conditions for any forward progress guarantee. An implementation and operating system can build on them to provide such a guarantee.

† **Programming Note**

The architecture does not include a “fairness guarantee”. In competing for a reservation, two processors can indefinitely lock out a third.

†

Chapter 2. Effect of Operand Placement on Performance

The placement (location and alignment) of operands in storage affects relative performance of storage accesses, and may affect it significantly. The best performance is guaranteed if storage operands are aligned. In order to obtain the best performance across the widest range of implementations, the programmer should assume the performance model described in Figure 1 with respect to the placement of storage operands. Performance of accesses varies depending on the following:

1. Operand Size
2. Operand Alignment
3. Crossing no boundary
4. Crossing a cache block boundary
5. Crossing a virtual page boundary
6. Crossing a segment boundary (see Book III, *PowerPC AS Operating Environment Architecture* for a description of storage segments)

The *Load and Store Multiple* instructions are defined to operate only on aligned operands. The *Move Assist* instructions have no alignment requirements.

Architecture Note

All processors will provide at a minimum the level of support implied by Figure 1.

Operand		**Boundary Crossing**			
Size	**Byte Align.**	**None**	**Cache Block**	**Virtual Page[2]**	**Seg.**
Integer					
8 Byte	8 4 < 4	optimal good good	– good good	– good good	– poor poor
4 Byte	4 < 4	optimal good	– good	– good	– poor
2 Byte	2 < 2	optimal good	– good	– good	– poor
1 Byte	1	optimal	–	–	–
lmw, stmw	4	good	good	good	poor
lmd, stmd	8	good	good	good	poor
string		good	good	good	poor
Float					
8 Byte	8 4 < 4	optimal good poor	– good poor	– poor poor	– poor poor
4 Byte	4 < 4	optimal poor	– poor	– poor	– poor

[1] If an instruction causes an access that is not atomic and any portion of the operand is in storage that is Write Through Required or Caching Inhibited, performance is likely to be poor.

[2] If the storage operand spans two virtual pages that have different storage control attributes, performance is likely to be poor.

Figure 1. Performance effects of storage operand placement

2.1 Instruction Restart

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

The following instructions are never restarted after having accessed any portion of the storage operand (unless the instruction causes a "Data Address Compare match" or a "Data Address Breakpoint match", for which the corresponding rules are given in Book III).

1. A *Store* instruction that causes an atomic access
2. A *Load* instruction that causes an atomic access to storage that is both Caching Inhibited and Guarded

Any other *Load* or *Store* instruction may be partially executed and then aborted after having accessed a portion of the storage operand, and then re-executed (i.e., restarted, by the processor or the operating system). If an instruction is partially executed, the contents of registers are preserved to the extent that the correct result will be produced when the instruction is re-executed.

Programming Note

There are many events that might cause a *Load* or *Store* instruction to be restarted. For example, a hardware error may cause execution of the instruction to be aborted after part of the access has been performed, and the recovery operation could then cause the aborted instruction to be re-executed.

When an instruction is aborted after being partially executed, the contents of the instruction pointer indicate that the instruction has not been executed, however, the contents of some registers may have been altered and some bytes within the storage operand may have been accessed. The following are examples of an instruction being partially executed and altering the program state even though it appears that the instruction has not been executed.

1. *Load Multiple*, *Load String*: Some registers in the range of registers to be loaded may have been altered.
2. Any *Store* instruction, ***dcbz***: Some bytes of the storage operand may have been altered.
3. Any floating-point *Load* instruction: The target register (FRT) may have been altered.

Chapter 3. Storage Control Instructions

3.1 Parameters Useful to Application Programs

It is suggested that the operating system provide a service that allows an application program to obtain the following information.

1. The two virtual page sizes
2. Coherence block size
3. Granule sizes for reservations
4. An indication of the cache model implemented (e.g., Harvard-style cache, combined cache)
5. Instruction cache size
6. Data cache size
7. Instruction cache line size (see Book IV, *PowerPC AS Implementation Features*)
8. Data cache line size (see Book IV)
9. Block size for ***icbi***
10. Block size for ***dcbt*** and ***dcbtst***
11. Block size for ***dcbz***, ***dcbst***, and ***dcbf***
12. Instruction cache associativity
13. Data cache associativity
14. Factors for converting the Time Base to seconds

If the caches are combined, the same value should be given for an instruction cache attribute and the corresponding data cache attribute.

Architecture Note

All processors in a symmetric multiprocessor must be identical with respect to the cache model, the coherence block size, and the reservation granule sizes.

3.2 Cache Management Instructions

The *Cache Management* instructions obey the sequential execution model except as described in Section 3.2.1, “Instruction Cache Instruction” on page 17.

In the instruction descriptions the statements “this instruction is treated as a *Load*” and “this instruction is treated as a *Store*” mean that the instruction is treated as a *Load* (*Store*) from (to) the addressed byte with respect to address translation, storage protection, reference and change recording, and the storage access ordering described in Section 1.7.1, “Storage Access Ordering” on page 6.

Engineering Note

An example of the requirements of the sequential execution model with respect to *Cache Management* instructions is that a *Load* instruction that specifies a storage location in the block specified by a preceding ***dcbf*** instruction must be satisfied from main storage (if the location is in storage that is not Memory Coherence Required) or from coherent storage (if the location is in storage that is Memory Coherence Required), and not from the copy of the location that existed in the cache when the ***dcbf*** instruction was executed.

Similar requirements apply to cache reload buffers. For example, if a cache reload request for a given instruction cache block is pending when an ***icbi*** instruction is executed specifying the same block, the results of the reload request must not be used to satisfy a subsequent instruction fetch.

An example of the requirements of data dependencies with respect to *Cache Management* instructions is that if a ***dcbf*** instruction depends on the value returned by a preceding *Load* instruction, the invalidation caused by the ***dcbf*** must be performed after the load has been performed.

Engineering Note

If, at any level of the storage hierarchy, a combined cache is implemented such that locations in that cache lack an indication of whether they were fetched as data or as instructions, the locations must be treated as if they were fetched as data and must not be treated as if they were fetched as instructions. E.g., ***dcbf*** must flush and invalidate them, and ***icbi*** must not invalidate them. (Permitting ***icbi*** to invalidate a block that was fetched as data would permit it to invalidate modified data, creating a security and data integrity exposure.)

† 3.2.1 Instruction Cache Instruction

The instruction cache is not necessarily kept consistent with the data cache or with main storage. When instructions are modified by processors or by other mechanisms, software must ensure that the instruction cache is made consistent with data storage and that the modifications are made visible to the instruction fetching mechanism. The following instruction sequence can be used to accomplish this when the instructions being modified are in storage that is Memory Coherence Required and one program both modifies the instructions and executes them. (Additional synchronization is needed when one program modifies instructions that another program will execute.) In this sequence, location “instr” is assumed to contain instructions that have been modified.

```
dcbst  instr   #update block in main storage
sync           #order update before invalidat'n
icbi   instr   #invalidate copy in instr cache
isync          #discard prefetched instructions
```

Programming Note

Because the optimal instruction sequence may vary between systems, many operating systems will provide a system service to perform the function described above.

Engineering Note

Correct operation of the instruction sequence shown above, and of any corresponding system-specific sequence, may require that an instruction fetch request not bypass a writeback of the same storage location caused by the sequence (including a writeback by another processor).

Instruction Cache Block Invalidate X-form †

icbi RA,RB

31	///	RA	RB	982	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

† If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches.

† If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is invalidated in that instruction cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 3.2), except that reference and change recording need not be done.

Special Registers Altered:
None

Programming Note

As stated above, the effective address is translated using translation resources used for data accesses, even though the block being invalidated was copied into the instruction cache based on translation resources used for instruction fetches (see Book III, *PowerPC AS Operating Environment Architecture*).

3.2.2 Data Cache Instructions

Data Cache Block Touch X-form

dcbt RA,RB

31	///	RA	RB	278	/
0	6	11	16	21	31

Let the effective address (EA) be the sum $(RA|0)+_{tea}(RB)$.

† The ***dcbt*** instruction provides a hint that the program will probably soon load from the block containing the byte addressed by EA. The hint is ignored if the block is Caching Inhibited or Guarded.

The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

This instruction is treated as a *Load* (see Section 3.2), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:
None

Engineering Note

See the description of the optional version of ***dcbt*** in Section 5.2.1.1 for additional information about this instruction.

Programming Note

In response to the hint provided by ***dcbt*** and ***dcbtst***, the processor may prefetch the specified block into the data cache, or take other actions that reduce the latency of subsequent *Load* or *Store* instructions that refer to the block.

Earlier implementations do not necessarily ignore the hint provided by ***dcbt*** and ***dcbtst*** if the specified block is in storage that is Guarded and not Caching Inhibited. Therefore a ***dcbt*** or ***dcbtst*** instruction should not specify an EA in such storage if the program is to be run on such implementations.

Data Cache Block Touch for Store X-form

dcbtst RA,RB

31	///	RA	RB	246	/
0	6	11	16	21	31

Let the effective address (EA) be the sum $(RA|0)+_{tea}(RB)$.

† The ***dcbtst*** instruction provides a hint that the program will probably soon store to the block containing the byte addressed by EA. The hint is ignored if the block is Caching Inhibited or Guarded.

The actions (if any) taken by the processor in response to the hint are not considered to be "caused by" or "associated with" the ***dcbtst*** instruction (e.g., ***dcbtst*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

This instruction is treated as a *Load* (see Section 3.2), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:
None

Engineering Note

Executing ***dcbtst*** does not cause the specified block to be considered to be modified in the data cache.

Engineering Note

† Programs that use ***dcbt*** or ***dcbtst*** are likely to contain multiple instances of the instruction preceding the *Load* or *Store* instructions that refer to the prefetched blocks. In designing the data cache and any associated prefetch buffers, consideration should be given to minimizing the extent to which the prefetched blocks displace other data needed or requested by the program, or are themselves displaced before they are used.

Data Cache Block set to Zero X-form

dcbz RA,RB

[POWER mnemonic: dclz]

31	///	RA	RB	1014	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
n ← block size (bytes)
m ← log2(n)
ea ← EA0:63−m || m0
MEM(ea, n) ← n0x00
MEMtag(EA, n) ← 0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

All bytes in the block containing the byte addressed by EA are set to zero.

This instruction is treated as a *Store* (see Section 3.2).

Special Registers Altered:

None

Programming Note

dcbz does not cause the block to exist in the data cache if the block is in storage that is Caching Inhibited.

For storage that is neither Write Through Required nor Caching Inhibited, ***dcbz*** provides an efficient means of setting blocks of storage to zero. It can be used to initialize large areas of such storage, in a manner that is likely to consume less memory bandwidth than an equivalent sequence of *Store* instructions.

For storage that is either Write Through Required or Caching Inhibited, ***dcbz*** is likely to take significantly longer to execute than an equivalent sequence of *Store* instructions.

See the section entitled “Cache Management Instructions” in Book III, *PowerPC AS Operating Environment Architecture* for additional information about ***dcbz***.

Engineering Note

If the specified block is in storage that is neither Write Through Required nor Caching Inhibited and is not already in the data cache, establishing the block in the data cache without fetching it from main storage may provide the best performance.

If the specified block is in storage that is either Write Through Required or Caching Inhibited, an Alignment exception may be generated.

If ***dcbz*** causes the specified block to be established in the data cache without being fetched from main storage, the contents of any byte of the cache block must not be made available to another processor or mechanism until that byte has been set to zero.

Data Cache Block Store X-form

dcbst RA,RB

31	///	RA	RB	54	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage, additional locations in the block may be written to main storage, and the block ceases to be considered to be modified in that data cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 3.2), except that reference and change recording need not be done.

Special Registers Altered:
None

Data Cache Block Flush X-form

dcbf RA,RB

31	///	RA	RB	86	/
0	6	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

If the block containing the byte addressed by EA is in storage that is Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in storage that is not Memory Coherence Required and a block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main storage and additional locations in the block may be written to main storage. The block is invalidated in the data cache of this processor.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in storage that is Write Through Required or Caching Inhibited.

This instruction is treated as a *Load* (see Section 3.2), except that reference and change recording need not be done.

Special Registers Altered:
None

† 3.3 Synchronization Instructions

† 3.3.1 Instruction Synchronize † Instruction

Instruction Synchronize XL-form

isync

[POWER mnemonic: ics]

19	///	///	///	150	/
0	6	11	16	21	31

† Executing an ***isync*** instruction ensures that all instructions preceding the ***isync*** instruction have completed before the ***isync*** instruction completes, and that no subsequent instructions are initiated until after the ***isync*** instruction completes. It also causes any prefetched instructions to be discarded, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the ***isync*** instruction.

The ***isync*** instruction may complete before storage accesses associated with instructions preceding the ***isync*** instruction have been performed.

This instruction is context synchronizing (see Book III, *PowerPC AS Operating Environment Architecture*).

Special Registers Altered:
None

3.3.2 Load And Reserve and Store Conditional Instructions

The *Load And Reserve* and *Store Conditional* instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location. See Section 1.7.2, “Atomic Update” on page 9 for additional information about these instructions.

The *Load And Reserve* and *Store Conditional* instructions are fixed-point *Storage Access* instructions; see the section entitled “Storage Access Instructions” in Book I, *PowerPC AS User Instruction Set Architecture*.

The storage location specified by the *Load And Reserve* and *Store Conditional* instructions must be in storage that is Memory Coherence Required if the location may be modified by other processors or mechanisms. If the specified location is in storage that is Write Through Required or Caching Inhibited, the system data storage error handler or the system alignment error handler is invoked.

Programming Note

The Memory Coherence Required attribute on other processors and mechanisms ensures that their stores to the reservation granule will cause the reservation created by the *Load And Reserve* instruction to be lost.

Programming Note

Because the *Load And Reserve* and *Store Conditional* instructions have implementation dependencies (e.g., the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (Test and Set, Compare and Swap, locking, etc.; see Appendix B) that are needed by application programs. Application programs should use these library programs, rather than use the *Load And Reserve* and *Store Conditional* instructions directly.

Load Word And Reserve Indexed X-form

lwarx RT,RA,RB

31	RT	RA	RB	20	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← 320 || 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.

This instruction creates a reservation for use by a *Store Word Conditional* instruction. An address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation.

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

Engineering Note

Causing an Alignment exception if attempt is made to execute a *Load And Reserve* or *Store Conditional* instruction having an incorrectly aligned effective address facilitates the debugging of software.

Load Doubleword And Reserve Indexed X-form

ldarx RT,RA,RB

31	RT	RA	RB	84	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
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.

This instruction creates a reservation for use by a *Store Doubleword Conditional* instruction. An address computed from the EA as described in Section 1.7.2.1 is associated with the reservation, and replaces any address previously associated with the reservation.

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.

Special Registers Altered:
None

Store Word Conditional Indexed X-form

stwcx. RS,RA,RB

31	RS	RA	RB	150	1
0	6	11	16	21	31

```
  if RA = 0 then b ← 0
  else           b ← (RA)
  EA ← b +tea (RB)
  if RESERVE then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 4) ← (RS)32:63
†     MEMtag(EA, 4) ← 0
      CR0 ← 0b00 || 0b1 || XERSO
      FXCC ← 0b00 || 0b1 || 0b0
    else
      u ← undefined 1-bit value
      if u then
        MEM(EA, 4) ← (RS)32:63
†       MEMtag(EA, 4) ← 0
      CR0 ← 0b00 || u || XERSO
      FXCC ← 0b00 || u || 0b0
    RESERVE ← 0
  else
    CR0 ← 0b00 || 0b0 || XERSO
    FXCC ← 0b00 || 0b0 || 0b0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

† If a reservation exists and the storage location specified by the ***stwcx.*** is the same as that specified by the *Load And Reserve* instruction that established the reservation, $(RS)_{32:63}$ are stored into the word in storage addressed by EA and the reservation is cleared.

† If a reservation exists but the storage location specified by the ***stwcx.*** is not the same as that specified by the *Load And Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 and the FXCC are set to reflect whether the store operation was performed, as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}
FXCC = 0b00 || store_performed || 0b0

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:
CR0 FXCC

†

Store Doubleword Conditional Indexed X-form

stdcx. RS,RA,RB

31	RS	RA	RB	214	1
0	6	11	16	21	31

```
  if RA = 0 then b ← 0
  else           b ← (RA)
  EA ← b +tea (RB)
  if RESERVE then
    if RESERVE_ADDR = real_addr(EA) then
      MEM(EA, 8) ← (RS)
†     MEMtag(EA, 8) ← 0
      CR0 ← 0b00 || 0b1 || XERSO
      FXCC ← 0b00 || 0b1 || 0b0
    else
      u ← undefined 1-bit value
      if u then
        MEM(EA, 8) ← (RS)
†       MEMtag(EA, 8) ← 0
      CR0 ← 0b00 || u || XERSO
      FXCC ← 0b00 || u || 0b0
    RESERVE ← 0
  else
    CR0 ← 0b00 || 0b0 || XERSO
    FXCC ← 0b00 || 0b0 || 0b0
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

† If a reservation exists and the storage location specified by the ***stdcx.*** is the same as that specified by the *Load And Reserve* instruction that established the reservation, (RS) is stored into the doubleword in storage addressed by EA and the reservation is cleared.

† If a reservation exists but the storage location specified by the ***stdcx.*** is not the same as that specified by the *Load And Reserve* instruction that established the reservation, the reservation is cleared, and it is undefined whether (RS) is stored into the doubleword in storage addressed by EA.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 and the FXCC are set to reflect whether the store operation was performed, as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}
FXCC = 0b00 || store_performed || 0b0

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.

Special Registers Altered:
CR0 FXCC

†

3.3.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses are performed with respect to other processors and mechanisms. Additional information about these instructions and about related aspects of storage management can be found in Book III, *PowerPC AS Operating Environment Architecture*.

Extended mnemonics for Synchronize

Extended mnemonics are provided for the *Synchronize* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand. These are shown as examples with the instruction. See Appendix A, "Assembler Extended Mnemonics" on page 39.

Synchronize X-form

sync L

[POWER mnemonic: dcs]

31	///	L	///	///	598	/
0	6	10	11	16	21	31

The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses that is ordered by the memory barrier depends on the value of the L bit.

L = 0 ("heavyweight sync")
The memory barrier provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the ***sync*** instruction. The applicable pairs are all pairs a_i,b_j in which b_j is a data access.

L = 1 ("lightweight sync")
The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and ***dcbz*** instructions that are executed by the processor executing the ***sync*** instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such accesses except those in which a_i is an access caused by a *Store* or ***dcbz*** instruction and b_j is an access caused by a *Load* instruction.

The ordering done by the memory barrier is cumulative.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed.

If L=0, the ***sync*** instruction has the following additional properties.

- Executing it ensures that all instructions preceding the ***sync*** instruction have completed before the ***sync*** instruction completes, and that no subsequent instructions are initiated until after the ***sync*** instruction completes.
- It is execution synchronizing (see Book III, *PowerPC AS Operating Environment Architecture*).

Special Registers Altered:
None

Extended Mnemonics:

Extended mnemonics for *Synchronize*:

Extended:	*Equivalent to:*
sync	sync 0
lwsync	sync 1

Except in the ***sync*** instruction description in this section, references to "***sync***" in Books I – III imply L=0 unless otherwise stated or obvious from context; "***lwsync***" is used when L=1 is intended.

Programming Note

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

Programming Note

The ***sync*** instruction can be used to ensure that all stores into a data structure, caused by *Store* instructions executed in a "critical section" of a program, will be performed with respect to another processor before the store that releases the lock is performed with respect to that processor; see Section B.2, "Lock Acquisition and Release, and Related Techniques" on page 43.

For instructions following a ***sync*** instruction, the storage accesses listed below need not be ordered after the memory barrier created by the ***sync*** instruction.

- implicit storage accesses (see Book III) for purposes of address translation and reference, change, and tag set recording
- instruction fetches

The memory barrier created by the ***sync*** instruction does not order the actions (if any) taken by the processor in response to the hint provided by a ***dcbt*** or ***dcbtst*** instruction.

Additional operations that are ordered by ***sync*** with L=0 include Reference, Change, and Tag Set bit updates and, with ***tlbsync***, TLB invalidations; see Book III.

If L=0 the functions performed by the ***sync*** instruction may take a significant amount of time to complete, so indiscriminate use of this form of the instruction may adversely affect performance. Using either ***sync*** with L=1 or the ***eieio*** instruction may be more appropriate than using ***sync*** with L=0 for many cases.

Engineering Note

Unlike a context synchronizing operation, ***sync*** need not cause prefetched instructions to be discarded.

Architecture Note

The functions provided by ***sync*** with L=1 are a strict subset of those provided by ***sync*** with L=0.

Enforce In-order Execution of I/O X-form

eieio

31	///	///	///	854	/
0	6	11	16	21	31

The ***eieio*** instruction creates a memory barrier (see Section 1.7.1), which provides an ordering function for the storage accesses caused by *Load*, *Store*, ***dcbz***, ***eciwx***, and ***ecowx*** instructions executed by the processor executing the ***eieio*** instruction. These storage accesses are divided into two sets, which are ordered separately. The storage access caused by an ***eciwx*** instruction is ordered as a load, and the storage access caused by a ***dcbz*** or ***ecowx*** instruction is ordered as a store.

1. Loads and stores to storage that is both Caching Inhibited and Guarded, and stores to main storage caused by stores to storage that is Write Through Required

   The applicable pairs are all pairs a_i,b_j of such accesses.

   The ordering done by the memory barrier for accesses in this set is *not* cumulative.

2. Stores to storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited

   The applicable pairs are all pairs a_i,b_j of such accesses.

   The ordering done by the memory barrier for accesses in this set is cumulative.

The ***eieio*** instruction may complete before storage accesses caused by instructions preceding the ***eieio*** instruction have been performed.

Special Registers Altered:
None

Programming Note

The ***eieio*** instruction is intended for use in managing shared data structures (see Appendix B, “Programming Examples for Sharing Storage” on page 41), in doing memory-mapped I/O, and in preventing load/store combining operations in main storage (see Section 1.6, “Storage Control Attributes” on page 4).

Because stores to storage that is both Caching Inhibited and Guarded are performed in program order (see Section 1.7.1, “Storage Access Ordering” on page 6), ***eieio*** is needed for such storage only when loads must be ordered with respect to stores or with respect to other loads, or when load/store combining operations must be prevented.

For accesses in set 1, a_i and b_j need not be the same kind of access or be to storage having the same storage control attributes. For example, a_i can be a load to Caching Inhibited, Guarded storage, and b_j a store to Write Through Required storage.

If stronger ordering is desired than that provided by ***eieio***, the ***sync*** instruction must be used, with the appropriate value in the L field.

Engineering Note

See the descriptions of ***tlbie*** and ***tlbsync*** in Book III for additional operations that are ordered by ***eieio***.

Architecture Note

The functions provided by ***eieio*** are a strict subset of those provided by ***sync*** with L=0. The functions provided by ***eieio*** for its second set are a strict subset of those provided by ***sync*** with L=1.

Chapter 4. Time Base

The Time Base (TB) is a 64-bit register (see Figure 2) containing a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 63). The frequency at which the integer is updated is implementation-dependent.

TBU (bits 0–31)	TBL (bits 32–63)

Field *Description*
TBU Upper 32 bits of Time Base
TBL Lower 32 bits of Time Base

Figure 2. Time Base

The Time Base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64}-1$). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt: see Book III, *PowerPC AS Operating Environment Architecture*) that this has occurred.

The period of the Time Base depends on the driving frequency. As an order of magnitude example, suppose that the CPU clock is 100 MHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{TB} = \frac{2^{64} \times 32}{100\ \text{MHz}} = 5.90 \times 10^{12}\ \text{seconds}$$

which is approximately 187,000 years.

The PowerPC AS Architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock, in a PowerPC AS system. The Time Base update frequency is not required to be constant. What *is* required, so that system software can keep time of day and operate interval timers, is one of the following.

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the Time Base changes, and a means to determine what the current update frequency is.
- The update frequency of the Time Base is under the control of the system software.

Engineering Note

See Book III, *PowerPC AS Operating Environment Architecture* for additional requirements related to secure systems.

Programming Note

If the operating system initializes the Time Base on power-on to some reasonable value and the update frequency of the Time Base is constant, the Time Base can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the Time Base are monotonically increasing (except when the Time Base wraps from $2^{64}-1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of Time Base values can be post-processed to become actual time values.

Successive readings of the Time Base may return identical values.

4.1 Time Base Instructions

Extended mnemonics

† Extended mnemonics are provided provided for the † *Move From Time Base* instruction so that it can be coded with the TBR name as part of the mnemonic rather than as a numeric operand. See the appendix entitled “Assembler Extended Mnemonics” in Book III, *PowerPC AS Operating Environment Architecture*.

Move From Time Base XFX-form

mftb RT,TBR

31	RT	tbr	371	/
0	6	11	21	31

```
n ← tbr5:9 || tbr0:4
if n = 268 then
    RT ← TB
else if n = 269 then
    RT ← 320 || TB0:31
```

The TBR field denotes either the Time Base or Time † Base Upper, encoded as shown in the table below. The contents of the designated register are placed into register RT. When reading Time Base Upper, the high-order 32 bits of register RT are set to zero.

	TBR* decimal	TBR* $tbr_{5:9}$ $tbr_{0:4}$	Register Name
†	268	01000 01100	TB
†	269	01000 01101	TBU

* Note that the order of the two 5-bit halves of the TBR number is reversed.

If the TBR 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.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Move From Time Base*:

Extended:	*Equivalent to:*
mftb Rx	mftb Rx,268
mftbu Rx	mftb Rx,269

Programming Note

mftb serves as both a basic and an extended mnemonic. The Assembler will recognize an ***mftb*** mnemonic with two operands as the basic form, and an ***mftb*** mnemonic with one operand as the † extended form. In the extended form the TBR † operand is omitted and assumed to be 268 (the † value that corresponds to TB).

Compiler and Assembler Note

The TBR 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.

Architecture Note

Some implementations may implement ***mftb*** and ***mfspr*** identically. Therefore a TBR number must not match an SPR number.

Engineering Note

The extended opcode for ***mftb*** differs from that of ***mfspr*** by only one bit. Implementations are permitted to ignore this bit and treat both instructions identically.

4.2 Reading the Time Base

The contents of the Time Base can be read into a GPR by the ***mftb*** extended mnemonic. To read the contents of the Time Base into register Rx, execute:

```
mftb Rx
```

Reading the Time Base has no effect on the value it contains or on the periodic incrementing of that value.

4.3 Computing Time of Day from the Time Base

Since the update frequency of the Time Base is implementation-dependent, the algorithm for converting the current value in the Time Base to time of day is also implementation-dependent.

As an example, assume that the Time Base is incremented at a constant rate of once for every 32 cycles of a 100 MHz CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock:[1] the number of whole seconds that have passed since midnight January 0, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume that:

- The value 0 in the Time Base represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction will make it so).
- The integer constant *ticks_per_sec* contains the value

  $$\frac{100\text{ MHz}}{32} = 3{,}125{,}000$$

  which is the number of times the Time Base is updated each second.
- The integer constant *ns_adj* contains the value

  $$\frac{1{,}000{,}000{,}000}{3{,}125{,}000} = 320$$

  which is the number of nanoseconds per tick of the Time Base.

The POSIX clock can be computed with an instruction sequence such as this:

```
mftb   Ry            # Ry = Time Base
lwz    Rx,ticks_per_sec
divd   Rz,Ry,Rx      # Rz = whole seconds
stw    Rz,posix_sec
mulld  Rz,Rz,Rx      # Rz = quotient * divisor
sub    Rz,Ry,Rz      # Rz = excess ticks
lwz    Rx,ns_adj
mulld  Rz,Rz,Rx      # Rz = excess nanoseconds
stw    Rz,posix_ns
```

Non-constant update frequency

In a system in which the update frequency of the Time Base may change over time, it is not possible to convert an isolated Time Base value into time of day. Instead, a Time Base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change via an interrupt (see Book III, *PowerPC AS Operating Environment Architecture*), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, Time Base value, and tick rate. Subsequent calls to compute time of day use the current Time Base value and the saved data.

1 Described in POSIX Draft Standard P1003.4/D12, *Draft Standard for Information Technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension* [*C Language*]. Institute of Electrical and Electronics Engineers, Inc., Feb. 1992.

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 below.

5.1 External Control

The External Control facility permits a program to communicate with a special-purpose device. Two instructions are provided, both of which must be implemented if the facility is provided.

- *External Control In Word Indexed* (***eciwx***), which does the following:
  - Computes an effective address (EA) as for any X-form instruction
  - Validates the EA as would be done for a load from that address
  - Translates the EA to a real address
  - Transmits the real address to the device
  - Accepts a word of data from the device and places it into a General Purpose Register
- *External Control Out Word Indexed* (***ecowx***), which does the following:
  - Computes an effective address (EA) as for any X-form instruction
  - Validates the EA as would be done for a store to that address
  - Translates the EA to a real address
  - Transmits the real address and a word of data from a General Purpose Register to the device

Permission to execute these instructions and identification of the target device are controlled by two fields, called the E bit and the RID field respectively. If attempt is made to execute either of these instructions when E=0 the system data storage error handler is invoked. The location of these fields is described in Book III, *PowerPC AS Operating Environment Architecture.*

The storage access caused by ***eciwx*** and ***ecowx*** is performed as though the specified storage location is Caching Inhibited and Guarded, and is neither Write Through Required nor Memory Coherence Required.

Interpretation of the real address transmitted by ***eciwx*** and ***ecowx*** and of the 32-bit value transmitted by ***ecowx*** is up to the target device, and is not specified by the PowerPC AS Architecture. See the System Architecture documentation for a given PowerPC AS system for details on how the External Control facility can be used with devices on that system.

Example

An example of a device designed to be used with the External Control facility might be a graphics adapter. The ***ecowx*** instruction might be used to send the device the translated real address of a buffer containing graphics data, and the word transmitted from the General Purpose Register might be control information that tells the adapter what operation to perform on the data in the buffer. The ***eciwx*** instruction might be used to load status information from the adapter.

A device designed to be used with the External Control facility may also recognize events that indicate that the address translation being used by the processor has changed. In this case the operating system need not "pin" the area of storage identified by an ***eciwx*** or ***ecowx*** instruction (i.e., need not protect it from being paged out).

5.1.1 External Access Instructions

In the instruction descriptions the statements "this instruction is treated as a *Load*" and "this instruction is treated as a *Store*" have the same meanings as for the *Cache Management* instructions; see Section 3.2, "Cache Management Instructions" on page 16.

External Control In Word Indexed X-form

eciwx RT,RA,RB

31	RT	RA	RB	310	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
raddr ← address translation of EA
send load word request for raddr to
  device identified by RID
RT ← 320 || word from device
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

A load word request for the real address corresponding to EA is sent to the device identified by RID, bypassing the cache. The word returned by the device is placed into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

The E bit must be 1. If it is not, the data storage error handler is invoked.

† 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.

This instruction is treated as a *Load*.

†

See Book III, *PowerPC AS Operating Environment Architecture* for additional information about this instruction.

Special Registers Altered:
None

Programming Note

The ***eieio*** instruction can be used to ensure that the storage accesses caused by ***eciwx*** and ***ecowx*** are performed in program order with respect to other Caching Inhibited and Guarded storage accesses.

Engineering Note

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

External Control Out Word Indexed X-form

ecowx RS,RA,RB

31	RS	RA	RB	438	/
0	6	11	16	21	31

```
if RA = 0 then b ← 0
else           b ← (RA)
EA ← b +tea (RB)
raddr ← address translation of EA
send store word request for raddr to
  device identified by RID
send (RS)32:63 to device
```

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

A store word request for the real address corresponding to EA and the contents of $RS_{32:63}$ are sent to the device identified by RID, bypassing the cache.

The E bit must be 1. If it is not, the data storage error handler is invoked.

† 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.

This instruction is treated as a *Store*, except that its storage access is not performed in program order with respect to accesses to other Caching Inhibited and Guarded storage locations unless software explicitly imposes that order.

†

See Book III, *PowerPC AS Operating Environment Architecture* for additional information about this instruction.

Special Registers Altered:
None

Architecture Note

Treating ***ecowx*** as a *Store* with respect to the storage access ordering done solely by virtue of the Caching Inhibited and Guarded storage control attributes would require the processor to detect this case during instruction decoding, instead of during address translation as for other Caching Inhibited and Guarded stores.

5.2 Storage Control Instructions

5.2.1 Cache Management Instructions

† 5.2.1.1 Data Cache Instruction

The optional version of the *Data Cache Block Touch* instruction includes a TH (Touch Hint) field, which permits a program to provide a hint that a sequence of data cache blocks is likely to be needed soon. The sequence is called a “data stream”.

Data Cache Block Touch X-form

dcbt RA,RB,TH

31	///	TH	RA	RB	278	/
0	6	9	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+ $_{tea}$(RB).

The ***dcbt*** instruction provides a hint that the program will probably soon load from the storage locations specified by EA and the TH field. The hint is ignored for storage locations that are Caching Inhibited or Guarded.

The encodings of the TH field are as follows.

TH **Description**

00 The storage location is the block containing the byte addressed by EA.

01 The storage locations are the block containing the byte addressed by EA and sequentially following blocks (i.e., the blocks containing the bytes addressed by EA + n × block_size, where n = 0, 1, 2, ...).

10 Reserved

11 The storage locations are the block containing the byte addressed by EA and sequentially preceding blocks (i.e., the blocks containing the bytes addressed by EA – n × block_size, where n = 0, 1, 2, ...).

The actions (if any) taken by the processor in response to the hint are not considered to be “caused by” or “associated with” the ***dcbt*** instruction (e.g., ***dcbt*** is considered not to cause any data accesses). No means are provided by which software can synchronize these actions with the execution of the instruction stream. For example, these actions are not ordered by the memory barrier created by a ***sync*** instruction.

This instruction is treated as a *Load* (see Section 3.2), except that the system data storage error handler is not invoked, and reference and change recording need not be done.

Special Registers Altered:
None

Programming Note

In response to the hint provided by ***dcbt***, the processor may prefetch the specified storage locations into the data cache, or take other actions that reduce the latency of subsequent *Load* instructions that refer to the locations.

Programming Note

dcbt serves as both a basic and an extended mnemonic. The Assembler will recognize a ***dcbt*** mnemonic with three operands as the basic form, and a ***dcbt*** mnemonic with two operands as the extended form. In the extended form the TH operand is omitted and assumed to be 0b00.

Programming Note

If the TH field is set to 0b00, the instruction operates as described in Section 3.2.2, “Data Cache Instructions” on page 18.

The TH field should not be set to 0b10, as that value may be assigned a meaning in some future version of the architecture.

Earlier implementations that do not support the optional version of ***dcbt*** ignore the TH field (i.e., treat it as if it were set to 0b00), and do not necessarily ignore the hint provided by ***dcbt*** if the specified block is in storage that is Guarded and not Caching Inhibited. Therefore a ***dcbt*** instruction with TH_1= 1 should not specify an EA in such storage if the program is to be run on such implementations.

Architecture Note

Some implementations use bit 8 of the ***dcbt*** instruction as an additional prefetch hint. This bit will not be assigned a meaning in the PowerPC AS Architecture except after careful consideration of the effect of such assignment on existing implementations.

Programming Note

Although optimal use of the data stream variant of ***dcbt*** ($TH_1=1$) depends on the characteristics of the prefetch mechanism and of the storage hierarchy (see Book IV), the programmer should assume that the following programming model is supported.

- Data stream resources are allocated in round-robin fashion. Therefore ***dcbt*** instructions (with $TH_1=1$) should be executed for the least important stream first and the most important stream last. If this technique is used and ***dcbt*** instructions are executed for more streams than the processor supports, the most important streams will be prefetched.
- The prefetch mechanism paces prefetching of a data stream with consumption of the prefetched data, prefetching only a limited number of blocks ahead of the block that is currently being loaded from by the program. As a consequence, when the program ceases to load from successive blocks of the stream, prefetching of the stream ceases.
- Certain conditions may cause prefetching to be terminated for a data stream that the program is still using. However, the prefetch mechanism will subsequently detect that the stream is still being loaded from and will resume prefetching of the stream. Therefore there is no need to code more than one ***dcbt*** instruction (with $TH_1=1$) for the stream.

Although the ***dcbt*** instruction described in Section 3.2.2 (equivalently, ***dcbt*** with TH=0b00) can be used to provide the same function as the data stream variant, the data stream variant may be easier to use because only one instance of the ***dcbt*** instruction is needed per stream, instead of one per cache block, and because the performance of processing the stream is less sensitive to how far ahead of the *Load* instructions the ***dcbt*** instruction is placed.

Engineering Note

Programs that use the data stream variant of ***dcbt*** are likely to contain multiple instances of the instruction (each for a different data stream) preceding the *Load* instructions that refer to the prefetched blocks. In designing the data cache and any associated prefetch buffers, consideration should be given to minimizing the extent to which the prefetched blocks displace other data needed or requested by the program, or are themselves displaced before they are used.

5.3 Little-Endian

If the optional Little-Endian facility is implemented (see the section entitled “Little-Endian” in Book I, *PowerPC AS User Instruction Set Architecture*), the programmer should assume the performance model described in Figure 3 with respect to the placement of storage operands that are accessed in Little-Endian mode.

Operand		Boundary Crossing			
Size	**Byte Align.**	**None**	**Cache Block**	**Virtual Page**[2]	**Seg.**
Integer					
8 Byte	8 4 < 4	optimal good poor	– good poor	– poor poor	– poor poor
4 Byte	4 < 4	optimal good	– good	– poor	– poor
2 Byte	2 < 2	optimal good	– good	– poor	– poor
1 Byte	1	optimal	–	–	–
Float					
8 Byte	8 4 < 4	optimal good poor	– good poor	– poor poor	– poor poor
4 Byte	4 < 4	optimal poor	– poor	– poor	– poor

[1] If an instruction causes an access that is not atomic and any portion of the operand is in storage that is Write Through Required or Caching Inhibited, performance is likely to be poor.

[2] If the storage operand spans two virtual pages that have different storage control attributes, performance is likely to be poor.

Figure 3. Performance effects of storage operand placement, Little-Endian mode

Appendix A. Assembler Extended Mnemonics

In order to make assembler language programs simpler to write and easier to understand, a set of extended mnemonics and symbols is provided for certain instructions. This appendix defines extended mnemonics and symbols related to instructions defined in Book II.

Assemblers should provide the extended mnemonics and symbols listed here, and may provide others.

A.1 Synchronize Mnemonics

The L field in the *Synchronize* instruction controls whether the instruction performs a “heavyweight” synchronization function or a “lightweight” synchronization function. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

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

sync	(equivalent to:	sync 0)
lwsync	(equivalent to:	sync 1)

† Appendix B. Programming Examples for Sharing Storage

This appendix gives examples of how dependencies
† and the *Synchronization* instructions can be used to control storage access ordering when storage is
† shared between programs.

† Many of the examples use extended mnemonics (e.g.,
† ***bne, bne-, cmpw***) that are defined in the Appendix
† entitled "Assembler Extended Mnemonics" in Book I,
† *PowerPC AS User Instruction Set Architecture*.

† Many of the examples use the *Load And Reserve* and
† *Store Conditional* instructions, in a sequence that
† begins with a *Load And Reserve* instruction and ends
† with a *Store Conditional* instruction (specifying the
† same storage location as the *Load Conditional*) fol-
† lowed by a *Branch Conditional* instruction that tests
† whether the *Store Conditional* instruction succeeded.

† In these examples it is assumed that contention for
† the shared resource is low; the conditional branches
† are optimized for this case by using "+" and "–" suf-
† fixes appropriately.

The examples deal with words; they can be used for
† doublewords by changing all word-specific mnemonics
† to the corresponding doubleword-specific mnemonics
† (e.g., ***lwarx*** to ***ldarx***, ***cmpw*** to ***cmpd***).

| In this appendix it is assumed that all shared storage
| locations are in storage that is Memory Coherence
| Required, and that the storage locations specified by
| *Load And Reserve* and *Store Conditional* instructions
| are in storage that is neither Write Through Required
| nor Caching Inhibited.

† B.1 Atomic Update Primitives

† This section gives examples of how the *Load And*
† *Reserve* and *Store Conditional* instructions can be
† used to emulate atomic read/modify/write operations.

† An atomic read/modify/write operation reads a
† storage location and writes its next value, which may
† be a function of its current value, all as a single
† atomic operation. The examples shown provide the
† effect of an atomic read/modify/write operation, but
† use several instructions rather than a single atomic
† instruction.

Fetch and No-op

The "Fetch and No-op" primitive atomically loads the current value in a word in storage.

In this example it is assumed that the address of the word to be loaded is in GPR 3 and the data loaded are returned in GPR 4.

```
loop: lwarx  r4,0,r3    #load and reserve
      stwcx. r4,0,r3    #store old value if
                        #  still reserved
      bne–   loop       #loop if lost reservation
```

Note:

1. The ***stwcx.***, if it succeeds, stores to the target location the same value that was loaded by the preceding ***lwarx***. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the
† ***lwarx*** is still the current value at the time the
† ***stwcx.*** is executed.

Fetch and Store

The "Fetch and Store" primitive atomically loads and replaces a word in storage.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR 3, the new value is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3    #load and reserve
      stwcx. r4,0,r3    #store new value if
                        #  still reserved
      bne–   loop       #loop if lost reservation
```

Fetch and Add

The “Fetch and Add” primitive atomically increments a word in storage.

In this example it is assumed that the address of the word to be incremented is in GPR 3, the increment is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3    #load and reserve
      add    r0,r4,r5   #increment word
      stwcx. r0,0,r3    #store new value if
                        #  still reserved
      bne–   loop       #loop if lost reservation
```

Fetch and AND

The “Fetch and AND” primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR 3, the value to AND into it is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3    #load and reserve
      and    r0,r4,r5   #AND word
      stwcx. r0,0,r3    #store new value if
                        #  still reserved
      bne–   loop       #loop if lost reservation
```

Note:

1. The sequence given above can be changed to perform another Boolean operation atomically on a word in storage, simply by changing the ***and*** instruction to the desired Boolean instruction (***or, xor,*** etc.).

Test and Set

This version of the “Test and Set” primitive atomically loads a word from storage, sets the word in storage to a nonzero value if the value loaded is zero,
† and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR 3, the new value (nonzero) is in GPR 4, and the old value is returned in GPR 5.

```
loop: lwarx  r5,0,r3    #load and reserve
      cmpwi  r5,0       #done if word
      bne–   $+12       #  not equal to 0
      stwcx. r4,0,r3    #try to store non-0
      bne–   loop       #loop if lost reservation
```

Compare and Swap

The “Compare and Swap” primitive atomically compares a value in a register with a word in storage, if they are equal stores the value from a second register into the word in storage, if they are unequal loads the word from storage into the first register,
† and sets the EQ bit of CR Field 0 to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR 3, the comparand is in GPR 4 and the old value is returned there, and the new value is in GPR 5.

```
loop: lwarx  r6,0,r3    #load and reserve
      cmpw   r4,r6      #1st 2 operands equal?
      bne–   exit       #skip if not
      stwcx. r5,0,r3    #store new value if
                        #  still reserved
      bne–   loop       #loop if lost reservation
exit: mr     r4,r6      #return value from storage
```

Notes:

1. The semantics given for “Compare and Swap” above are based on those of the IBM System/370 Compare and Swap instruction. Other architectures may define a Compare and Swap instruction differently.
2. “Compare and Swap” is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by ***lwarx*** and ***stwcx.***. A major weakness of a System/370-style Compare and Swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a Compare and Swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second ***bne-*** instruction and/or the ***mr*** instruction can be omitted. The ***bne-*** is needed only if the application requires that if the EQ bit of CR Field 0 on exit indicates “not equal” then (r4) and (r6) are in fact not equal. The ***mr*** is needed only if the application requires that if the comparands are not equal then the word from storage is loaded into the register with which it was compared (rather than into a third register). If either or both of these instructions is omitted, the resulting Compare and Swap does not obey System/370 semantics.

† B.2 Lock Acquisition and Release, and Related Techniques

† This section gives examples of how dependencies and † the *Synchronization* instructions can be used to imple- † ment locks, import and export barriers, and similar † constructs.

B.2.1 Lock Acquisition and Import Barriers

An "import barrier" is an instruction or sequence of instructions that prevents storage accesses caused by instructions following the barrier from being performed before storage accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. † A ***sync*** instruction can be used as an import barrier, but the approaches shown below will generally yield better performance because they order only the relevant storage accesses.

B.2.1.1 Acquire Lock and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain the lock, an import barrier can be constructed by placing an ***isync*** instruction immediately following the loop containing the ***lwarx*** and ***stwcx.***. The following † example uses the "Compare and Swap" primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop: lwarx  r6,0,r3    #load lock and reserve
      cmpw   r4,r6      #skip ahead if
      bne-   wait       #  lock not free
      stwcx. r5,0,r3    #try to set lock
      bne-   loop       #loop if lost reservation
      isync             #import barrier
      lwz    r7,data1(r9) #load shared data
      .
      .
wait: ...               #wait for lock to free
```

The second ***bne-*** does not complete until CR0 has been set by the ***stwcx.***. The ***stwcx.*** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the ***stwcx.*** completes successfully. Together, the second ***bne-*** and the subsequent ***isync*** create an import barrier that prevents the load from "data1" from being performed until the branch has been resolved not to be taken.

| If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***lwsync*** instruction can be used instead of the ***isync*** instruction. If ***lwsync*** is used, the load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the second branch is taken and the ***lwarx*** is reexecuted. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even if the load is performed before the ***stwcx.***, because the ***lwsync*** ensures that the load is performed after the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

B.2.1.2 Obtain Pointer and Import Shared Storage

If ***lwarx*** and ***stwcx.*** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the "Fetch and † Add" primitive to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop: lwarx  r5,0,r3    #load pointer and reserve
      add    r0,r4,r5   #increment the pointer
      stwcx. r0,0,r3    #try to store new value
      bne-   loop       #loop if lost reservation
      lwz    r7,data1(r5) #load shared data
```

The load from "data1" cannot be performed until the pointer value has been loaded into GPR 5 by the † ***lwarx***. The load from "data1" may be performed before the ***stwcx.***. But if the ***stwcx.*** fails, the branch is taken and the value returned by the load from "data1" is discarded. If the ***stwcx.*** succeeds, the value returned by the load from "data1" is valid even † if the load is performed before the ***stwcx.***, because the load uses the pointer value returned by the instance of the ***lwarx*** that created the reservation used by the successful ***stwcx.***.

An ***isync*** instruction could be placed between the ***bne-*** and the subsequent ***lwz***, but no ***isync*** is needed if all accesses to the shared data structure depend on the value returned by the ***lwarx***.

B.2.2 Lock Release and Export Barriers

An "export barrier" is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock will be performed with respect to any other processor before the store that releases the lock is performed with respect to that processor.

B.2.2.1 Export Shared Storage and Release Lock

A ***sync*** instruction can be used as an export barrier independent of the storage control attributes (e.g., presence or absence of the Caching Inhibited attribute) of the storage containing the shared data structure. Because the lock must be in storage that is neither Write Through Required nor Caching Inhibited, if the shared data structure is in storage that is Write Through Required or Caching Inhibited a ***sync*** instruction *must* be used as the export barrier.

In this example it is assumed that the shared data structure is in storage that is Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw   r7,data1(r9) #store shared data (last)
sync               #export barrier
stw   r4,lock(r3)  #release lock
```

The ***sync*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***sync*** have been performed with respect to that processor.

B.2.2.2 Export Shared Storage and Release Lock using eieio or lwsync

If the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, an ***eieio*** instruction can be used as the export barrier. Using ***eieio*** rather than ***sync*** will yield better performance in most systems.

In this example it is assumed that the shared data structure is in storage that is neither Write Through Required nor Caching Inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw   r7,data1(r9) #store shared data (last)
eieio              #export barrier
stw   r4,lock(r3)  #release lock
```

The ***eieio*** ensures that the store that releases the lock will not be performed with respect to any other processor until all stores caused by instructions preceding the ***eieio*** have been performed with respect to that processor.

However, for storage that is neither Write Through Required nor Caching Inhibited, ***eieio*** orders only stores and has no effect on loads. If the portion of the program preceding the ***eieio*** contains loads from the shared data structure and the stores to the shared data structure do not depend on the values returned by those loads, the store that releases the lock could be performed before those loads. If it is necessary to ensure that those loads are performed before the store that releases the lock, ***lwsync*** should be used instead of ***eieio***. Alternatively, the technique described in Section B.2.3 can be used.

B.2.3 Safe Fetch

If a load must be performed before a subsequent store (e.g., the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the storage operand to be loaded is in GPR 3, the contents of the storage operand are returned in GPR 4, and the address of the storage operand to be stored is in GPR 5.

```
lwz    r4,0(r3)   #load shared data
cmpw   r4,r4      #set CR0 to "equal"
bne-   $-8        #branch never taken
stw    r7,0(r5)   #store other shared data
```

An alternative is to use a technique similar to that described in Section B.2.1.2, by causing the ***stw*** to depend on the value returned by the ***lwz*** and omitting the ***cmpw*** and ***bne-***. The dependency could be created by ANDing the value returned by the ***lwz*** with zero and then adding the result to the value to be stored by the ***stw***. If both storage operands are in storage that is neither Write Through Required nor Caching Inhibited, another alternative is to replace the ***cmpw*** and ***bne-*** with an ***lwsync*** instruction.

B.3 List Insertion

This section shows how the ***lwarx*** and ***stwcx.*** instructions can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The "next element pointer" from the list element after which the new element is to be inserted, here called the "parent element", is stored into the new element, so that the new element points to the next element in the list; this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR 3, the address of the new element is in GPR 4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop: lwarx  r2,0,r3    #get next pointer
      stw    r2,0(r4)   #store in new element
      eieio             #order stw before stwcx.
      stwcx. r4,0,r3    #add new element to list
      bne–   loop       #loop if stwcx. failed
```

In the preceding example, ***lwsync*** can be used instead of ***eieio***.

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, "livelock" can occur. (Livelock is a state in which processors interact in a way such that no processor makes forward progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, sequence.

```
       lwz    r2,0(r3)   #get next pointer
loop1: mr     r5,r2      #keep a copy
       stw    r2,0(r4)   #store in new element
       sync              #order stw before stwcx.
                         #  and before lwarx
loop2: lwarx  r2,0,r3    #get it again
       cmpw   r2,r5      #loop if changed (someone
       bne–   loop1      #  else progressed)
       stwcx. r4,0,r3    #add new element to list
       bne–   loop2      #loop if failed
```

In the preceding example, livelock is avoided by the fact that each processor reexecutes the ***stw*** only if some other processor has made forward progress.

B.4 Notes

1. To increase the likelihood that forward progress is made, it is important that looping on ***lwarx*/*stwcx.*** pairs be minimized. For example, in the "Test and Set" sequence shown in Section B.1, this is achieved by testing the old value before attempting the store; were the order reversed, more ***stwcx.*** instructions might be executed, and reservations might more often be lost between the ***lwarx*** and the ***stwcx.***.

2. The manner in which ***lwarx*** and ***stwcx.*** are communicated to other processors and mechanisms, and between levels of the storage hierarchy within a given processor, is implementation-dependent. In some implementations performance may be improved by minimizing looping on a ***lwarx*** instruction that fails to return a desired value. For example, in the "Test and Set" sequence shown in Section B.1, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the "bne- $+12" to "bne- loop". However, in some implementations better performance may be obtained by using an ordinary *Load* instruction to do the initial checking of the value, as follows.

```
loop: lwz    r5,0(r3)   #load the word
      cmpwi  r5,0       #loop back if word
      bne–   loop       #  not equal to 0
      lwarx  r5,0,r3    #try again, reserving
      cmpwi  r5,0       #  (likely to succeed)
      bne–   loop
      stwcx. r4,0,r3    #try to store non-0
      bne–   loop       #loop if lost reserv'n
```

3. In a multiprocessor, livelock is possible if there is a *Store* instruction (or any other instruction that can clear another processor's reservation; see Section 1.7.2.1) between the ***lwarx*** and the ***stwcx.*** of a ***lwarx*/*stwcx.*** loop and any byte of the storage location specified by the *Store* is in the reservation granule. For example, the first code sequence shown in Section B.3 can cause livelock if two list elements have next element pointers in the same reservation granule.

Appendix C. Cross-Reference for Changed POWER Mnemonics

The following table lists the POWER instruction mnemonics that have been changed in the PowerPC AS Virtual Environment Architecture, sorted by POWER mnemonic.

To determine the PowerPC AS mnemonic for one of these POWER mnemonics, find the POWER mnemonic in the second column of the table: the remainder of the line gives the PowerPC AS mnemonic and the page on which the instruction is described, as well as the instruction names.

POWER mnemonics that have not changed are not listed.

	Page	POWER		PowerPC AS	
		Mnemonic	**Instruction**	**Mnemonic**	**Instruction**
	19	dclz	Data Cache Line Set to Zero	dcbz	Data Cache Block set to Zero
†	25	dcs	Data Cache Synchronize	sync	Synchronize
	21	ics	Instruction Cache Synchronize	isync	Instruction Synchronize

Appendix D. New Instructions

The following instructions in the PowerPC AS Virtual Environment Architecture are new: they are not in the POWER Architecture. The ***eciwx*** and ***ecowx*** instructions are optional.

	dcbf	Data Cache Block Flush
	dcbst	Data Cache Block Store
	dcbt	Data Cache Block Touch
	dcbtst	Data Cache Block Touch for Store
	eciwx	External Control In Word Indexed
	ecowx	External Control Out Word Indexed
	eieio	Enforce In-order Execution of I/O
	icbi	Instruction Cache Block Invalidate
†	***ldarx***	Load Doubleword And Reserve Indexed
†	***lwarx***	Load Word And Reserve Indexed
	mftb	Move From Time Base
†	***stdcx.***	Store Doubleword Conditional Indexed
†	***stwcx.***	Store Word Conditional Indexed

Appendix E. PowerPC AS Virtual Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Page	Mnemonic	Instruction
	Primary	Extend				
X	31	86		20	dcbf	Data Cache Block Flush
X	31	54		20	dcbst	Data Cache Block Store
X	31	278		18	dcbt	Data Cache Block Touch
X	31	246		18	dcbtst	Data Cache Block Touch for Store
X	31	1014		19	dcbz	Data Cache Block set to Zero
X	31	310		34	eciwx	External Control In Word Indexed
X	31	438		34	ecowx	External Control Out Word Indexed
X	31	854		27	eieio	Enforce In-order Execution of I/O
X	31	982		17	icbi	Instruction Cache Block Invalidate
XL	19	150		21	isync	Instruction Synchronize
† X	31	84		23	ldarx	Load Doubleword And Reserve Indexed
† X	31	20		23	lwarx	Load Word And Reserve Indexed
XFX	31	371		30	mftb	Move From Time Base
† X	31	214		24	stdcx.	Store Doubleword Conditional Indexed
† X	31	150		24	stwcx.	Store Word Conditional Indexed
† X	31	598		25	sync	Synchronize

[1]Key to Mode Dependency Column

† Except as described in the section entitled “Effective Address Calculation” in Book I, all instructions in the PowerPC AS Virtual Environment Architecture are independent of whether the processor is in 32-bit or 64-bit mode and of whether the processor is in *tags active* or *tags inactive* mode.

Index

Last Page - End of Document