PowerPC Virtual Environment Architecture

Book II

Version 2.02

January 28, 2005

Manager:

Joe Wetzel/Poughkeepsie/IBM

Technical Content:

Ed Silha/Austin/IBM Cathy May/Watson/IBM Brad Frey/Austin/IBM

Junichi Furukawa/Austin/IBM Giles Frazier/Austin/IBM

The following paragraph does not apply to the United Kingdom or any country or state where such provisions are inconsistent with local law.

The specifications in this manual are subject to change without notice. This manual is provided “AS IS”. International Business Machines Corp. makes no warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

International Business Machines Corp. does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

Address comments to IBM Corporation, Internal Zip 9630, 11400 Burnett Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

This document defines the additional instructions and facilities, beyond those of the PowerPC User Instruction Set Architecture, that are provided by the PowerPC 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 User Instruction Set Architecture, the PowerPC Operating Environment Architecture, and PowerPC Implementation Features. Book I, *PowerPC User Instruction Set Architecture* defines the base instruction set and related facilities available to the application programmer. Book III, *PowerPC Operating Environment Architecture* defines the system (privileged) instructions and related facilities. Book IV, *PowerPC Implementation Features* defines the implementation-dependent aspects of a particular implementation.

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

Note: Change bars indicate changes from Version 2.01.

Table of Contents

Figures

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. In these definitions, “*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".

- **processor**
  A hardware component that executes the 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 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 *Load* or *Store* instruction.

- **program order**
  The execution of instructions in the order required by the sequential execution model. (See the section entitled "Instruction Execution Order" in Book I. A ***dcbz*** instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a *Store* instruction as described there.)

- **storage location**
  A contiguous sequence of bytes in storage. When used in association with a specific instruction or the instruction fetching mechanism, the length of the sequence of bytes is typically implied by the operation. In other uses, it may refer more abstractly to a group of bytes which share common storage attributes.

- **storage access**
  An access to a storage location. There are three (mutually exclusive) kinds of storage access.

  - **data access**
    An access to the storage location specified by a *Load* or *Store* instruction, or, if the access is performed "out-of-order" (see Book III), an access to a storage location as if it were the storage location specified by a *Load* or *Store* instruction.

  - **instruction fetch**
    An access for the purpose of fetching an instruction.

  - **implicit access**
    An access by the processor for the purpose of address translation or reference and change recording (see Book III).

- **caused by, associated with**
  - **caused by**
    A storage access is said to be caused by an instruction if the instruction is a *Load* or *Store* and the access (data access) is to the storage location specified by the instruction.
  - **associated with**
    A storage access is said to be associated with an instruction if the access is for the purpose of fetching the instruction (instruction fetch), or is a data access caused by the instruction, or is an implicit access that occurs as a side effect of fetching or executing the instruction.

- **prefetched instructions**
  Instructions for which a copy of the instruction has been fetched from instruction storage, but the instruction has not yet been executed.

- **uniprocessor**
  A system that contains one processor.

- **multiprocessor**
  A system that contains two or more processors.

- **shared storage multiprocessor**
  A multiprocessor that contains some common storage, which all the 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 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 Virtual Environment Architecture, described herein, expands this simple storage model to include caches, virtual storage, and shared storage multiprocessors. The PowerPC Virtual Environment Architecture, in conjunction with services based on the PowerPC 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 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 Architecture that enable programmers to write correct programs for this storage model.

1.3 Virtual Storage

The PowerPC system implements a virtual storage model for applications. 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 system , 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 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

An access that is not atomic is performed as a set of smaller disjoint atomic accesses.

- any *Load* or *Store* instruction for which the operand is unaligned
- ***lmw***, ***stmw***, ***lswi***, ***lswx***, ***stswi***, ***stswx***
- 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.

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.

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 Architecture, e.g., in the descriptions of the *Cache Management* instructions in Section 3.2, “Cache Management Instructions” on page 18. 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 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***[***l***])

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 ***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 Coherency 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 Implementation Features* document for the implementation), but is larger than a word and is usually 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.

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.

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 *Synchronize* 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 associated 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 Virtual 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 45.

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.

Example1:

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.

1.7.2 Storage Ordering of I/O Accesses

A “coherence domain” consists of all processors and all interfaces to main storage. Memory reads and writes initiated by mechanisms outside the coherence domain are performed within the coherence domain in the order in which they enter the coherence domain and are performed as coherent accesses.

1.7.3 Atomic Update

The *Load And Reserve* and *Store Condition*al 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 performed.

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 specified 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 45.

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 by another processor may return a “stale” value. However, a subsequent ***lwarx*** from the given location by the other processor followed by a successful ***stwcx.*** by 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.

1.7.3.1 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 l***darx*** instruction, and is lost (or may be lost, in the case of the third, fourth, and sixth 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.
- The processor holding the reservation executes a ***dcbf***[***l***] to the reservation granule: whether the reservation is lost is undefined.
- The processor holding the reservation modifies a Reference or Change bit (see Book III, *PowerPC Operating Environment Architecture*) in the same reservation granule: whether the reservation is lost is undefined.
- Some other processor executes a *Store* or ***dcbz*** to the same reservation granule, or modifies a Reference or Change bit in the same reservation granule.
- Some other processor executes a ***dcbtst***, ***dcbst***, ***dcbf*** (but not ***dcbfl***) 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 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 45. 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

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 49).

1.7.3.2 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.

1.8 Instruction Storage

The instruction execution properties and requirements described in this section, including its subsections, apply only to instruction execution that is required by the sequential execution model.

In this section, including its subsections, it is assumed that all instructions for which execution is attempted are in storage that is not Caching Inhibited and (unless instruction address translation is disabled; see Book III) is not Guarded, and from which instruction fetching does not cause the system error handler to be invoked (e.g., from which instruction fetching is not prohibited by the “address translation mechanism” or the “storage protection mechanism”; see Book III).

Programming Note

The results of attempting to execute instructions from storage that does not satisfy this assumption are described in Sections 1.6.2 and 1.6.4 of this Book and in Book III.

For each instance of executing an instruction from location X, the instruction may be fetched multiple times.

The instruction cache is not necessarily kept consistent with the data cache or with main storage. It is the responsibility of software to ensure that instruction storage is consistent with data storage when such consistency is required for program correctness.

After one or more bytes of a storage location have been modified and before an instruction located in that storage location is executed, software must execute the appropriate sequence of instructions to make instruction storage consistent with data storage. Otherwise the results of attempting to execute the instruction are boundedly undefined except as described in Section 1.8.1, “Concurrent Modification and Execution of Instructions” on page 13.

Programming Note

Following are examples of how to make instruction storage consistent with data storage. Because the optimal instruction sequence to make instruction storage consistent with data storage may vary between systems, many operating systems will provide a system service to perform this function

Case 1: The given program does not modify instructions executed by another program nor does another program modify the instructions executed by the given program.

Assume that location X previously contained the instruction A0; the program modified one of more bytes of that location such that, in data storage, the location contains the instruction A1; and location X is wholly contained in a single cache block. The following instruction sequence will make instruction storage consistent with data storage such that if the ***isync*** was in location X-4, the instruction A1 in location X would be executed immediately after the ***isync***.

```
dcbst  X       #copy the block to main storage
sync           #order copy before invalidation
icbi   X       #invalidate copy in instr cache
isync          #discard prefetched instructions
```

Case 2: One or more programs execute the instructions that are concurrently being modified by another program.

Assume program A has modified the instruction at location X and other programs are waiting for program A to signal that the new instruction is ready to execute. The following instruction sequence will make instruction storage consistent with data storage and then set a flag to indicate to the waiting programs that the new instruction can be executed.

```
dcbst  X       #copy the block in main storage
sync           #order copy before invalidation
icbi   X       #invalidate copy in instr cache
sync           #order invalidation before store
               #  to flag
stw r0,flag(3) #set flag indicating instruction
               #  storage is now consistent
```

The following instruction sequence, executed by the waiting program, will prevent the waiting programs from executing the instruction at location X until location X in instruction storage is consistent with data storage, and then will cause any prefetched instructions to be discarded.

```
lwz    r0,flag(3)  #loop until flag = 1 (when 1
cmpwi  r0,1    #   is loaded, location X in
bne    $-8     #   instruction storage is
               #   consistent with location X
               #   in data storage)
isync          #discard any prefetched inst'ns
```

In the preceding instruction sequence any context synchronizing instruction (e.g., ***rfid***) can be used instead of ***isync***. (For Case 1 only ***isync*** can be used.)

For both cases, if two or more instructions in separate data cache blocks have been modified, the ***dcbst*** instruction in the examples must be replaced by a sequence of ***dcbst*** instructions such that each block containing the modified instructions is copied back to main storage. Similarly, for ***icbi*** the sequence must invalidate each instruction cache block containing a location of an instruction that was modified. The ***sync*** instruction that appears above between “***dcbst*** X” and “***icbi*** X” would be placed between the sequence of ***dcbst*** instructions and the sequence of ***icbi*** instructions.

1.8.1 Concurrent Modification and Execution of Instructions

The phrase “concurrent modification and execution of instructions” (CMODX) refers to the case in which a processor fetches and executes an instruction from instruction storage which is not consistent with data storage or which becomes inconsistent with data storage prior to the completion of its processing. This section describes the only case in which executing this instruction under these conditions produces defined results.

In the remainder of this section the following terminology is used.

- Location X is an arbitrary word-aligned storage location.
- X_0 is the value of the contents of location X for which software has made the location X in instruction storage consistent with data storage.
- X_1, X_2, ..., X_n are the sequence of the first n values occupying location X after X_0.
- X_n is the first value of X subsequent to X_0 for which software has again made instruction storage consistent with data storage.
- The “patch class” of instructions consists of the I-form *Branch* instruction (***b***[***l***][***a***]) and the preferred no-op instruction (***ori*** 0,0,0).

If the instruction from location X is executed after the copy of location X in instruction storage is made consistent for the value X_0 and before it is made consistent for the value X_n, the results of executing the instruction are defined if and only if the following conditions are satisfied.

1. The stores that place the values X_1, ..., X_n into location X are atomic stores that modify all four bytes of location X.
2. Each X_i, $0 \leq i \leq n$, is a patch class instruction.
3. Location X is in storage that is Memory Coherence Required.

If these conditions are satisfied, the result of each execution of an instruction from location X will be the execution of some X_i, $0 \leq i \leq n$. The value of the ordinate i associated with each value executed may be different and the sequence of ordinates i associated with a sequence of values executed is not constrained, (e.g., a valid sequence of executions of the instruction at location X could be the sequence X_i, X_{i+2}, then X_{i-1}). If these conditions are not satisfied, the results of each such execution of an instruction from location X are boundedly undefined, and may include causing inconsistent information to be presented to the system error handler.

Programming Note

An example of how failure to satisfy the requirements given above can cause inconsistent information to be presented to the system error handler is as follows. If the value X_0 (an illegal instruction) is executed, causing the system illegal instruction handler to be invoked, and before the error handler can load X_0 into a register, X_0 is replaced with X_1, an *Add Immediate* instruction, it will appear that a legal instruction caused an illegal instruction exception.

Programming Note

It is possible to apply a patch or to instrument a given program without the need to suspend or halt the program. This can be accomplished by modifying the example shown above where one program is creating instructions to be executed by one or more other programs.

In place of the *Store* to a flag to indicate to the other programs that the code is ready to be executed, the program that is applying the patch would replace a patch class instruction in the original program with a *Branch* instruction that would cause any program executing the *Branch* to branch to the newly created code. The first instruction in the newly created code must be an ***isync***, which will cause any prefetched instructions to be discarded, ensuring that the execution is consistent with the newly created code. The instruction storage location containing the ***isync*** instruction in the patch area must be consistent with data storage with respect to the processor that will execute the patched code before the *Store* which stores the new *Branch* instruction is performed.

Programming Note

It is believed that all processors that comply with versions of the architecture that precede Version 2.01 support concurrent modification and execution of instructions as described in this section if the requirements given above are satisfied, and that most such processors yield boundedly undefined results if the requirements given above are not satisfied. However, in general such support has not been verified by processor testing. Also, one such processor is known to yield undefined results in certain cases if the requirements given above are not satisfied.

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 Operating Environment Architecture* for a description of storage segments)

The *Move Assist* instructions have no alignment requirements.

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 <4	good poor	good poor	good poor	poor 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 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***, ***dcbf***[***l***]
12. Instruction cache associativity
13. Data cache associativity
14. Number of stream IDs supported for the stream variant of ***dcbt***
15. 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.

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”.

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, the definition of program order on page i, storage protection, reference and change recording, and the storage access ordering described in Section 1.7.1, “Storage Access Ordering” on page 7.

3.2.1 Instruction Cache Instruction

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)+(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, “Cache Management Instructions”), 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 that are 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 Operating Environment Architecture*).

Programming Note

The invalidation of the specified block need not have been performed with respect to the processor executing the ***icbi*** instruction until a subsequent ***isync*** instruction has been executed by that processor. No other instruction or event has the corresponding effect.

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)+(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 only operation that is "caused by" the ***dcbt*** instruction is the providing of the hint. 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 memory barriers.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

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

The processor's response to the hint provided by ***dcbt*** or ***dcbtst*** is to take actions that reduce the latency of subsequent loads or stores that access the specified block. (Such actions may include prefetching the block into levels of the storage hierarchy that are "near" the processor.)

Processors that comply with versions of the architecture that precede Version 2.00 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.

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)+(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 only operation that is "caused by" the ***dcbtst*** instruction is the providing of the hint. 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 memory barriers.

The ***dcbtst*** instruction may complete before the operation it causes has been performed.

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

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 + (RB)
n ← block size (bytes)
m ← $\log_2$(n)
ea ← $EA_{0:63-m} \,||\, {}^{m}0$
MEM(ea, n) ← $^{n}$0x00
```

Let the effective address (EA) be the sum (RA|0)+(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 Operating Environment Architecture* for additional information about ***dcbz***.

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)+(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 the block 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)+(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 the block 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 ensures that all instruction cache block invalidations caused by ***icbi*** instructions preceding the ***isync*** instruction have been performed with respect to the processor executing the ***isync*** instruction, and then causes any prefetched instructions to be discarded.

Except as described in the preceding sentence, 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 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 1.7.3, “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 “Fixed-Point Storage Access Instructions” in Book I, *PowerPC 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 +(RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← $^{32}0$ || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+(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 1.7.3.1 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

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 +(RB)
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
RT ← MEM(EA, 8)
```

Let the effective address (EA) be the sum (RA|0)+(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 1.7.3.1 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 + (RB)
if RESERVE then
  if RESERVE_ADDR = real_addr(EA) then
    MEM(EA, 4) ← (RS)$_{32:63}$
    CR0 ← 0b00 || 0b1 || XER$_{SO}$
  else
    u1 ← undefined 1-bit value
    if u1 then
      MEM(EA, 4) ← (RS)$_{32:63}$
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XER$_{SO}$
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER$_{SO}$
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists and the storage location specified by the ***stwcx.*** is the same as the location 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 the location 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 is set as follows. n is a 1-bit value that indicates whether the store was performed, except that if a reservation exists but the storage location specified by the ***stwcx***. is not the same as the location specified by the *Load And Reserve* instruction that established the reservation the value of n is undefined.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || XER_{SO}

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

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 + (RB)
if RESERVE then
  if RESERVE_ADDR = real_addr(EA) then
    MEM(EA, 8) ← (RS)
    CR0 ← 0b00 || 0b1 || XER$_{SO}$
  else
    u1 ← undefined 1-bit value
    if u1 then
      MEM(EA, 8) ← (RS)
    u2 ← undefined 1-bit value
    CR0 ← 0b00 || u2 || XER$_{SO}$
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER$_{SO}$
```

Let the effective address (EA) be the sum (RA|0)+(RB).

If a reservation exists and the storage location specified by the ***stdcx.*** is the same as the location 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 the location 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 is set as follows. n is a 1-bit value that indicates whether the store was performed, except when a reservation exists but the storage location specified by the stwcx. is not the same as the location specified by the *Load And Reserve* instruction that established the reservation, the value of n is undefined.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || n || XER_{SO}

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

3.3.3 Memory Barrier Instructions

The *Memory Barrier* instructions can be used to control the order in which storage accesses are performed. Additional information about these instructions and about related aspects of storage management can be found in Book III, *PowerPC 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 43.

Synchronize X-form

sync L

[POWER mnemonic: dcs]

31	///	L	///	///	598	/
0	6	9	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 field.

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, except that if a_i is the storage access caused by an ***icbi*** instruction then b_j may be performed with respect to the processor executing the ***sync*** instruction before a_i is performed with respect to that processor.

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.

L = 2

The set of storage accesses that is ordered by the memory barrier is described in the section entitled “Synchronize Instruction” in Book III, as are additional properties of the ***sync*** instruction with L=2.

The ordering done by the memory barrier is cumulative.

If L=0 (or L=2), the ***sync*** instruction has the following additional properties.

- Executing the ***sync*** instruction 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.
- The ***sync*** instruction is execution synchronizing (see Book III, *PowerPC Operating Environment Architecture*). However, address translation and reference and change recording (see Book III) associated with subsequent instructions may be performed before the ***sync*** instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a *Store* in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an ***icbi*** instruction executed by the processor executing the ***sync*** instruction (P1) may not have been performed with respect to P1 (see the description of the ***icbi*** instruction on page 18).

  The cumulative properties of the barrier apply to the execution of the given instruction as they would to a Load that returned a value that was the result of a *Store* in set B.
- The ***sync*** instruction provides an ordering function for the operations caused by ***dcbt*** instructions with TH_0 = 1.

The value L=3 is reserved.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed. The ***sync*** instruction may complete before operations caused by ***dcbt*** instructions with TH_0 = 1 preceding the ***sync*** instruction have been performed.

Special Registers Altered:

None

Extended Mnemonics:

Extended mnemonics for *Synchronize*:

Extended:	**Equivalent to:**
sync	sync 0
lwsync	sync 1
ptesync	sync 2

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; the appropriate extended mnemonics are used when other L values are intended.

Programming Note

Section 1.8, "Instruction Storage" on page 11 contains a detailed description of how to modify instructions such that a well-defined result is obtained.

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 47.

The memory barrier created by a ***sync*** instruction with L=0 or L=1 does not order implicit storage accesses. The memory barrier created by a ***sync*** instruction with any L value does not order instruction fetches.

(The memory barrier created by a ***sync*** instruction with L=0 -- or L=2; see Book III -- *appears* to order instruction fetches for instructions preceding the ***sync*** instruction with respect to data accesses caused by instructions following the ***sync*** instruction. However, this ordering is a consequence of the first "additional property" of ***sync*** with L=0, not a property of the memory barrier.)

In order to obtain the best performance across the widest range of implementations, the programmer should use either the ***sync*** instruction with L=1 or the ***eieio*** instruction if either of these is sufficient for his needs; otherwise he should use ***sync*** with L=0. ***sync*** with L=2 should not be used by application programs.

Programming Note

The functions provided by ***sync*** with L=1 are a strict subset of those provided by ***sync*** with L=0. (The functions provided by ***sync*** with L=2 are a strict superset of those provided by ***sync*** with L=0; see Book III.)

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, “Storage Access Ordering”), 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 operations caused by ***dcbt*** instructions with TH_0 = 1 are ordered by ***eieio*** as a third set of operations, which is independent of the two sets specified above.

The ***eieio*** instruction may complete before storage accesses associated with instructions preceding the ***eieio*** instruction have been performed. The ***eieio*** instruction may complete before operations caused by ***dcbt*** instructions having TH_0 = 1 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 45), 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 5).

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 7), ***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.

Programming 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	TBL	
0	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 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 1 GHz and that the Time Base is driven by this frequency divided by 32. Then the period of the Time Base would be

$$T_{\mathrm{TB}} = \frac{2^{64} \times 32}{1\mathrm{GHz}} = 5.90 \times 10^{11} \text{ seconds}$$

which is approximately 18,700 years.

The PowerPC 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 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.

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 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 Operating Environment Architecture*.

Move From Time Base XFX-form

mftb RT,TBR

31	RT	tbr	371	/
0	6	11	21	31

```
n ← $tbr_{5:9}$ || $tbr_{0:4}$
if n = 268 then
  RT ← TB
else if n = 269 then
  RT ← $^{32}0$ || $TB_{0:31}$
```

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*		**Register**
decimal	**$tbr_{5:9}$**	**$tbr_{0:4}$**	**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.

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 1 GHz 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

  $$\frac{1\ \text{GHz}}{32} = 31{,}250{,}000$$

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

  $$\frac{1{,}000{,}000{,}000}{31{,}250{,}000} = 32$$

  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 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: Real-time 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 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 Architecture. See the System Architecture documentation for a given PowerPC 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 18.

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 + (RB)
raddr ← address translation of EA
send load word request for raddr to
   device identified by RID
RT ← $^{32}$0 || word from device
```

Let the effective address (EA) be the sum (RA|0)+(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 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.

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 + (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)+(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 Operating Environment Architecture* for additional information about this instruction.

Special Registers Altered:
None

5.2 Storage Control Instructions

5.2.1 Cache Management Instructions

5.2.1.1 Data Cache Instructions

The optional version of the *Data Cache Block Touch* instruction permits a program to provide a hint regarding a sequence of contiguous data cache blocks. Such a sequence is called a “data stream”. A ***dcbt*** instruction in which $TH_{0\ 3} \neq 0b00$ is said to be a “data stream variant” of ***dcbt***. In the remainder of this section, “data stream” may be abbreviated to “stream”.

Warning: The variants of ***dcbt*** in which $TH_3 = 1$ are being phased out of the architecture.

The optional version of the *Data Cache Block Flush* instruction permits a program to limit the scope of the "flush" operation.

Extended mnemonics for Data Cache Block Flush

Extended mnemonics are provided for the *Data Cache Block Flush* 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 43.

Data Cache Block Touch X-form

dcbt RA,RB,TH

31	/	TH	RA	RB	278	/
0	6	7	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

The ***dcbt*** instruction provides a hint that describes a block or data stream, or indicates the expected use thereof. The nature of the hint depends, in part, on the value of bit 0 of the TH field. A hint that the program will probably soon load from a given storage location is ignored if the location is Caching Inhibited or Guarded.

When, and how often, effective addresses for a data stream are translated is implementation-dependent.

When $TH_0 = 0$ the ***dcbt*** instruction provides a hint that the program will probably soon load from the specified block or data stream.

The encodings of the TH field in which $TH_0 = 0$ are as follows.

TH **Description**

0000 The program will probably soon load from the block containing the byte addressed by EA.

0001 The program will probably soon load from the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially following blocks.

0011 The program will probably soon load from the data stream consisting of the block containing the byte addressed by EA and an unlimited number of sequentially preceding blocks.

When $TH_0 = 1$ the ***dcbt*** instruction provides a hint that describes a data stream, or indicates that the program will probably soon load from data streams that have been thus described or will probably no longer load from such data streams.

The address and length of such data streams are specified in terms of aligned 128-byte units of storage; in the remainder of this instruction description, “aligned 128-byte unit of storage” is abbreviated to “unit”.

Each such data stream is associated, by software, with a stream ID, which is a resource that the processor uses to distinguish the data stream from other such data streams. The number of stream IDs is an implementation-dependent value in the range 1:16 (see the Book IV, *PowerPC AS Implementation Features* document for the implementation). Stream IDs are numbered sequentially starting from 0.

Programming Note

Architecturally, stream IDs are not used for data streams that are specified using ***dcbt*** instructions in which $TH_3 = 1$, or for “hardware-detected data streams” (defined in a subsequent Programming Note). Therefore, elsewhere in this instruction description, references to data streams associated with stream IDs apply only to data streams that are specified using ***dcbt*** instructions in which $TH_0 = 1$.

The encodings of the TH field in which TH$_0$ = 1, and of the corresponding EA values, are as follows. In the EA layout diagrams, fields shown as “/”s are reserved. These fields, and reserved values of defined EA fields, are treated in the same manner as the corresponding cases for instruction fields (see the section entitled "Reserved Fields and Reserved Values" in Book I), except that a reserved value in a defined EA field does not make the instruction form invalid. If a defined EA field contains a reserved value, the hint provided by the instruction is undefined.

TH Description

1000 The ***dcbt*** instruction provides a hint that describes certain attributes of a data stream, and optionally indicates that the program will probably soon load from the stream.

The EA is interpreted as follows.

EATRUNC	D	UG	/	ID
0	57		59	60 63

Bit(s) Description

0:56 ***EATRUNC***

High-order 57 bits of effective address of first unit of data stream (i.e., the effective address of the first unit of the stream is EATRUNC || $^{7}0$)

57 ***Direction*** (D)

0 Subsequent units are the sequentially following units.

1 Subsequent units are the sequentially preceding units.

58 ***Unlimited/GO*** (UG)

0 No information is provided by the UG field.

1 The number of units in the data stream is unlimited, the program’s need for each block of the stream is not likely to be transient, and the program will probably soon load from the stream.

59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream

1010 The ***dcbt*** instruction provides a hint that describes certain attributes of a data stream, or indicates that the program will probably soon load from data streams that have been described using ***dcbt*** instructions in which TH$_0$ = 1 or will probably no longer load from such data streams.

The EA is interpreted as follows. If GO = 1 and $S \neq$ 0b00 the hint provided by the instruction is undefined; the remainder of this instruction description assumes that this combination is not used.

///	GO	S	///	UNITCNT	T	U	/	ID
0	32		35	47	57		59	60 63

Bit(s) Description

0:31 Reserved

32 ***GO***

0 No information is provided by the GO field.

1 The program will probably soon load from all nascent data streams that have been completely described, and will probably no longer load from all other nascent data streams. All other fields of the EA are ignored. (“Nascent” and “completely described” are defined below.)

33:34 ***Stop*** (S)

00 No information is provided by the S field.

01 Reserved

10 The program will probably no longer load from the data stream (if any) associated with the specified stream ID. (All other fields of the EA except the ID field are ignored.)

11 The program will probably no longer load from the data streams associated with all stream IDs. (All other fields of the EA are ignored.)

35:46 Reserved

47:56 ***UNITCNT***

Number of units in data stream

57 ***Transient*** (T)

If T=1, the program’s need for each block of the data stream is likely to be transient (i.e., the time interval during which the program accesses the block is likely to be short).

58 ***Unlimited*** (U)

If U=1, the number of units in the data stream is unlimited (and the UNITCNT field is ignored).

59 Reserved

60:63 ***Stream ID*** (ID)

Stream ID to use for this data stream (GO = 0 and S = 0b00), or stream ID associated with the data stream from which the program will probably no longer load (S = 0b10)

If the specified stream ID value is greater than m -1, where m is the number of stream IDs provided by the implementation, and either (a) TH = 0b1000 or (b) TH = 0b1010 and GO = 0 and S ≠ 0b11, no hint is provided by the instruction.

All TH values that are not shown above are reserved. If the TH field contains a reserved value, the hint provided by the instruction is undefined.

The only operation that is performed by the ***dcbt*** instruction is the providing of the hint. 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.

The ***dcbt*** instruction may complete before the operation it causes has been performed.

If TH ≠ 0b1010 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

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 0b0000.

Programming Note

If TH = 0b0000, the instruction operates as described in Section 3.2.2, “Data Cache Instructions” on page 19.

Programming Note

Warning: The variants of ***dcbt*** in which TH_3 = 1 are being phased out of the architecture. On future implementations these variants are likely to be treated as if TH = 0b0000. On implementations that support both kinds of data stream variant (TH_0 = 1 and TH_3 = 1), performance is likely to be degraded if a program uses both kinds. Therefore the variants in which TH_3 = 1 should be used only in programs that will run on implementations that do not support the variants in which TH_0 = 1. For other programs, each existing instance of a ***dcbt*** instruction with TH = 0b00d1 can be replaced by a ***dcbt*** instruction with TH = 0b1000, D = d, and UG = 1.

The remainder of this instruction description applies only to *dcbt* instructions in which TH_0 = 1 and to data streams specified thereby. Corresponding material for ***dcbt*** instructions in which TH = 0b0000 can be found in the ***dcbt*** instruction description in Section 3.2.2; corresponding material for ***dcbt*** instructions in which TH_3 = 1 can be found in Section 5.2.1.1 of Version 2.01 of the architecture specification.

The following terminology is used to describe the state of a data stream. Except as described in the paragraph after the next paragraph, the state of a data stream at a given time is determined by the most recently provided hint for the stream.

- A data stream for which only descriptive hints have been provided (by ***dcbt*** instructions with TH = 0b1000 and UG = 0 or with TH = 0b1010 and GO = 0 and S = 0b00) is said to be “nascent”. A nascent data stream for which both kinds of descriptive hint have been provided (by both of the ***dcbt*** usages listed in the preceding sentence) is considered to be “completely described”.
- A data stream for which a hint has been provided (by a ***dcbt*** instruction with TH = 0b1000 and UG = 1 or with TH = 0b1010 and GO = 1) that the program will probably soon load from it is said to be “active”.
- A data stream that is either nascent or active is considered to “exist”.
- A data stream for which a hint has been provided (e.g., by a ***dcbt*** instruction with TH = 0b1010 and S ≠ 0b00) that the program will probably no longer load from it is considered no longer to exist.

The hint provided by a ***dcbt*** instruction with TH = 0b1000 and UG = 1 implicitly includes a hint that the program will probably no longer load from the data stream (if any) previously associated with the specified stream ID. The hint provided by a ***dcbt*** instruction with TH = 0b1000 and UG = 0 or with TH = 0b1010 and GO = 0 and S = 0b00 implicitly includes a hint that the program will probably no longer load from the *active* data stream (if any) previously associated with the specified stream ID.

Interrupts (see Book III) cause all existing data streams to cease to exist. In addition, depending on the implementation, certain conditions and events may cause an existing data stream to cease to exist.

Programming Note

Processors that support only the two-bit version of the TH field occupying instruction bits 9:10 (i.e. those that support versions 2.00 or 2.01 of the architecture, e.g. POWER4 and POWER4+) treat both TH = 0b1000 and TH = 0b1010 as if TH = 0b0000. (For those processors and versions of the architecture, TH = 0b10 was a reserved value and was treated as if TH = 0b00.)

Processors that do not support the TH field at all (i.e. those that comply with versions of the architecture that precede Version 2.00) treat both TH = 0b1000 and TH = 0b1010 as if TH = 0b0000. (For those processors and versions of the architecture, the TH field was reserved and ignored, so that all instances of ***dcbt*** would behave as specified in Section 3.2.2.) However, note that whereas newer processors will ignore any ***dcbt*** for which the specified block is Caching Inhibited or Guarded, processors complying with versions of the architecture that precede Version 2.00 do not necessarily ignore the hint if the specified block is Guarded and not Caching Inhibited.

Programming Note

To obtain the best performance across the widest range of implementations that support the variants of ***dcbt*** in which $TH_0 = 1$, the programmer should assume the following model when using those variants.

- The processor's response to a hint that the program will probably soon load from a given data stream is to take actions that reduce the latency of loads from the first few blocks of the stream. (Such actions may include prefetching the blocks into levels of the storage hierarchy that are "near" the processor.) Thereafter, as the program loads from each successive block of the stream, the processor takes latency-reducing actions for additional blocks of the stream, pacing these actions with the program's loads (i.e., taking the actions for only a limited number of blocks ahead of the block that the program is currently loading from).

  The processor's response to a hint that the program will probably no longer load from a given data stream, or to the cessation of existence of a data stream, is to stop taking latency-reducing actions for the stream.
- A data stream having finite length ceases to exist when the latency-reducing actions have been taken for all blocks of the stream.
- If the program ceases to need a given data stream before having loaded from all blocks of the stream (always the case for streams having unlimited length), performance may be improved if the program then provides a hint that it will no longer load from the stream (e.g., by executing the appropriate ***dcbt*** instruction with TH = 0b1010 and S ≠ 0b00).
- At each level of the storage hierarchy that is "near" the processor, blocks of a data stream that is specified as transient are most likely to be replaced. As a result, it may be desirable to stagger addresses of streams (choose addresses that map to different cache congruence classes) to reduce the likelihood that a unit of a transient stream will be replaced prior to being accessed by the program.
- On some implementations, data streams that are not specified by software may be detected by the processor. Such data streams are called "hardware-detected data streams". On some such implementations, data stream resources (resources that are used primarily to support data streams) are shared between software-specified data streams and hardware-detected data streams. On these latter implementations, the programming model includes the following.
  - Software-specified data streams take precedence over hardware-detected data streams in use of data stream resources.
  - The processor's response to a hint that the program will probably no longer load from a given data stream, or to the cessation of existence of a data stream, includes releasing the associated data stream resources, so that they can be used by hardware-detected data streams.

Programming Note

This Programming Note describes several aspects of using ***dcbt*** instructions in which TH_0 = 1.

- A non-transient data stream having unlimited length can be completely specified, including providing the hint that the program will probably soon load from it, using one ***dcbt*** instruction. The corresponding specification for a data stream having other attributes requires three ***dcbt*** instructions. However, one ***dcbt*** instruction with TH = 0b1010 and GO = 1 can apply to a set of the data streams described in the preceding sentence, so the corresponding specification for n such data streams requires 2×n + 1 ***dcbt*** instructions. (There is no need to execute a ***dcbt*** instruction with TH = 0b1010 and S = 0b10 for a given stream ID before using the stream ID for a new data stream; the implicit portion of the hint provided by ***dcbt*** instructions that describe data streams suffices.)
- If it is desired that the hint provided by a given ***dcbt*** instruction be provided in program order with respect to the hint provided by another ***dcbt*** instruction, the two ***dcbt*** instructions must be separated by an ***eieio*** (or ***sync***) instruction. For example, if a ***dcbt*** instruction with TH = 0b1010 and GO = 1 is intended to indicate that the program will probably soon load from nascent data streams described (completely) by preceding ***dcbt*** instructions, and is intended *not* to indicate that the program will probably soon load from nascent data streams described (completely) by *following* ***dcbt*** instructions, an ***eieio*** instruction must separate the ***dcbt*** instruction with GO = 1 from the preceding ***dcbt*** instructions, and another ***eieio*** instruction must separate that ***dcbt*** instruction from the following ***dcbt*** instructions.
- In practice, the second ***eieio*** described above can sometimes be omitted. For example, if the program consists of an outer loop that contains the ***dcbt*** instructions and an inner loop that contains the *Load* instructions that load from the data streams, the characteristics of the inner loop and of the implementation's branch prediction mechanisms may make it highly unlikely that hints corresponding to a given iteration of the outer loop will be provided out of program order with respect to hints corresponding to the previous iteration of the outer loop. (Also, any providing of hints out of program order affects only performance, not program correctness.)
- To mitigate the effects of interrupts on data streams, it may be desirable to specify a given "logical" data stream as a sequence of shorter, component data streams. Similar considerations apply to conditions and events that, depending on the implementation, may cause an existing data stream to cease to exist; see Book IV.
- If it is desired to specify data streams without regard to the number of stream IDs provided by the implementation, stream IDs should be assigned to data streams in order of decreasing stream importance (stream ID 0 to the most important stream, stream ID 1 to the next most important stream, etc.). This order ensures that the hints for the most important data streams will be provided.

Data Cache Block Flush X-form

dcbf RA,RB,L

31	///	L	RA	RB	86	/
0	6	10	11	16	21	31

Let the effective address (EA) be the sum (RA|0)+(RB).

L = 0

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 the block 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.

L = 1 ("dcbf local")

If the 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. If L = 1, the function of this instruction is also independent of whether the block containing the byte addressed by EA is in storage that is Memory Coherence Required.

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

Extended Mnemonics:

Extended Mnemonics for *Data Cache Block Flush*:

Extended:	**Equivalent to:**
dcbf RA,RB	dcbf RA,RB,0
dcbfl RA,RB	dcbf RA,RB,1

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

Programming Note

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

Programming Note

dcbf with L = 1 can be used to cause a block that will not be reused soon to be removed from the processor's data cache, and thereby potentially to cause that data cache to be used more efficiently.

Programming Note

The functions provided by ***dcbf*** with L = 1 are identical to those that would be provided if L were 0 and the specified block were in storage that is not Memory Coherence Required.

5.3 Little-Endian

If the optional Little-Endian facility is implemented (see the section entitled “Little-Endian” in Book I, *PowerPC 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 Data Cache Block Flush Mnemonics

The L field in the optional version of the *Data Cache Block Flush* instruction controls the scope of the flush function performed by the instruction. Extended mnemonics are provided that represent the L value in the mnemonic rather than requiring it to be coded as a numeric operand.

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

dcbf RA,RB	(equivalent to: dcbf RA,RB,0)
dcbfl RA,RB	(equivalent to: dcbfl RA,RB,1)

A.2 Synchronize Mnemonics

The L field in the *Synchronize* instruction controls the scope of the synchronization function performed by the instruction. 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)
ptesync	(equivalent to:	sync	2)

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 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*) followed 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 “-” suffixes 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 implement 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 1.7.3.1) 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 Virtual Environment Architecture, sorted by POWER mnemonic.

To determine the PowerPC 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 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	
	Mnemonic	Instruction	Mnemonic	Instruction
20	dclz	Data Cache Line Set to Zero	dcbz	Data Cache Block set to Zero
26	dcs	Data Cache Synchronize	sync	Synchronize
22	ics	Instruction Cache Synchronize	isync	Instruction Synchronize

Appendix D. New Instructions

The following instructions in the PowerPC 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 Virtual Environment Instruction Set

Form	Opcode		Mode Dep.[1]	Page	Mnemonic	Instruction
	Primary	Extend				
X	31	86		21, 41	dcbf	Data Cache Block Flush
X	31	54		21	dcbst	Data Cache Block Store
X	31	278		19, 35	dcbt	Data Cache Block Touch
X	31	246		19	dcbtst	Data Cache Block Touch for Store
X	31	1014		20	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		28	eieio	Enforce In-order Execution of I/O
X	31	982		18	icbi	Instruction Cache Block Invalidate
XL	19	150		22	isync	Instruction Synchronize
X	31	84		24	ldarx	Load Doubleword And Reserve Indexed
X	31	20		24	lwarx	Load Word And Reserve Indexed
XFX	31	371		30	mftb	Move From Time Base
X	31	214		25	stdcx.	Store Doubleword Conditional Indexed
X	31	150		25	stwcx.	Store Word Conditional Indexed
X	31	598		26	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 Virtual Environment Architecture are independent of whether the processor is in 32-bit or 64-bit mode.

Index

Last Page - End of Document