POWER8 Processor User's Manual for the Single-Chip Module

Advance

April 22, 2014
Version 1.0

Printed in the United States of America April 2014

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com**®.

Version 1.0
April 22, 2014

Contents

List of Tables

List of Figures

Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was modified from the previous release of this document.

Revision Date	Description
April 22, 2014	Version 1.0 (initial version).

About this Document

This user's manual describes the IBM® POWER8™ processor. This document provides information about the registers, facilities, initialization, and use of the POWER8 processor.

This document provides information about the POWER8 processor that is visible from a programming model point of view, and is intended to be a companion to the baseline architecture documentation (see *Related Documents* on page 20). While there are some programming model considerations associated with chips and subsystems outside of the Central Electronics Complex (CEC), this document focuses primarily on the microprocessor core and the storage subsystem. For information about other chips that might appear in POWER8 systems, see the functional specifications for these individual chips.

Who Should Read this Document

This manual is intended for system software and hardware developers and application programmers who want to develop products for the POWER8 processor. It is assumed that the reader understands operating systems, microprocessor system design, basic principles of reduced instruction set computer (RISC) processing, and details of the Power ISA.

Conventions Used in This Document

This section explains numbers, bit fields, instructions, and signals that are in this document.

Representation of Numbers

Numbers are generally shown in decimal format, unless designated as follows:

- Hexadecimal values are preceded by an "x" and enclosed in single quotation marks.
  For example: x'0A00'.
- Binary values in sentences are shown in single quotation marks.
  For example: '1010'.

Note: A bit value that is immaterial, which is called a "don't care" bit, is represented by an "X."

Bit Significance

In the POWER8 documentation, the smallest bit number represents the most significant bit of a field, and the largest bit number represents the least significant bit of a field.

Other Conventions

POWER8 instruction mnemonics are shown in lower-case, bold text. For example: **tlbivax**. I/O signal names are shown in upper case.

Related Documents

The following documents can be helpful when reading this specification. Contact your IBM representative to obtain any documents that are not available through IBM Customer Connect or Power.org.

POWER8 Processor Single-Chip Module Datasheet

POWER8 Memory Buffer Datasheet

POWER8 Memory Buffer User's Manual

Power ISA User Instruction Set Architecture - Book I (Version 2.07)

Power ISA Virtual Environment Architecture - Book II (Version 2.07)

Power ISA Operating Environment Architecture (Server Environment) - Book III-S (Version 2.07)

Power ISA Transactional Memory (Version 2.07)

PowerPC Architecture Platform Requirements (PAPR+) Specification

PCI Express Base Specification, Revision 3.0

1. POWER8 Processor Overview

The POWER8 processor is a superscalar symmetric multiprocessor designed for use in servers and large-cluster systems. It uses IBM CMOS 22 nm SOI technology with 15 metal layers.

1.1 General Features

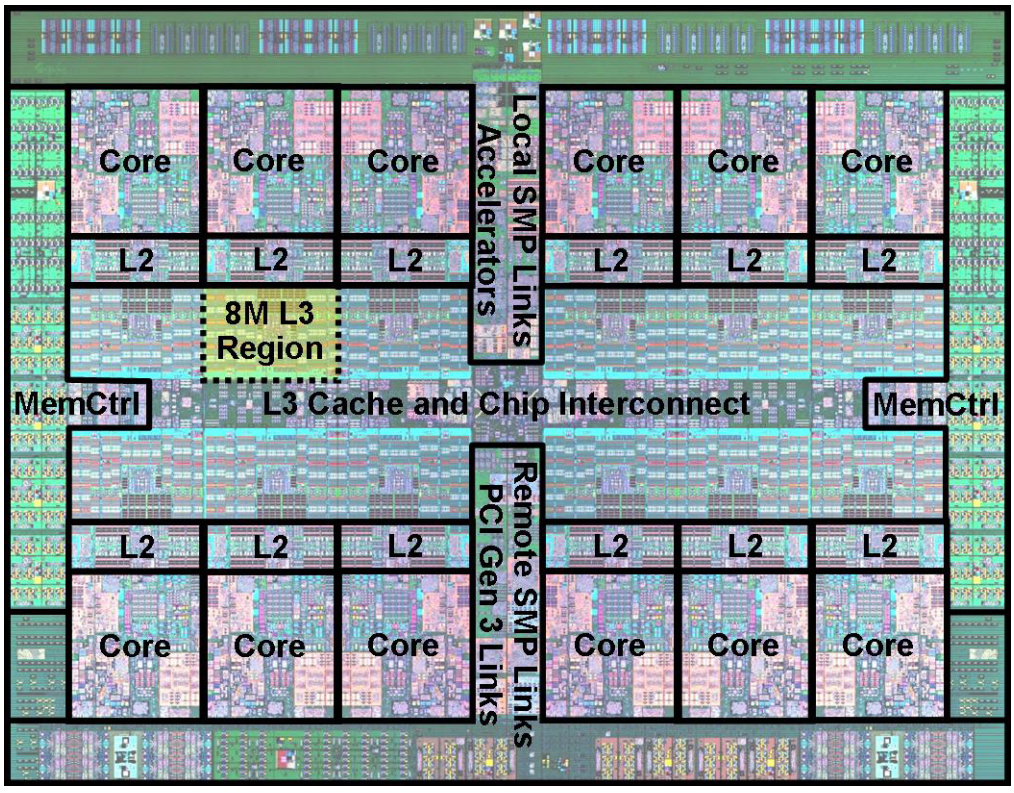
The POWER8 processor can have up to 12 cores enabled on a single chip in the single-chip module (SCM) configuration. Each core has eight threads using simultaneous multithreading (SMT). The SMT is dynamically tunable, so that each core can have one, two, four, or eight threads.

The POWER8 processor supports the following architectural features:

- PowerPC Architecture Book I, II, and III version 2.07
- PowerPC Architecture Platform Requirements (PAPR+), Version 2.1
- IEEE P754-2008 floating-point compliant
- Big-endian, little-endian, strong-ordering support extension
- 50-bit real address, 68-bit virtual address

Figure 1-1 provides a block diagram of the POWER8 processor on the SCM.

Figure 1-1. Block Diagram for the POWER8 Processor



The following features describe the main components of the 12-core POWER8 processor:

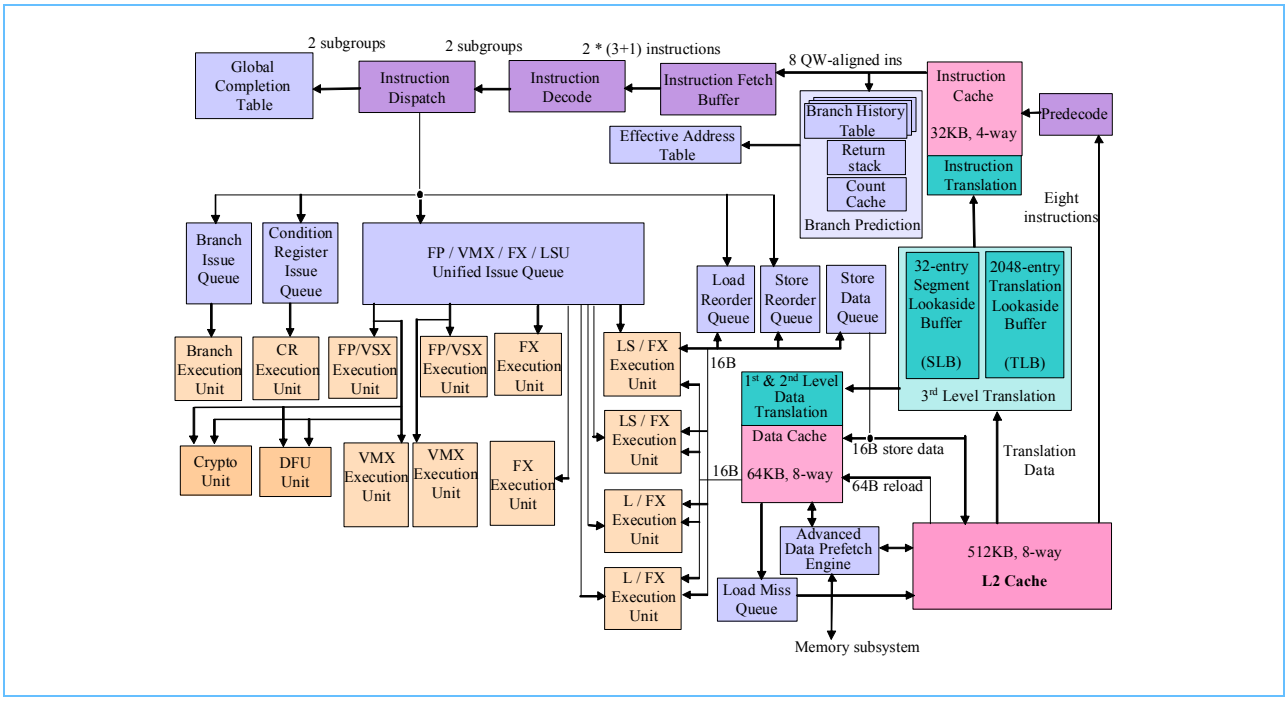
- POWER8 core and cache
  - Up to 12 processor cores
  - 16 execution pipes
  - 8-way SMT, Out-of-Order (O-o-O)
  - 124 × 2 GPR and 144 × 2 VMX/VSX/FPR renames
  - Software-architected register file
  - Concurrent support of 1 - 4 LPARs per core
- Chiplet
  - 32 KB instruction cache (I-cache)
  - 64 KB data cache (D-cache)
  - 512 KB private L2 cache
  - Local 8 MB L3 cache region
- POWER8 SMP on-chip interconnect
  - Eight 16-byte data buses, two address snoop buses, 32 on/off ramps
  - Asynchronous interface to core/L2/L3 and off-chip interconnect
- Four 9.6 GHz differential memory interfaces (each with 2-byte read and1-byte write) to the POWER8 Memory Buffer chip
- POWER8 SMP off-chip interconnect
  - Maximum 48-way SMP
  - Three 6.4 GHz differential SMP interfaces (each with 2 bytes per direction)
  - 8 GHz differential PCIe Generation 3 buses (32 lanes configured as 16× + 16×, or 16× + 8× + 8×)
- Coherent Accelerator Interface Architecture (CAPI) allows an FPGA or ASIC to connect coherently to the POWER8 processor SMP interconnect via the PCIe.
- Power management support
  - Hypervisor-directed power change requests using a PState mechanism
  - Sensors
    - Digital thermal sensor (DTS2) ±5°C
    - Off-chip analog thermal diode ±1 - 2°C
    - Dedicated performance, microarchitecture, and event counters
  - On-chip controller (OCC)
    - On-chip PowerPC 405 for real-time frequency and voltage modification
    - On-chiplet hardware assist (automated core chiplet management)
    - On-chip power management controls automated communications to the voltage regulation modules (VRMs) anf voltage and frequency sequencers for automated Pstate and idle state support
  - Actuators
    - Per-chiplet frequency control through the DPLL
    - Per-chiplet internal VRMs for independent voltage control
    - Architected idle states: nap, sleep, and winkle; each with increasing power savings capability (and latency)

    - SPR power management control registers (PMCR, PMICR, PMSR) for hypervisor support
  - Memory/DIMM throttling for memory subsystem power and thermal management
- Features
  - On-chip accelerators
    - On-chip: compression, encryption, data move initiated by hypervisor
    - In-core: user invocation encryption (AES, SHA)
  - Cloud computing enhancements: page replacement/affinity assist, micropartition prefetching, IPL time reduction, four concurrent LPARs per core
  - Transactional memory
  - Random number generator
  - RAID6 support in VMX
  - Support for industry standard BMC
  - Multi-level TCE support
  - Turbo mode support

2. POWER8 Processor Core

This section provides an overview of the POWER8 processor core, including key design fundamentals, an overview of the master pipeline operation, and a detailed summary of key design features.

Figure 2-1. POWER8 Processor Core



2.1 Key Design Fundamentals

This section describes the key design fundamentals of the POWER8 processor core.

2.1.1 64-Bit Implementation of the Power ISA (version 2.07)

- Compatibility for all Power ISA application-level code (problem state)
  - Supports partition mobility
- Supports IEEE P754-2008 for binary floating-point arithmetic
- Support for 32-bit operating system bridge facility

2.1.2 Layered Implementation Strategy for High-Frequency Operation

- Deeply pipelined design:
  - 16 stages from I-cache access to writeback for most fixed-point register-to-register operations
  - 18 stages for most load/store operations (assuming an L1 D-cache hit) from I-cache to writeback
  - 23 stages for most floating-point operations from I-cache access to writeback
- Dynamic instruction cracking[1] for some instructions allows for simpler inner core data flow:
  - Dedicated data flow for cracking one instruction into two or more internal operations
  - Microcoded templates for longer emulation sequences

2.1.3 Speculative Superscalar Inner Core Organization

- Multi-threaded core design:
  - Single thread (ST), 2-way multithread (SMT2), 4-way multithread (SMT4), and 8-way multithread (SMT8) modes supported in single LPAR mode.
  - (SMT8) supported in 2 and 4 LPAR mode.
- Aggressive branch prediction:
  - Prediction for up to eight branches per cycle
  - Support for up to 24 predicted taken branches in flight per thread in ST and SMT2 mode, 12 predicted taken branches in SMT4 mode, and six predicted taken branches in SMT8 mode. The number of predicted not-taken branches tracked can be higher.
  - Prediction support for branch direction and branch target addresses
- In single-thread mode, in-order dispatch of up to eight internal operations (iops) into distributed issue queues per cycle:
  - Up to two branches in a dispatch group; the first branch can be predicted taken or not-taken
  - Up to six non-branch instructions in the dispatch group
  - Second branch terminates the group
  - No VS-routed instruction after a branch in the group
- In SMT2 and beyond there are two dispatch sets each with
  - Up to one branch in dispatch group, the first branch can be predicted taken or not-taken
  - Up to three non-branch instructions in the dispatch group
  - No VS routed instruction after a branch in the group
- Out-of-order issue of up to 10 operations into the following 10 issue ports:
  - Two ports to do loads or fixed-point operations.
  - Two ports to do stores, fixed-point loads, or fixed-point operations.
  - Two fixed-point operations
  - Two issue ports shared by two floating-point, two VSX, two VMX, one crypto, and one DFP operations
  - One branch operation
  - One condition register operation
- Register renaming on GPRs, FPRs, VRFs (VMX and VSX Registers), CR fields, XER (parts), FPSCR, VSCR, Link and Count Registers

1.Process by which some complex instructions are broken into multiple simpler, more RISC-like instructions.

- Sixteen execution units:
  - Two symmetric load/store units (LSU), capable of executing stores, fixed-point loads, and simple fixed-point operations
  - Two load-only units (LU) also capable of executing simple fixed-point operations
  - Two symmetric fixed-point units (FXU)
  - Four floating-point units (FPU), implemented as two 2-way SIMD operations for double- and single-precision. Scalar binary floating-point instructions can only use two FPUs.
  - Two VMX execution units capable of executing simple FX, permute, complex FX, and 4-way SIMD single-precision floating-point operations
  - One Crypto unit
  - One decimal floating-point unit (DFU)
  - One branch execution unit (BR)
  - One condition register logical execution unit (CRL)
- Large number of instructions in flight:
  - Up to 64 instructions in the instruction fetch buffer in ST mode and up to 128 instructions total in SMT2, SMT4, and SMT8 modes (up to 64 instructions per thread in SMT2 mode, 32 instructions per thread in SMT4, and 16 instructions per thread in SMT8)
  - Up to 48 instructions in 3 decode pipe stages and 3 dispatch buffers
  - Up to 224 instructions in the inner-core (after dispatch)
  - Up to 40 stores queued in the SRQ (available for forwarding), shared by the available threads
- Fast, selective flush of incorrect speculative instructions and results

2.1.4 Specific Focus on Storage Latency Management

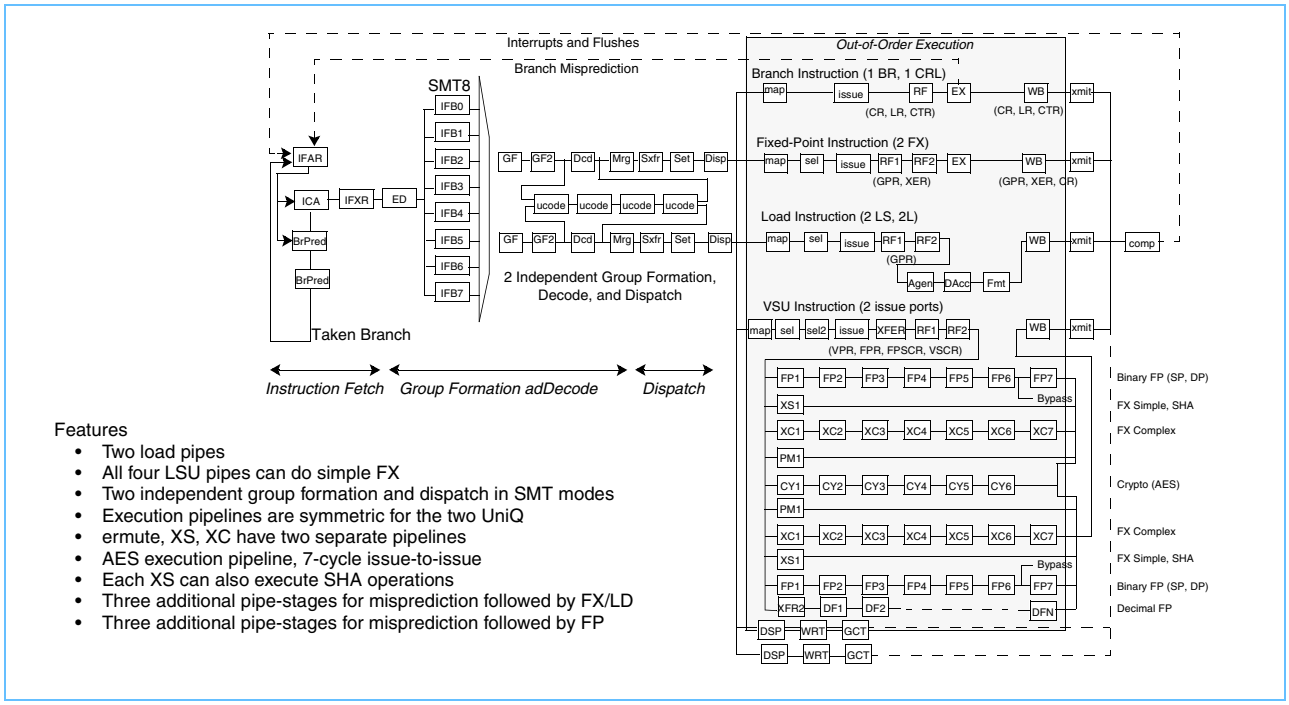
- Out-of-order and speculative issue of load operations
- Support for up to 16 outstanding L1 cache-line misses
- Hardware- or software-initiated instruction prefetching from L2 cache, L3 cache, and memory
- Hardware-initiated data-stream prefetching (using effective addresses). Support for up to 12 active streams
- Critical word forwarding, critical sector first
- Hardware instruction prefetching supported

2.2 Pipeline Structure

The pipeline structure for the processor can be subdivided into a *master pipeline* and several different *execution unit pipelines*. The master pipeline presents speculative in-order instructions to the mapping, sequencing, and dispatch functions. It ensures an orderly completion of the real execution path (throwing away any other potential speculative results associated with mispredicted paths). The execution unit pipelines allow out-of-order issuing of both speculative and non-speculative operations. The execution unit pipelines progress independently from the master pipeline and from one another.

Figure 2-2 illustrates these pipelines, where each box represents a pipeline stage.

Figure 2-2. Pipeline Structure



Features

- Two load pipes
- All four LSU pipes can do simple FX
- Two independent group formation and dispatch in SMT modes
- Execution pipelines are symmetric for the two UniQ
- ermute, XS, XC have two separate pipelines
- AES execution pipeline, 7-cycle issue-to-issue
- Each XS can also execute SHA operations
- Three additional pipe-stages for misprediction followed by FX/LD
- Three additional pipe-stages for misprediction followed by FP

The legend for *Figure 2-2* is as follows:

IFAR	Instruction fetch address register
ICA	Instruction cache access
ixfer	Instruction transfer
ED	Early decode cycle
D0	IDU predecode stage and instruction-fetch buffer latches
GF and GF2	Dispatch group determination
Dcd	Main decoder
Mrg	For assembly
ucode	Microcode
gxfer	Microcode selection and group transfer
Disp	Inter-dependency determination for instructions in the group
map	Register mapping
Sel	Issue queue selection
Issue	Instruction Issue
RF and RF2	Register file access
EX	Execution
WB	Writeback to the register file

EA/CA	Effective-address generation and data-cache decode
CA/fmt	Data-cache access and data formatting
FP1	Floating-point alignment and multiply
FP2	Floating-point alignment and multiply
FP3	Floating-point add
FP4	Floating-point add
FP5	Floating-point normalize result
FP6	Floating-point round and local 6-cycle forwarding
xmit	Finish and transmit
DSP	Group dispatch
WRT	Format and write into the GCT
GCT	Global completion table
Comp	Group completion
For VMX operations	
FP1 - FP6	Pipeline stages for the 4-way SIMD single-precision pipeline stages
XS1	Simple FX operation stage
XC1 - XC6	Complex FX operation stages
PM1/PM2	Permute stages

The processor core is divided into following seven units:

- IFU - Instruction fetch and decode unit
- ISU - Instruction dispatch and issue unit
- LSU - Load/store unit
- FXU - Fixed-point execution unit
- VSU - Vector and scalar unit (consists of VMX, binary floating-point, Crypto, and VSX)
- DFU - Decimal floating-point unit
- PC - Pervasive core unit

2.3 Microprocessor Core - Detailed Features

See *Section 10.1.11 Instruction Fusion* on page 226, *Section 10.1.12 Instruction Dispatch* on page 227, and *Section 10.1.13 Instruction Issue* on page 227 for additional details.

2.3.1 Instruction Fetching and Branch Prediction

- 32 KB, 8-way set-associative I-cache:
  - 128-byte lines (broken into four 32-byte sectors).
  - Dedicated 64-byte interface from the L2 cache that can supply 64 bytes in every other processor clock
  - Critical-sector-first reload policy
  - Effective-address index, real-address tags.

  - Banked I-cache, supports one read and one write per cycle when there is no bank conflict.
  - Eight additional predecode bits per word to aid in fast decoding and group formation.
  - Parity protected; force invalidate and reload on parity error.
- 64-entry effective-to-real address (ERAT) translation cache, fully associative.
  - Each entry can translate 4 KB, 64 KB, or 16 MB pages. For MSR[IR] = '1' and VRMA accesses, 16 MB and 16 GB pages take multiple 64 KB entries (the same is true for 1 MB pages; however, 1 MB pages are not visible to server workloads).
  - In MSR[IR] = '0' mode and non-VRMA accesses, 16 MB and 16 GB pages are installed as 4 KB translation blocks in the I-ERAT (the same is true for 1 MB pages).
  - In SMT mode, each entry is tagged to indicate invalid, valid for thread 0, valid for thread 1, valid for thread 2, valid for thread 3, valid for thread 4, valid for thread 5, valid for thread 5, or valid for thread 7.
- Fetch quadword aligned block of eight instructions per cycle.
  - In ST mode, instructions are fetched from the thread in every cycle
  - In all other modes, instructions are fetched from a given thread based on thread priority. If the threads are of equal priority, each thread gets approximately an equal number of fetch cycles, while optimizing the core throughput.
- Branch prediction:
  - Scan all eight fetched instructions for branches in each cycle
  - Predict up to eight branches per cycle (if the first one is predicted, fall-through)
  - Three table prediction structures: global, local, and selector (16K entries $\times$ 2 bits, 16K entries $\times$ 2 bits and 16K entries $\times$ 2 bits, respectively).
    - BHT tables are 16-way banked for concurrent read and write, when there is no bank conflict.
    - Global BHT is accessed using 20 bits of past global fetch history (folded into 11 bits to access global BHT)
  - 32-entry link stack for subroutine return address prediction (with some of these entries allocated for speculation) per thread in ST and SMT2 modes. For SMT4 mode, there are 16 entries available per thread. In SMT8 mode, there are 8 entries available per thread.
  - 768-entry count cache for address prediction shared by all the active threads running on the core. Five-hundred twelve (512) of the count cache entries are accessed based on the effective address of the **bcctr** instruction XORed with the folded GHV with two confidence bits used to provide hysteresis for replacement. The other 256 entries are accessed with just the effective address of the **bcctr** with one confidence bit.
  - No instruction fetch bubble to fetch from sequential path of a predicted not-taken branch.
  - In ST mode, there are two cycles of instruction fetch bubbles to fetch from the target address of a predicted taken branch.
  - In SMT2 and SMT4 mode, thread priority is factored in to determine which thread to fetch from (to improve overall fetch throughput).
  - Track up to 24 outstanding taken branches per thread in ST and SMT2 mode, 12 outstanding taken branches in SMT4 mode, and six outstanding taken branches per thread in SMT8 mode. The number of predicted not-taken branches tracked can be higher.

2.3.2 Instruction Decode and Preprocessing

- 3-cycle pipeline to decode and preprocess instructions
  - Dedicated data flow for cracking one instruction into two or four internal operations. All instructions that crack into two internal operations and a subset of instructions that crack into four internal operations use the dedicated data flow. The rest of the cracked instructions use the microcoded templates.
  - Microcoded templates for longer emulation sequences of internal operations.
  - All internal operations expand into an approximately 80-bit internal form to simplify subsequent processing and explicitly expose register dependencies for all register pools.
  - Dispatch groups with up to eight instructions are formulated in ST mode. Two groups of up to four instructions are formulated in all other modes.
  - All cracked instructions must be the first instruction in a group.
  - Cracked and microcoded instructions have access to three renamed eGPRs, one renamed eCR-field, and two eVRs (for cracking 128-bit DFP instructions). The eGPR, eCR and eVR are extensions to the architected facilities.
- Logically there is one instruction fetch buffer (IFB) per thread (sizes differ based on the ST, SMT2, or SMT4 mode). Each IFB entry has four instructions.
  - There are 16 entries in an IFB per thread in ST and SMT2 mode, eight entries per thread in SMT4, and four entries per thread in SMT8 mode.
  - Physically, the IFB is implemented as one register file, partitioned for ST, SMT2, and SMT4 modes.
- Up to eight instructions can be placed in the IFB in a cycle (ST or SMT mode).
- Up to eight instructions can be taken out from the IFB in a cycle (ST or SMT mode).
- Instructions taken out for group formation and decode are from one (ST mode) or two threads.

2.3.3 Instruction Dispatch, Sequencing, and Completion Control

- Three dispatch buffers that can hold up to three dispatch groups when GCT is full.
- Inter-instruction dependence generation for RAW and WAW dependences.
- 28-entry global completion table:
  - Each entry is assigned to a particular thread at instruction dispatch.
  - Entries can be allocated nonsequentially and intermixed among the threads in SMT modes.
  - Group-oriented tracking associates up to two 4-operation dispatch groups or one 8-operation dispatch group to a single GCT entry.
  - Tracks internal operations from dispatch to instruction completion for up to 224 operations.
  - Branch instruction can be placed in the middle of a group (predicted taken or not-taken). A second branch always ends the group.
  - Branch misprediction for a branch placed in the middle of a group causes a partial group flush. The LSU flushes cause a flush of the entire group, even though the load/store operation might be in the middle of the group (no partial flush for LSU).
  - Capable of restoring the machine state for any of the instructions in flight.
    - — Fast restoration for instructions on group boundaries (such as, branches).
    - — Slower restoration for instructions contained within a group (such as, load/store operations).

- Supports precise exceptions (including machine check exceptions).
- Register renaming resources. A given thread can use any entry in the renamed register files, which can be dynamically shared amongst threads.
  - GPR rename mapper:
    - 124 entry in ST mode (32 architected and 92 for rename)
    - 2 × 124 entry in SMT2 mode (32 architected and 92 rename, per thread)
    - 2 × 124 entry in SMT4 mode. One GPR register file supports half the threads, and the other GPR register file supports the other half. Each GPR register file has 64 architected, leaving 60 for rename.
    - 2 × 124 entry in SMT8 mode. One physical register file supports half of the threads and the other register file supports the other half. The number of architected GPRS is limited to 64 at a time in the GPRs. The other least recently used GPRs are located in the SAR.This leaves 60 for rename per thread set.
    - Four eGPRs used on demand per thread (used for microcoded instructions)
    - In SMT2, SMT4, and SMT8 modes, a total of 106 renames (nonarchitected) are available, across both thread sets
    - The SAR is 72 entries per thread (32 GPR + 4 eGPR, × 2 for TM checkpointing) and contains castouts from the register file.
  - FPR and VR rename mapper:
    - 144 entry in ST mode (64 architected and 80 rename).
    - 144 × 2 entry in SMT2 mode (64 architected and 80 rename, per thread).
    - 144 × 2 entry in SMT4 and SMT8 modes (limited to 64 architected and 80 rename, per thread set - castouts in the SAR).
    - In SMT2, SMT4, and SMT8 modes, a total of 106 renames (non-architected) are available, across both thread sets.
    - The SAR is 128 entries per thread (64 VSX, × 2 for TM checkpointing) and contains castouts from the register file.
  - 30-entry XER rename mapper plus 32-entry ARF for the current architected values (XER broken into four mappable fields (ov, ca/oc, fxcc, tgcc) and one non-mappable field per thread)
  - Non-mappable bits: dc, ds, string-count; other_bits special fields (value predict): so
  - 20-entry LR/CTR/TAR rename mapper plus 24-entry ARF for architected values (one LR and one CTR, and one TAR per thread).
  - 32-entry CR rename mapper plus 64-entry ARF for architected values (eight CR fields per thread).
  - 28-entry FPSCR rename mapper (each entry corresponds to a GCT entry, which can belong to at most one particular thread at a given time).
- No register renaming on any architected registers not mentioned previously.
- Instruction queuing resources:
  - Two 32-entry unified issue queues (UniQ) are used for fixed-point, floating-point, VMX, VSX, DFU, Crypto, and load/store instructions. The two queues are split per thread-set in SMT2, SMT4, and SMT8 mode.
  - In ST, each IOP is assigned to the opposite queue, half from the IOP before it.
  - One 15-entry issue queue for branch instructions.
  - One 8-entry issue queue for CR-logical instructions.

2.3.4 Fixed-Point Execution Pipelines

- Two symmetric fixed-point execution pipelines:
  - Both are capable of basic arithmetic, logical and shifting operations.
  - Both are capable of multiplies, divides, and SPR operations.
- Out-of-order issue with a bias towards oldest operations first.
- Symmetric forwarding between fixed-point and load/store execution pipelines.
- In ST mode, instructions from a given thread can be executed in either pipeline.
- In SMT2, SMT4, and SMT8 mode, instructions from thread set 0 execute in pipeline 0, and instructions from thread set 1 execute in pipeline 1.
- Threads are dynamically assigned to a given thread set.

2.3.5 Load and Store Execution Pipelines

- Two load/store and two load execution pipelines, with 3-cycle, load-to-use latency (2-cycle bubble) for fixed-point loads and 5-cycle load-to-use latency for VS and floating-point loads. See *Table 10-17 Instruction Latencies and Throughputs* on page 246 for additional information.
- Load/store units execute both load and store operations.
- Load units execute only load operations.
- Loads that update a FP, VSX, or VMX register execute in the LU. Loads that update an XER, execute in the LSU. Loads that update only a GPR can execute in either the LU or LSU.
- All four units can execute simple fixed-point operations:
  - In ST mode, a given load/store instruction can execute in either pipeline.
  - In SMT2, SMT4 and SMT8 modes, instructions from thread set 0 execute in pipeline 0, and instructions from thread set 1 execute in pipeline 1.
- Out-of-order issue with bias towards oldest operations first:
  - Fixed-point D-form stores are issued twice: an address-generation operation (issued to the LSU) and a data-steering operation (issued to the LU).
  - Fixed-point X-form stores are cracked into store_agen and store_data by the IFU (a cracked instruction starts a new group).
  - 64-bit floating-point and 64-bit VSX stores are issued twice: the store_agen is issued to the LSU and the store_data is issued to the VSU.
- 64 KB, 8-way set-associative, banked D-cache:
  - Supports four reads and one write in every cycle, when there is no bank conflict between write or read and a read. A given bank can support either two reads or one write in a given cycle.
  - 3-cycle load-use penalty for FXU loads (2-cycle bubble between a load and a dependent operation).
  - 5-cycle load-use penalty for FPU/VMX/VSX loads (4-cycle bubble between a load and a dependent operation).
  - Store-through (to L2 cache) policy; no allocate on store misses.
  - 128-byte cache line .
  - True LRU replacement policy.

  - Dedicated 64-byte reload interface from the L2 cache, which can supply 64 bytes in every processor clock.
  - Effective address index, real address tags (hardware fix-up on alias cases. That is, two different EAs that map to the same RA are not allowed to co-exist in the D-cache).
  - Parity protected; precise machine check interrupt on parity error (software fix-up).
- 48-entry, fully-associative primary and 144-entry secondary data effective-to-real address (D-ERAT) translation cache shared across the two thread sets:
  - Each entry translates either 4 KB, 64 KB, or 16 MB pages:
    — 16 GB pages take multiple 16 MB pages (used for server-mode only).
    — MSR[DR] = '0' is also created in the D-ERAT and shared by all threads.
  - Binary LRU replacement policy.
  - In SMT mode, each entry is tagged to indicate the valid threads.
  - In ST mode, 48-entry primary and 256-entry secondary D-ERAT are available
  - In SMT mode, there are 48-entry primary and 128-entry secondary D-ERAT available for each thread set. Entries are dynamically shared between the two threads (a given entry is not shared by multiple threads, unless it is MSR[DR] = '0').
- 32-entry, fully-associative segment lookaside buffer (SLB) per thread:
  - Each entry can support 256 MB or 1 TB segment sizes.
  - Multiple pages per segment (MPSS) feature is supported: 4 KB, 64 KB, and 16 MB pages (at most 2 pages) can be present concurrently in a given segment.
- 40-entry store re-order queue logically above the D-cache (real address based; CAM structure):
  - Sixty-four virtual entries (with no physical entity in SRQ) are available to allow a total of 64 outstanding stores to be dispatched per thread (for dispatch, virtual entry is sufficient).
  - A total of 40 outstanding stores can be issued (for issue, a real entry is required):
    — The SRQ is dynamically shared among the available threads.
    — The SRQ entry is allocated at the time of a store issue and deallocated when the store is written in the cache (after the completion point).
  - Store addresses and store data can be supplied on different cycles.
  - Stores wait in this queue until they are completed; then, they write the cache.
  - Supports store forwarding to inclusive subsequent loads (even if both are speculative). Store forwarding takes five additional cycles compared to a D-cache hit for a load.
  - For each SRQ entry, there is a store data queue (SDQ) entry of 16 bytes.
  - 16 bytes of store data can be sent to the L2 cache (and also to the D-cache, on a hit) in every processor cycle. Two distinct stores can be combined into one 16-byte store, under certain conditions).
- 44-entry load re-order queue (real address based; CAM structure):
  - Sixty-four virtual entries (with no physical entity in LRQ) are available to allow a total of 64 outstanding loads to be dispatched per thread in ST, SMT2, or SMT4 modes (for dispatch, virtual entry is sufficient).
  - A total of 44 outstanding loads can be issued. For instruction dispatch, virtual entry is sufficient; for issue, a real entry is required.
  - LRQ is dynamically shared among the available threads.

  - Keeps track of out-of-order loads and watches for hazards. For example:
    - Previous store to the same address that gets executed after the load; system executes a flush.
    - Previous load from the same address and a cross-invalidate has occurred; system executes a flush.
    - Atomic load-quad instruction and a cross invalidate has occurred; system executes a flush.
- 16-entry load miss queue (real address based):
  - Keeps track of loads that have missed in the L1 D-caches.
  - Dynamically shared among the threads in SMT2, SMT4, and SMT8 modes.
  - Prefetches from L1 cache are also tracked using the LMQ.
- Two 16-byte loads and one 16-byte store operation are supported for VMX and VSX operations per cycle. All architecturally-allowed alignments are supported in hardware.
- True little-endian (LE) mode is supported. All architecturally allowed alignments are supported in hardware.

2.3.6 Branch and Condition Register Execution Pipelines

- One branch execution pipeline:
  - Computes actual branch address and branch direction for comparison with prediction.
  - Redirects instruction fetching if either direction or target prediction was incorrect.
  - Assists in training and maintaining the branch history table predictors, the link stack, and the count cache.
- One Condition Register logical pipeline:
  - Executes CR logical instructions and the CR movement operations.
  - Also executes some **mfspr** instructions .
- Out-of-order issue with bias towards oldest operations first.

2.3.7 Unified Second-Level Memory Management (Address Translation)

- 2048-entry, 4-way set associative TLB:
  - 4 KB, 64 KB, 16 MB, and 16 GB pages are supported in TLB.
  - The TLB also supports "Virtualized Page Class Key Protection" with 32 keys.
  - Hardware-based reload (from the L2 cache interface, no L1 D-cache corruption).
  - Hardware-based update of the R bit, C bit, and TS bit.
  - Parity protected; precise machine check interrupt on parity error (software fix-up).
  - ITLB entries are shared by the eight threads as long as the entry belongs to the logical partition running on the core.
  - 12-bit LPAR ID per entry.
- Hit-under-miss is allowed in the TLB.
- Support for four concurrent table walks (without any restriction on thread of D-side or I-side requests).
- 32-entry fully-associative SLB, one per thread:
  - An SLB miss results in an interrupt (software reloads the SLB).
  - An SLB can also be loaded via the 32-bit PowerPC segment register instructions.
  - An SLB supports 256 MB and 1 TB segment sizes.
- A segment with a 4 KB base page size is allowed to have mixed pages of sizes 4 KB, 64 KB, and 16 MB pages.
- A segment with a 64 KB base page size is allowed to have mixed pages of sizes 64 KB and 16 MB pages.

- A read of an invalid SLB entry returns zeros for enhanced security.
- Supports 68-bit virtual address and 50-bit real address.
- Both software and hardware TLB management is allowed.
- True LRU replacement policy.

2.3.8 Data Prefetch

- Software initiated streams can use up to 16 entries. The 16 entries are shared in ST and SMT2 mode. In SMT4 mode there are two groups of eight (two threads share a group) and in SMT8 mode there are four groups of four entries (again two threads share a group) managed with effective addresses.
- Supports hardware- and software-initiated streams.
- Hardware-initiated streams are dynamically shared among the available theads.
- Sixteen independent data streams capable of striding up or down.
- Stride one cache line support.
- Stride N support
- The 4-entry table tracks 32-byte strides across the previous eight miss addresses [50:58] to detect 32-byte strides
- The stream is installed in the main 16-entry queue when a stride is detected.
- Prefetches and allocates up to two cache lines ahead of a load into the L1 D-cache.
- The ramp and depth of prefetch is controled using the DSCR Register.
- Support for software-initiated stream startup (special variant of the **dcbt** instruction).

2.3.9 VSU Execution Pipeline

- The VSU unit contains a binary floating-point execution unit, SIMD double-precision floating-point (VSX) execution unit, decimal floating-point unit (DFU), Crypto unit, and the VMX execution unit.
- Up to two instructions can be issued to the VSU in a given cycle to the two pipelines.
  - The pipes are fully symmetrical except for the Crypto and DFU engines which are shared.
  - In ST mode, instructions can be issued to both pipes.
  - In SMT2, SMT4, and SMT8 mode, pipe0 executes instructions from thread set 0 and pipe 1 executes instructions from thread set 1.
  - Out-of-order issue with a bias towards oldest operations first.
  - Two load result bus to the VRF; each supports up to 16-byte loads in a cycle.
  - Store data bus from VRF to the SDQ supports two 16-byte stores in a cycle.
- Floating-point execution.
  - Two symmetric floating-point execution pipelines, with 6-stage execution.
    - Both are capable of the full set of floating-point instructions.
    - All data formats supported in hardware (no floating-point assist interrupts).
    - Back-to-back 6-cycle issue to both local and remote FPU (symmetric forwarding between the floating-point pipelines, with no additional cycle of latency).
- VSX execution.
  - Two symmetric SIMD floating-point execution pipelines, with 6-stage execution.
    - Both are capable of the full set of VSX instructions (single-precision and double-precision).
    - All data formats supported in hardware (no assist interrupts).

  - — A test instruction facilitates execution of multiple concurrent divide or square-root operations.
  - — Back-to-back 6-cycle issue to both local and remote VSX pipe.

- VMX execution.
  - Four execution pipelines within VMX: simple fixed-point, complex fixed-point, permute, and 4-way SIMD single-precision floating-point unit.
    - — Simple fixed-point operations take two execution cycles.
    - — Complex fixed-point operations take seven execution cycles.
    - — Permute operations take two execution cycles.
    - — Vector floating-point operations take six execution cycles.
- Crypto execution.
  - The Crypto unit executes symmetric AES instructions that include polynomial multiply to support the Galios Counter Mode (GCM).
  - Allows out-of-order issue with a bias towards the oldest instruction.
  - Crypto operations can be issued from either issue queue.
  - Pipelined execution allowed.
  - Crypto operations take six execution cycles.
- See *Table 10-17 Instruction Latencies and Throughputs* on page 246 for additional information.

2.3.10 DFP Execution Pipeline

- The DFP unit can execute 64-bit or 128-bit decimal floating-point operations.
- Allows out-of-order issue with a bias towards the oldest instruction.
- Pipelined execution allowed.
- DFP operations can be issued from either issue queue.
- The 128-bit DFP instructions can be cracked into 2-way or 4-way internal operation.

3. Power Architecture Compliance

The following sections are intended to be read with their respective companion documents. Throughout these sections, it is assumed that the reader is familiar with the following architecture documents:

- *Power ISA User Instruction Set Architecture - Book I (version 2.07)*
- *Power ISA Virtual Instruction Set Architecture - Book II (version 2.07)*
- *Power ISA Operating Environment Architecture (Server Environment) - Book III-S (version 2.07)*

3.1 Book I - User Instruction Set Architecture

This section of the document identifies architectural implications of the POWER8 design point as they relate to the User Instruction Set Architecture. This is accomplished by walking through each of the relevant sections of Book I and highlighting the POWER8 solution to the architectural flexibility provided by the Power ISA.

In many cases, the architecture defines certain scenarios as *invalid forms*. In these cases, although this document provides information on how the POWER8 processor handles these scenarios, it is strongly recommended that software avoid building any type of reliance on these behaviors because they are likely to be different in future generation machines. This information is primarily provided as an aid to the design verification and debug efforts

3.1.1 Defined Instructions

The POWER8 processor core implements all Book I instructions in the categories listed as being required for the server platform in *Appendix B. Platform Support Requirements* of the *Power ISA (Version 2.07)*.

3.1.1.1 Illegal Instructions

An attempt to execute an illegal instruction as defined in the *Appendix D. Illegal Instructions* of the *Power ISA (Version 2.07)* results in a hypervisor emulation assistance interrupt.

3.1.1.2 Instructions Supported

The POWER8 processor supports all of the instructions described in Book I: Power ISA User Instruction Set Architecture of the *Power ISA*, except those instructions with the designation of *Embedded*. Furthermore, it supports the Service Processor "Attention" described in *Appendix E. Reserved Instructions* of the *Power ISA (Version 2.07)*. This instruction is conditionally enabled by HID0[31] = '1'. When enabled, this instruction is a user-level instruction.

3.1.1.3 Invalid Forms

In general, the POWER8 processor handles *invalid forms* of instructions in the manner that is most convenient for the particular case (within the scope of meeting the boundedly-undefined definition described in the Power ISA). This document specifies the cases where a system-level error handler is invoked, but does not always describe actions for other cases of invalid forms. It is not recommended that software or other system facilities make use of the POWER8 behavior in these cases, because it is not formally specified and might be different in another processor that implements the Power ISA.

The POWER8 processor ignores the state of reserved bits in the instructions (denoted by "///" in the instruction definition) and executes the instruction normally. Software should set these bits to '0' per the Power ISA.

3.1.2 Branch Processor

3.1.2.1 Instruction Fetching

In an effort to increase performance, the POWER8 processor does instruction prefetching before it determines whether or not particular instructions will actually execute. This prefetching follows all of the architectural constraints relative to cache inhibited and guarded regions of storage. A set of software-accessible mode bits is implemented to allow control over the various types of prefetch supported (for more information, see *Section 3.8 HID Registers (HID0, HID1, HID4, and HID5)* on page 102.

3.1.2.2 Branch Prediction

The POWER8 processor core uses several dynamic branch prediction mechanisms to improve performance. A set of three branch history tables (local, global, and selector) is used to predict the direction of branch instructions early in the pipeline. To improve the efficiency of these predictors, the POWER8 core uses the architected BO field hint bits associated with many of the branch instructions (the "a" and "t" bits).

In addition, for **bclr** instructions, a link stack (or call-return stack) is used to predict the target address of the branch. Similarly, for **bcctr** instructions, local and global count caches are used to predict the target address for this type of branch. To improve the efficiency of these address predictors, the POWER8 core uses the architected BH-field hints associated with several of the branch instructions. These hints are used by the hardware to improve the accuracy of the link stack and the count cache.

As the branch instructions progress through the pipeline, eventually they become fully executed. At that point, the hardware determines whether the predicted target address and/or the direction of the branch matches the actual outcome of the branch. If the prediction was incorrect, the hardware takes the appropriate actions with respect to flushing undesired instructions and results, redirecting the pipeline, and updating the branch prediction information in the branch history tables.

Although the overall performance of the machine is strongly dependent on these branch prediction mechanisms, a set of firmware-accessible mode bits is available to disable these features via scan initialization.

3.1.2.3 Instruction Cache Block Touch Hint

The POWER8 processor supports the instruction cache block touch instruction. Instead of bringing the data into the level 1 (L1) cache, it prefetches the data into the level 2 (L2) cache, which works just like a data cache block touch for store (dcbtst) hint instruction.

3.1.2.4 Out-of-Order Execution and Instruction Flushes

The POWER8 processor uses out-of-order instruction execution. Instructions can be speculative on a predicted branch direction, or simply speculative beyond an instruction that might cause an interrupt condition. In the event of a misprediction or an interrupt, instructions from the mispredicted path and the results produced by those instructions are discarded, presenting the effect of sequentially executed instructions down the appropriate branch paths and precise exceptions as required.

3.1.2.5 Branch Processor Instructions with Undefined Results

The results of executing an invalid form of a branch instruction or an instance of a branch instruction for which the architecture specifies that some results are undefined are described as follows. Only results that differ from those specified by the architecture are described in the following list.

- Instructions with reserved fields
  Bits in reserved fields including the z-bits in the BO field are ignored; the results of executing an instruction in which one or more of these bits are '1' is the same as if the bits were '0'.
- **bcctr** and **bcctrl** instructions
  If BO[2] = '0', the contents of CTR, before any update, are used as the target address and for the test of the contents of CTR to resolve the branch. The contents of the CTR are then decremented and written back to the CTR.
- System call instructions (opcode 17)

Bits 30:31	Description
'00'	**sc** instruction
'01'	illegal instruction exception
'10'	**sc** instruction
'11'	**sc** instruction

3.1.3 Fixed-Point Processor

3.1.3.1 Fixed-Point Exception Register (XER)

The Power ISA defines XER[0:31] and XER[35:56] as reserved. A **mfxer** returns the value as shown in *Table 3-1*

In the POWER8 core, the XER is implemented in several parts:

- XER renamed fields F0:F3 (see *Table 3-1*) are stored in an architected register file (ARF). The ARF consists of latches that store the F0:F3 fields for each of eight threads. The ARF contains eight (one per thread) Transactional Memory (TM) copies of the F0:F3 fields.
- XER nonrenamed bits (for example, F3NR) other than SO are stored in latches. Access is scoreboard managed. One copy for each of eight threads and eight TM copies are shared by FX0 and FX1.
- SO architected bits stored in latches with accompanying state machine. One copy for each of the eight threads and eight TM copies are shared by FX0 and FX1.

Table 3-1. XER Bits and Fields (Sheet 1 of 2)

XER Bits	Name	Field	Read/Write Behavior
0:31	Reserved	Unimplemented	Returns zeros on **mfxer.**
32	SO	F3NR	Set to '1' whenever OV = '1', except when **mtxer** sets SO = '0' and OV = '1'. This bit can be set to '0' or '1' by **mtxer**. A **mfxer** instruction reads the bit contents.
33	OV	F0	Set to '0' or '1' by various fixed-point instructions with OE = '1' or by **mtxer**. A **mfxer** instruction reads the bit contents.

Table 3-1. XER Bits and Fields (Sheet 2 of 2)

XER Bits	Name	Field	Read/Write Behavior
34	CA	F1	Set to '0' or '1' by add-carrying, subtract-from carrying and shift-right algebraic type instructions, and by **mtxer**. A **mfxer** instruction reads the bit contents.
35:43	Reserved		Returns zeros on **mfxer**
44:56	Reserved	F3NR	Written by **mtxer. mfxer** reads the bit contents.
57:63	String length	F3NR	String length field used **lswx** and **stswx**. Written by **mtxer**. A **mfxer** instruction reads the bit contents.

3.1.4 Storage Access Alignment Support Overview

Most storage accesses are performed without software intervention (such as, an alignment interrupt). The relative performance of these accesses depends to some degree on their alignment. In many cases, unaligned storage accesses are handled with performance equivalent to aligned accesses. However, in some cases POWER8 is forced to break unaligned accesses into multiple internal operations. Further, because effective address alignment for storage references cannot be determined until execution time, and POWER8 has dataflow-oriented execution pipelines that do not support iteration, some unaligned storage accesses actually cause a pipeline flush to allow a microcoded emulation of the instruction.

3.1.4.1 Misaligned Flushes

The LSU initiates a misaligned flush for the following conditions (See *Table 3-2* on page 44):

- Load crossing a 128-byte cache-line boundary and one of the cache-line misses.
- Load crossing a 32-byte sector boundary with either sector having an L1 D-cache miss and a 32-byte reload occurs instead of a 64-byte reload.
- Load/store crossing a 4 KB small page boundary.

Additionally, a misaligned flush is initiated in Data Address Watchpoint Register (DAWR) mode when the following conditions occur (see *Table 3-3* on page 52):

- Load crossing doubleword boundary when DAWR[63] = '1'
  - The following instructions are never considered to be crossing a doubleword boundary: **lq, lqarx, lfdp, lfdpx, lvebx, lvehx, lvewx, lvsl, lvsr, lvx, lvxl**
- Store crossing doubleword boundary when DAWR[62] = '1'
  - The following instructions are never considered to be crossing a doubleword boundary: **stq, stqcx., stfdp, stfdpx, stvebx, stvehx, stvewx, stvx, stvxl**

Note: If both a misaligned flush condition and an alignment interrupt condition are present, the alignment interrupt has precedence.

3.1.4.2 Alignment Interrupts

The LSU reports an alignment interrupt for the following conditions:

- Instruction not on a natural alignment boundary
  - Halfword boundary:
    - **lharx**
    - **sthcx.**
  - Word boundary:
    - **lmw**, **lwarx**,
    - **stmw**, **stwcx**,
  - Doubleword boundary:
    - **ldarx**, **stdcx**
  - Quadword boundary:
    - **lfdp**, **lfdpx**, **lq**, **stfdp**, **stfdpx**, **stq**, **stqcx**, **lqarx**
- Little-endian mode
  - **lmw**, **lswi**, **lswx**
  - **stmw**, **stswi**, **stswx**
- Caching-inhibited storage
  - **lfdp**, **lfdpx**, **lmw**, **lswi**, **lswx**
  - **dcbz**, **stfdp**, **stfdpx**, **stmw**, **stswi**, **stswx**
  - Any load or store not on natural alignment boundary:
    - Halfword boundary:
      - **lha**, **lhau**, **lhaux**, **lhax**, **lhbrx**, **lhz**, **lhzcix**, **lhzu**, **lhzux**, **lhzx**
      - **sthbrx, sth**, **sthcix**, **sthu**, **sthux**, **sthx**
    - Word boundary:
      - **lfiwax**, **lfiwzx**, **lfs**, **lfsu**, **lfsux**, **lfsx**, **lwa**, **lwaux**, **lwax**, **lwbrx**, **lwz**, **lwzcix**, **lwzu**, **lwzux**, **lwzx**, **lxswux**, **lxswx**
      - **stfiwx**, **stfs**, **stfsu**, **stfsux**, **stfsx**, **stwbrx**, **stw**, **stwcix**, **stwu**, **stwux**, **stwx**, **stxswux**, **stx-swx**
    - Doubleword boundary:
      - **ld**, **ldbrx**, **ldcix**, **ldu**, **ldux**, **ldx**, **lfd**, **lfdu**, **lfdux**, **lfdx**, **lxsdux**, **lxsdx**
      - **std**, **stdbrx**, **stdcix**, **stdu**, **stdux**, **stdx**,**stfd**, **stfdu**, **stfdux**, **stfdx**, **stxsdux**, **stxsdx**
    - Quadword boundary:
      - **lxvdsx**, **lxvd2ux**, **lxvd2x**, **lxvw4ux**, **lxvw4x**, **stxvd2x**, **stxvw4x**

    Note: A data storage interrupt (DSI) is reported on the following instructions to caching-inhibited storage regardless of the address alignment: **lbarx, lharx, lwarx, ldarx, lqarx, stbcx., sthcx., stwcx., stdcx., stqcx., lq, stq**

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 1 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
All loads/stores Caching Inhibited, not-naturally aligned		DSI or Alignment (see above for specific instructions)	Yes	No	Yes	No			
stxvw4x, stxvd2x	QW		No	No	No	No	N/A	EA	N/A
	DW		No	4K crossing	No	4K crossing	N/A	EA	EA + 16 ucode
	Even word (same as DW)		No	4K crossing	No	4K crossing	N/A	EA	EA + 16 ucode
	Odd word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	EA + 16 ucode
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	EA + 16 ucode
stxswx	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	N/A
stxsdx	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	EA + 4
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	EA + 4

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 2 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stvebx, stvehx, stvewx	QW		No	No	No	No	N/A	EA as defined by ISA	N/A
	DW		No	No	No	No	N/A	EA as defined by ISA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA as defined by ISA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
stvx(l)	QW		No	No	No	No	N/A	EA as defined by ISA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
stq	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
stfdp(x)	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	Always	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 3 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stfd	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stfs, stfliawx	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stxsspx, stxsiwx	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stswi, stswx	QW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	DW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Odd word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Non-word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 4 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stmw	QW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	DW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Odd word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A
lxvd2x, lxvw4x	QW		No	No	No	No	N/A	EA	N/A
	DW		No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Even word (same as DW)		No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Odd word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
lxswx	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA	E/A first byte in second page
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	E/A first byte in second page

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 5 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lxvdsx lxsdx	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 4
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 4
lvebx, lvehx, lvewx	QW		No	No	No	No	N/A	EA as defined by ISA	N/A
	DW		No	No	No	No	N/A	EA as defined by ISA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA as defined by ISA	N/A
	Odd word		No	No	No	No	N/A	EA as defined by ISA	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
lvx(l)	QW		No	No	No	No	N/A	EA as defined by ISA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by ISA	N/A

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 6 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lq	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lfdp(x)	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	Always	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A
dcbz	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	No	No	No	N/A	EA	N/A
	Odd word	Always	No	No	No	No	N/A	EA	N/A
	Non-word	Always	No	No	No	No	N/A	EA	N/A
lqarx	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
ldarx	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 7 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lwarx	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	No	No	No	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lharx	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	No	No	No	No	N/A	EA	N/A
	halfword	DSI	No	No	No	No	N/A	EA	N/A
	non half-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lxsspx, lxsiwax, lxsiwzx	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	N/A
lswi, lswx	QW	Always	No	Yes	Yes	No	N/A	EA	N/A
	DW	Always	No	Yes	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	N/A
	Odd word	Always	No	Yes	Yes	No	N/A	EA	N/A
	Non-word	Always	No	Yes	Yes	No	N/A	EA	N/A

Table 3-2. Operand Alignment Effects on Performance (Non-Watchpoint Mode) (Sheet 8 of 8)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt?	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW, see other tab	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lmw	QW	Always	No	Yes	Yes	No	N/A	EA	N/A
	DW	Always	No	Yes	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	N/A
	Odd word	Always	No	Yes	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 1 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
All loads/stores Caching Inhibited not-naturally aligned		Always	Yes	No	Yes	No			
All DW operations or less (word, hw)	crossing a DW boundary	Always	No	yes	No	yes			
All QW operations	crossing a QW boundary	DSI or Alignment (see above for specific instructions)	No	yes	No	yes			
stxvw4x, stxvd2x									
	QW		No	No	No	Yes	DW	EA	N/A
	DW		No	Yes	No	Yes	N/A	EA	EA + 16 ucode
	Even word (same as DW)		No	Yes	No	Yes	N/A	EA	EA + 16 ucode
	Odd word	Not naturally aligned	No	Yes	No	Yes	N/A	EA	EA + 16 ucode
	Non-word	Not naturally aligned	No	Yes	No	Yes	N/A	EA	EA + 16 ucode
stxswx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 2 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stxsdx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stvebx, stvehx, stvewx							DW		
	QW		No	No	No	No	N/A	EA as defined by arch	N/A
	DW		No	No	No	No	N/A	EA as defined by arch	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA as defined by arch	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
stvx(l)							QW		
	QW		No	No	No	No	N/A	EA as defined by arch	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 3 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stq							QW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
stfdp(x)							QW		
	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	Always	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A
stfd							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stfs, stfliawx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 4 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
stxsspx, stxsiwx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing	No	4K crossing	N/A	EA	E/A first byte in second page
stswi, stswx							DW		
	QW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	DW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Odd word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Non-word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
stmw							DW		
	QW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	DW	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Odd word	Always	No	Yes	Yes	No	N/A	EA	E/A first byte in second page
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 5 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lxvd2x, lxvw4x							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Even word (same as DW)		No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Odd word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 16
lxswx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA	E/A first byte in second page
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	E/A first byte in second page

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 6 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lxvdsx, lxsdx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA	N/A
	Odd word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA + 4
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	EA of first byte in second dw
lvebx, lvehx, lvewx							DW		
	QW		No	No	No	No	N/A	EA as defined by arch	N/A
	DW		No	No	No	No	N/A	EA as defined by arch	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA as defined by arch	N/A
	Odd word		No	No	No	No	N/A	EA as defined by arch	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 7 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lvx(l)							QW		
	QW		No	No	No	No	N/A	EA as defined by arch	N/A
	DW	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Even word (same as DW)	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Odd word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
	Non-word	Not naturally aligned	No	No	No	No	N/A	EA as defined by arch	N/A
lq							QW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lfdp(x)							DW		
	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	Always	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	No	Yes	No	N/A	EA	N/A
dcbz							DW		
	QW	Always	No	No	No	No	N/A	EA	N/A
	DW	Always	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	No	No	No	N/A	EA	N/A
	Odd word	Always	No	No	No	No	N/A	EA	N/A
	Non-word	Always	No	No	No	No	N/A	EA	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 8 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lqarx							QW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Even word (same as DW)	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
ldarx							DW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	Yes	No	Yes	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lwarx							DW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	No	No	No	No	N/A	EA	N/A
	Non-word	DSI	Yes	No	Yes	No	N/A	EA	N/A
lharx							DW		
	QW	DSI	No	No	No	No	N/A	EA	N/A
	DW	DSI	No	No	No	No	N/A	EA	N/A
	Even word (same as DW)	DSI	No	No	No	No	N/A	EA	N/A
	Odd word	DSI	No	No	No	No	N/A	EA	N/A
	halfword	DSI	No	No	No	No	N/A	EA	N/A
	non half-word	DSI	Yes	No	Yes	No	N/A	EA	N/A

Table 3-3. Operand Alignment Effects on Performance (Watchpoint Mode) (Sheet 9 of 9)

Power ISA Instructions	EA Alignment	Caching Inhibited Alignment Interrupt	BE: Takes Alignment Interrupt?	BE: Handled by Microcode?	LE: Takes Alignment Interrupt?	LE: Handled by Microcode?	DAWR Match on QW or DW	DAR Value if DSI (non-DAWR) or Alignment Interrupt on First Page	DAR Value if DSI (non-DAWR) or Alignment Interrupt on Second Page
lxsspx, lxsiwax, lxsiwzx							DW		
	QW		No	No	No	No	N/A	EA	N/A
	DW		No	No	No	No	N/A	EA	N/A
	Even word (same as DW)		No	No	No	No	N/A	EA	N/A
	Odd word		No	No	No	No	N/A	EA	N/A
	Non-word	Not naturally aligned	No	4K crossing OR 32-byte crossing L1 miss	No	4K crossing OR 32-byte crossing L1 miss	N/A	EA	E/A first byte in second page
lswi, lswx							DW		
	QW	Always	No	Yes	Yes	No	N/A	EA	N/A
	DW	Always	No	Yes	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	N/A
	Odd word	Always	No	Yes	Yes	No	N/A	EA	N/A
	Non-word	Always	No	Yes	Yes	No	N/A	EA	N/A
lmw							DW		
	QW	Always	No	Yes	Yes	No	N/A	EA	N/A
	DW	Always	No	Yes	Yes	No	N/A	EA	N/A
	Even word (same as DW)	Always	No	Yes	Yes	No	N/A	EA	N/A
	Odd word	Always	No	Yes	Yes	No	N/A	EA	N/A
	Non-word	Always	Yes	Yes	Yes	No	N/A	EA	N/A

3.1.4.3 Fixed-Point Load Instructions

Most forms of unaligned load operations are executed entirely in hardware. If a basic load operation crosses a page boundary, and either page translation signals an exception condition, then when the interrupt occurs, it appears as though none of the load instruction has executed. This is not always the case for load multiple or load string instructions.

The "Load Algebraic", "Load with Byte Reversal," and "Load with Update" instructions might have greater latency than other load instructions. These instructions are implemented as a sequence of internal operations. Due to the dynamic scheduling and out-of-order execution capability of the processor, these effects are somewhat minimized. Also note that although these instructions are broken up in this manner, the effects are never visible from a programming model perspective.

With the exception of the "Load and Reserve" and "Load Quadword" instructions, any load from storage marked *caching inhibited* that is not aligned causes an alignment interrupt. The "Load and Reserve" and "Load Quadword" instructions instead cause a data storage interrupt when they attempt to access a caching inhibited page regardless of their alignment.

An attempt to execute a non-quadword-aligned **lq** or **lqarx** instruction to a *cacheable* address causes an alignment interrupt.

See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

3.1.4.4 Fixed-Point Store Instructions

Most forms of unaligned store operations are executed entirely in hardware. If a store operation crosses a page boundary and the second page translation signals an exception condition, then after the interrupt is taken, it appears as though none of the storage updates have occurred to either page. This is not always the case for store multiple or store string instructions.

With the exception of the "Store Conditional" and "Store Quadword" instructions, any store to storage marked *caching inhibited* that is not aligned causes an alignment interrupt. The "Store Conditional" and "Store Quadword" instructions instead cause a data storage interrupt when they attempt to store to a *caching inhibited* page regardless of their alignment.

An attempt to execute a non-quadword-aligned **stq** or **stqcx.** instruction to a *cacheable* address causes an alignment interrupt.

See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

3.1.4.5 Fixed-Point Load and Store Multiple Instructions

The **lmw** instruction is executed in a manner such that up to two registers are loaded each cycle. Similarly, the **stmw** instruction is executed in a manner such that up to two registers are stored each cycle. The 40-entry store queue can accept up to two 8-byte stores per cycle; the cache can accept one 8-byte store per cycle. Because these instructions are emulated through the use of microcoded templates, after a small start-up penalty, they are processed at a rate of up to two registers per cycle.

Most forms of **lmw** and **stmw** instructions, even those that cross page and segment boundaries, are executed entirely in hardware. These instructions and the individual storage accesses associated with the instructions are not atomic. If a **stmw** crosses a page boundary, and the second page translation signals an exception condition, then after the interrupt is taken, it appears as though none, some, or all of the accesses

to the first page have occurred, and none of the accesses to the second page have occurred. On the other hand, for the **lmw** instruction that cross a page boundary where the second page translation signals an exception condition, some of the target registers may not be updated.

An attempt to execute a non-word-aligned **lmw** or **stmw** causes an alignment interrupt.

An attempt to execute an **lmw** or **stmw** to storage marked cache inhibited causes an alignment interrupt.

See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

The architecture allows these instructions to be interrupted by certain types of asynchronous interrupts (external interrupts, decrementer interrupts, machine check interrupts, and system reset interrupts). In these cases, for the load multiple instructions, all of the registers that were to be updated will have an undefined value, and the instruction must be completely restarted to achieve the full effect (that is, no partial restart capability is supported). For the store multiple instructions, some of the storage locations referenced by the instruction might have been updated, but to guarantee full completion of the instruction, it must also be completely restarted.

3.1.4.6 Fixed-Point Move Assist Instructions

The Load String Word instructions are executed in a manner such that up to two registers are loaded each cycle. Similarly, store string word instructions are executed in a manner such that up to two registers are stored each cycle. The 40-entry store queue can accept up to two 8-byte stores per cycle; the cache itself can only accept one 8-byte store per cycle.

Because the immediate forms of these instructions are implemented using microcoded templates, they incur a small start-up penalty. The X-form of the instructions contains a dependency on bits in the fixed-point XER register. If the dependent move assist x-form instruction is dispatched too soon after the instruction updating the XER, it will be flushed and re-executed.

Most "load string" and "store string" instructions that cross page or segment boundaries are executed entirely in hardware. If a "store string" crosses a page boundary and the second page translation signals an exception condition, then after the interrupt is taken, it appears as though none, some, or all of the accesses to the first page completed, and none of the accesses to the second page have occurred. On the other hand, for "load string" instructions that cross a page boundary where the second page translation signals an exception condition, all of the target registers have an undefined value.

If the storage operand of an **lswi** is word aligned, the accesses are performed in an optimal manner. If the operands are so aligned, the accesses are performed in an optimal manner if the operand resides entirely within a 64-byte aligned block that is resident in the L1 D-cache or resides entirely within a 32-byte aligned block (which either hits or misses in the L1 D-cache). Although other "unaligned" string operations are supported in hardware, they can cause machine flushes and require long sequences of microcode. As a result, these types of "unaligned" string instructions might have significantly longer latencies. For additional information, see *Table 10-17 Instruction Latencies and Throughputs* on page 246.

An attempt to execute an **lswi**, **lswx**, **stswi**, or **stswx** instruction to storage marked cache inhibited causes an alignment interrupt.

See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

The architecture allows these instructions to be interrupted by certain types of asynchronous interrupts (external interrupts, decrementer interrupts, machine check interrupts, and system reset interrupts). In these cases, for the load string instructions, all of the registers that were to be updated have an undefined value.

The instruction must be completely restarted to achieve the full effect (for example, no partial restart capability is supported). For the store string instructions, some of the storage locations referenced by the instruction might have been updated.

The architecture describes some preferred forms for the use of load and store string instructions. These preferred forms have no effect on the performance of the instructions in the POWER8 processor.

The architecture states that instructions that have the OE bit set, or instructions that might set the CA bit, might execute more slowly than instructions that do not. In the POWER8 processor, the SO bit in the XER is not renamed because the setting of the SO bit is expected to be rare. For instructions that execute when the OE bit in the XER is set, it is initially assumed that no overflow will occur and that the SO bit does not need to be changed. If the instruction does cause an overflow and the SO bit was not set before the instruction executed (and therefore needs to be set), the machine flushes this instruction and those beyond this instruction, sets the nonrenamed SO bit, and then refetches and re-executes the instructions that follow. In general, if no overflow occurs or the SO bit has already been set, this strategy will not have an adverse effect on performance.

On the other hand, most instructions that set and use the CA bit do not have any particular performance considerations. This field of the XER is renamed, and many of the common dependence hazards are minimized. The CA bit and the OC bit are renamed as one field. As a result, instructions that use the CA bit are falsely dependent on instructions that set the OC bit, and visa versa.

3.1.4.7 Integer Select (ISEL)

The POWER8 processor implements the integer select instruction as defined in the Power ISA.

3.1.4.8 Fixed-Point Logical Instructions

The architecture defines the *preferred NOP* to be '**ori** 0,0,0'. In POWER8 processor, this NOP form is recognized by the hardware and allowed to complete without taking any execution resources. This makes the instruction valuable for padding other instructions to achieve better alignment or better separation

3.1.4.9 Move To/From Special Purpose Register (SPR) Instructions

The POWER8 processor supports problem state read access from the following optional special purpose registers (SPRs):

- PMC1 - Performance Monitor Counter 1
- PMC2 - Performance Monitor Counter 2
- PMC3 - Performance Monitor Counter 3
- PMC4 - Performance Monitor Counter 4
- PMC5 - Performance Monitor Counter 5
- PMC6 - Performance Monitor Counter 6
- MMCR0 - Monitor Mode Control Register 0
- MMCR1 - Monitor Mode Control Register 1
- SIAR - Sampled Instruction Address Register
- SDAR - Sampled Data Address Register

3.1.4.10 Move to Condition Register Fields Instruction

The architecture warns that updating a subset of the **CR** fields on an ***mtcrf*** instruction might have worse performance than updating all of the fields. In the POWER8 processor, both the ***mtcrf*** instruction and the ***mfcr*** instruction are emulated through the use of microcode templates. For best performance, software must use the single-field variants (**mtocrf** and **mfocrf**) of these instructions as described in the Power ISA.

3.1.4.11 Fixed-Point Invalid Forms and Undefined Conditions

The results of executing an invalid form of a fixed-point instruction or an instance of a fixed-point instruction for which the architecture specifies that some results are undefined are described below, for the cases in which executing an instruction does not cause an exception. The list below describes the results of executing invalid instruction forms on the POWER8 processor core.

- **Instruction with Reserved Fields**
  Bits in reserved fields are ignored; the results of executing an instruction in which one or more reserved bits are '1' is the same as if the bits were '0'.
- **Load with Update Instructions** (RA = 0)
  EA is placed into R0.
- **Load with Update Instructions** (RA = RT)
  EA is placed into RT. The storage operand addressed by EA is accessed, but the data returned by the load is discarded.
- **Load Quadword Instruction** (RT is odd and RTp = RA)
  The POWER8 processor always takes a hypervisor emulation assistance interrupt anytime RT is an odd register for **lq**.
- **Load Quadword And Reserve Indexed Instruction** (RTp is odd, RTp = RA, RTp = RB)
  The POWER8 processor always takes a hypervisor emulation assistance interrupt anytime RT = RA, RT = RB, RA = zero.
- **Load Multiple Instruction**s (RA in the range of registers to be loaded)
  If an exception (for example, data storage or external) causes the execution of the instruction to be interrupted, the instruction is restarted, RA has been altered by the previous partial execution of the instruction, and RA is less than or greater than '0', the new contents of RA are used to compute EA.
- **Load Multiple Instructions** (causing a misaligned access)
  For a Load Multiple Word instruction, if the storage operand specified by EA is not a multiple of 4, an alignment interrupt is taken. For a Load Multiple Doubleword instruction, if the storage operand specified by EA is not a multiple of eight, an alignment interrupt is taken.
- **Load String Instructions** (zero length string)
  RT is not altered.
- **Load String Instructions** (RA and/or RB in the range of registers to be loaded)
  If RA and/or RB is in the range of registers to be loaded, the results are as follows.
  Indexed Form: If RA = 0, let Rx be RB; otherwise let Rx be the register specified by the smaller of the two values in instruction fields RA and RB. If RT = Rx, no registers are loaded; otherwise, registers RT through RX - 1 are loaded as specified in the architecture (that is, only part of the storage operand is loaded).
  Immediate Form: If RA = 0, the instruction is executed as if it were a valid form. If RA = RT, no registers are loaded; otherwise registers RT through RA - 1 are loaded as if the instruction were a valid form but specifying a shorter operand length**.**

- **Store with Update Instructions** (RA = 0)
  EA is placed into R0.
- **Store Quadword Instruction** (RS is odd)
  The contents of RS are stored into the word of storage addressed by EA. If RS < 31, the contents of RS+1 are stored into the word in storage addressed by EA+4; otherwise, the contents of R0 are stored there.
- **subfic**, **subfc**, and **subfco Instructions and their Rc = 1 forms**
  If RA[0:15] = x'0000', XER[CA] reflects the carry-out of bit 16; otherwise, it reflects the carry-out of bit 40.
- **divw**, **divwo**, **divwu**, and **divwuo Instructions**
  RT[0:31] is set to x'00000000'.
- **mulhw** and **mulhwu Instructions**
  RT[0:31] contains the same result as RT[32:63].
- **Divide Instructions** (divide by zero)
  If the divisor is 0, RT is set to zero. If Rc = '1' also, CR0 is set to '0010'.
- **Trap Word Immediate** and **Trap Word Instructions** (TO = (0b11110 | 0b11100))
- **Move To/From Special Purpose Register Instructions**
  The read/write behavior describes the results of specifying an **spr** value that is not defined for the implementation.
- **Move From Time Base Instruction**
  The **mftb** instruction is treated as an alias for the **mfspr** instruction; the results are the same as for executing an **mfspr** instruction.
- **Move From Condition Register Instruction** (bit 11 = '1')
  One CR Field is copied into the associated bits of RT, and the remaining bits of RT are set to zeros.
- **Move From Condition Register Instruction** (bit 11 = '1'; multiple bits of FXM set to '1')
  CR(**n**) is updated where **n** is the CR field specified by the bit in FXM that is set and has the smallest index value. If no bit in FXM is set to '1', the results are the same as if FXM was set to '00000001'.
- **Move To Condition Register Fields Instruction** (bit 11 = '1'; multiple bits of FXM set to '1')
  CR(**n**) is updated where **n** is the CR field specified by the bit in FXM that is set and has the smallest index value. If no bit in FXM is set to '1', executing the instruction does not modify the CR.

3.2 Floating-Point Processor (FP, VMX, and VSX)

The POWER8 VSU contains four double-precision floating-point units. Each of these units is optimized for fully pipelined double-precision multiply-add functionality. In addition, each unit is capable of performing the floating-point divide and square root instructions. The complex integer instructions from the VMX architecture are also executed on the VSU datapath.

The POWER8 VSU implements the VSX architecture, specifying 2-way DP or 4-way SP operations. Dependent floating-point instructions have a minimum issue-to-issue interval of six cycles. The vector SP throughput has improved because it is possible to execute two 4-way SIMD SP instructions per cycle.

Legacy BFU and VMX architectures are also fully supported in the POWER8 VSU.

3.2.1 Vector Single-Precision Bandwidth

In the POWER8 core, the double-precision FPU supports simultaneous execution of two vector single-precision operations. This increases the single-precision bandwidth of the POWER8 core to 16 FLOPs per cycle.

In the POWER8 core, the convert and estimate instructions are executed in a fully pipelined manner, as well as the increased bandwidth of the multiply-add and move instructions. From the floating-point instructions, only the divide and square-root instructions cannot be started every cycle.

The compares, min/max, and test-for-software divide/square-root instructions are now executed on the vector integer (XS) pipeline to take advantage of the shorter latency. Also, the move-from-FPSCR and move-to-FPSCR instructions are separated from the floating-point datapath.

3.2.2 IEEE Compliance

The POWER8 implementation of binary floating-point (BFP), decimal floating-point (DFP) and vector-scalar floating-point (VSX) architecture complies with the IEEE P754-2008 standard.

In the POWER8 processor, the setting of the non-IEEE (FPSCR[NI]) bit has no effect on the instruction execution performance nor the results of the instruction.

3.2.3 Divide and Square-Root Latencies

All divide and square-root instructions can be executed independently on both pipes. The latencies of the instructions are listed in *Table 3-4*.

If the result of a double-precision divide instruction (**fdiv**, **xsdivdp**, **xvdivdp**) is tiny, an additional iteration is necessary to denormalize the result. This increases the instruction latency by six cycles. For all other instructions, the latency does not depend on the input operands. For additional information, see *Table 10-17 Instruction Latencies and Throughputs* on page 246.

Table 3-4. Latencies of Floating-Point Divide/Square-Root Instructions

Instruction	Issue-to-Finish	Dependent Operations (issue-to-issue)	Next Independent Multicycle Operation on Same Pipe (issue-to-issue)	Number of Free Floating-Point Slots until Next Multicycle Can Start
fdiv **xsdivdp** **xvdivdp**	35	32	26	18/26
fdivs **xsdivsp**	29	26	20	14/20
xvdivsp	31	28	22	12/22
fsqrt **xssqrtdp** **xvsqrtdp**	46	43	37	26/37
fsqrts **xssqrtsp**	34	31	25	19/25
xvsqrtsp	35	32	26	14/26

3.2.4 Early Forwarding Conditions

The back-to-back latency for all single-cycle floating-point instructions is six cycles. The 6-cycle result forwards the correct result only if both the result and the input operand are compatible floating-point values. For the following cases, the consumer instruction would misunderstand the data; instead it finishes with a flush request, triggering the ISU to eventually repeat the dependent instruction with operands from the register file:

- The forwarded result is the result of a convert-float-to-integer instruction.
- The forwarded result is used as an input operand for a convert-integer-to-float instruction.
- The forwarded result is a 32-bit floating-point value, but the input operand is a 64-bit floating-point value (for example, **xscvdpsp** to **fadd**).
- The forwarded result is a 64-bit floating-point value, but the input operand is a 32-bit floating-point value (for example, **xvadddp** to **xvcvspuxds**).

The previous scenarios are rare in real programs, because they require mixing of incompatible data types. Hence, the performance loss is acceptable.

Note: It might make sense to execute a convert-float-to-integer instruction followed by a dependent convert-integer-to-float instruction. Because this also causes a flush request if the issue-to-issue-interval is six cycles, this scenario must be avoided by the compiler. (If the purpose is to round a floating-point number to an integral value, one of the atomic round-to-integral-value instructions must be used.)

There are out-of-range scenarios where the result of the instruction is not defined by the ISA. Because exponent MSBs might be dropped while packing the result into the target format, the value on the 6-cycle result (before packing) can be inconsistent with the value written to the target register (after packing). To guarantee that program results are independent of the issue behavior, the 6-cycle result is invalidated if there is such an inconsistency risk. Instructions that consume a result below via the 6-cycle result bus will finish with a flush request:

- The result of **xscvdpsp** with FPSCR[UE] = 1 and $0 < |XB| < 2^{-255}$ (an underflow will occur).
- The result of **xscvdpsp** with FPSCR[OE] = 1 and $|XB| >= 2^{257}$ (an overflow will occur).

Instructions that consume the data below via the 6-cycle result might finish with a flush request. Whether or not it actually happens depends on the exponent value of the unnormalized result.

- The result of a single-precision arithmetic instruction with at least one out-of-range input operand, FPSCR[UE] = 1, and an underflow exception occurs.
- The result of a single-precision arithmetic instruction with at least one out-of-range input operand, FPSCR[OE] = 1, and an overflow exception occurs.

3.2.5 Floating-Point Exceptions

Precise floating-point exceptions are provided whenever either of the floating-point enabled exception mode bits (MSR[FE0], MSR[FE1]) are set. In all cases, the floating-point instructions are executed out-of-order (as required), and any resulting exceptions are sorted out at completion time. In some cases, due to the group-oriented instruction tracking scheme used, when an exception is detected, the hardware flushes the pipeline and re-dispatches the instructions individually to provide the precise exception. Because this only happens if an interrupt is to be taken, it does not represent a measurable slowdown in performance.

3.2.6 Floating-Point Load and Store Instructions

Most forms of unaligned floating-point storage accesses are executed entirely in hardware. In the POWER8 core, the load/store operations have been improved such that byte-aligned, 16-byte VSX load/store operations are executed efficiently in hardware. However, care must be taken that the unaligned VSX 16-byte storage accesses do not cross a 4K-page boundary, because doing so results in a micro-coded implementation with slower execution time. See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

3.2.7 Heterogeneous Precision Arithmetic

The following instructions are referred to as scalar single-precision arithmetic instructions:

- **fadds**[**.**], **xsaddsp**, **fsubs**[**.**], **xssubsp**, **fmuls**[**.**], **xsmulsp**
- **fmadds**[**.**], **xsmadd**[**am**]**sp**, **fmsubs**[**.**], **xsmsub**[**am**]**sp**
- **fnmadds**[.], **xsnmadd**[**am**]**sp**, **fnmsubs**[**.**], **xsnmsub**[**am**]**sp**
- **fsqrts**[**.**], **xssqrtsp**, **fdivs**[**.**], **xsdivsp**
- **fres**[**.**], **xsresp**, **frsqrtes**[**.**], **xsrsqrtesp**

3.2.7.1 NaN Propagation

If a single-precision arithmetic instruction propagates a NaN where any of the fraction bits [24:52] is nonzero, the resulting QNaN has all of the fraction bits [24:52] cleared to zero.

3.2.7.2 Square Root Overflow and Underflow

Due to the compacting nature of the square-root operation, the instructions **fsqrts**, **xssqrtsp**, **frsqrtes**, and **xsrsqrtesp** cannot underflow or overflow if their operands are representable in single-precision format. However, if the operand is not representable in single-precision format, an underflow or overflow can occur. This will be recorded in the FPSCR.

3.2.7.3 Hardware Behavior on Enabled Underflow and Enabled Overflow Exception

If FPSCR[UE] is enabled and an underflow occurs, the contents of the result register and FPSCR status are not defined for scalar SP instructions. The hardware takes the following actions:

1. Underflow exception is set, FPSCR[UX] = '1'.
2. The exponent of the normalized intermediate result is adjusted by adding 192.
3. The double-precision bias of 1023 is added to the exponent.
4. The biased exponent is reduced to 11 bits by ANDing with x'7FF'.
5. The adjusted rounded result is placed into the target FPR.
6. FPSCR[FPRF] is set to indicate Normalized Number.

If FPSCR[OE] is enabled and an overflow occurs, the contents of the result register and FPSCR status are not defined for scalar SP instructions. The hardware takes the following actions:

1. Overflow exception is set, FPSCR[OX] = '1'.

2. The exponent of the normalized intermediate result is adjusted by subtracting 192.
3. The double-precision bias of 1023 is added to the exponent.
4. The biased exponent is reduced to 11 bits by ANDing with x'7FF'.
5. The adjusted rounded result is placed into the target FPR.
6. FPSCR[FPRF] is set to indicate Normalized Number.

3.2.8 Handling of Denormal Single-Precision Values in Double-Precision Format

3.2.8.1 Producing Single-Precision Denorms

If a scalar single-precision instruction produces a denormal result, the FPU is not able to normalize the data after rounding, which is needed to compute the architected register file format. Instead the FPU sets the exponent bits to 897 = x'381', corresponding to an implied bit magnitude of 2^{-126}, and returns a denormal fraction. Additionally, a flag in the result register is set, which marks the 64-bit result single-precision denormal. This flag is called "dirty". Dirty results can be produced by any of the following instructions:

- All scalar SP arithmetic instructions as listed above.
- The round-to-single-precision instructions **frsp**[**.**] and **xsrsp.**
- The load single-precision instructions **lfs** and **lxsspx**.
- The floating-point select instruction **fsel**[**.**].

Note: If an enabled underflow exception occurs, the arithmetic instructions, as well as the rounding instructions, return a normalized fraction. Hence, if FPSCR[UE] is set, an arithmetic or rounding instruction never sets the 'dirty' flag.

The **fsel**[**.**] instruction only produces a dirty result if the selected operand was dirty. Execution of **fsel**[**.**] does not depend on FPSCR[UE].

A single-precision load instruction always sets the dirty flag if single-precision denormal data are loaded into the register file regardless of FPSCR[UE].

Any instruction not listed previously cannot produce dirty results.

3.2.8.2 Consuming Single-Precision Denorms

The majority of the BFU and VSX double-precision format operations, including all stores, can handle dirty input operands and perform the required normalizing on-the-fly. Thus, inputting dirty data to one of these instructions has no side effect.

To ensure that a register does not contain dirty data, it is recommended to move the content of the register to itself using the VSX DP copy sign instruction

xvcpsgndp xi, xi, xi

This instruction normalizes a potentially dirty input and writes it back to the register file in 64-bit double-precision format.

If an instruction that cannot handle dirty input operands sees a dirty input operand, it generates a denormal input exception that results in a soft patch interrupt to the hypervisor with HSRR1[43] = '1'. The hypervisor can use the contents of the HEIR to examine the instruction that caused the interrupt and perform the previously mentioned **xvcpsgndp** for each of the source registers for that instruction. Alternatively, the hypervisor can simply perform the previous **xvcpsgndp** for all 64 VSR Registers. In either case, after the dirty data has been reformatted by the hypervisor software, the instruction can then be re-executed to produce the result. The VSX Scalar SP instructions, introduced in the POWER8 processor, can also write SP denorms into VSR Registers 0 - 31.

Instructions without Support of Dirty Inputs

The following instructions do not support dirty inputs and generate denormal input exceptions whenever any of the operands is marked dirty.

- Any instruction operating on 32-bit operands
- Any convert from integer
- The move to FPSCR instruction **mtfsf**[**.**]
- Any compare, test-for-software, or min/max instructions
- Any extract instructions: **fmrg**[**oe**]**w**, **mfvsrd**, **mfvsrwz**
- Any DFU instructions
- Any VSX fixed-point arithmetic, logical, or permute instruction

If it is expected that a scalar single-precision workload produces denormal results quite often and also does single-precision compares frequently, it is recommended to precede the compare instructions with a normalizing instruction that does nothing, for example by using **fmr** or **xscpsgndp**. Alternatively, consider using scalar DP or vector SP code, where result precision and result format always match.

Table 3-5 on page 72 provides a complete list of VSU instructions that trigger an exception on consumption of a dirty operand.

3.2.9 Floating-Point Invalid Forms and Undefined Conditions

The results of executing an invalid form of a floating-point instruction or an instance of a floating-point instruction for which the architecture specifies that some results are undefined are described below for the cases when executing an instruction does not cause an exception.

- Scalar single-precision instructions with operands not representable in single-precision format
  See *Section 3.2.7 Heterogeneous Precision Arithmetic* on page 68.
- Instruction with reserved fields
  Bits in reserved fields are ignored. The results of executing an instruction in which one or more reserved bits are '1' is the same as if the bits were '0'.
- Load or Store Floating-Point with Update instructions (RA = 0)
  EA is placed into R0.
- Floating-Point Store Single instructions (exponent < 874 and FRS[9:31] <> 0)
  The value placed in storage is a 0 with the same sign as the value in the register.
- Scalar Floating-Point instructions
  $VSR_{64:127}$ is set to x'0000_0000_0000_0000'.

- Scalar Convert to Integer Word instructions (**xscvdpuxws**, **xscvdpsxws**, **fctiwuz**, **fctiwz**, **ctiwu**, **fctiw**)
  VSR[0:31] is set to VSR[32:63].
- VSX Scalar Convert From Double-Precision to Single-Precision (**xscvdpsp**, **xscvdpspn**)
  VSR[32:63] is set to VSR[0:31].
- Scalar Convert to Integer instructions
  FPSCR[FPRF] is set to '00000'.
- VSX Vector Convert From Double-Precision to Single-Precision (**xvcvdpsp**),
  VSX Vector Convert Double-Precision to Integer Word (**xvcvdpsxws**, **xvcvdpuxws**),
  VSX Vector Convert From Integer Doubleword to Single-Precision (**xvcvsxdsp**, **xvcvuxdsp**)

  VSR[32:63] is set to VSR[0:31].
  VSR[96:127] is set to VSR[64:95].
- Move from FPSCR instruction
  FRT[0:63] is set to FPSCR[0:63] with the first 29 bits set to zero.
- Scalar Reciprocal Estimate instructions:
  (**fre**, **fres**, **xsredp**, **xsresp**, **frsqrte**, **frsqrtes**, **xsrsqrtep**, **xsrsqrtesp**)
  FPSCR[FR] and FPSCR[FI] are set to '0' and FPSCR[XX] is unchanged, even if an overflow exception occurs.
- VSX Vector Floating-Point Reciprocal Estimate instructions: **xvredp**, **xvresp**, **xvrsqrtedp**, **xvrsqrtesp**
  FPSCR[XX] is unchanged, even if an overflow exception occurs.
- Disabled Overflow Exception (OX = '1', OE = '0')
  For divide and square root instructions, FPSCR[FR] is set to '1' if the result is rounded to $\pm\infty$, and set to '0' if the result is rounded to the largest representable number. For Scalar Reciprocal Estimate instructions, FPSCR[FR] is set to '0'. For all other instructions, FPSCR[FR] is set to '1' if a disabled overflow exception occurs.

3.3 Optional Facilities and Instructions

None.

3.4 Little Endian

The POWER8 core supports true little endian. Byte swapping is performed before data is written to the I-cache and before data is fetched into the execution units; that is, between the D-cache and the execution units (for example, GPR, FPR/VR/VSR).

The load and store multiple instructions and the move assist instructions are not supported in little-endian mode. Attempting to execute any of these instructions in little-endian mode causes the system alignment error handler to be invoked.

3.5 Instruction That Can Soft Patch

Table 3-5 contains the explicit list of instructions that trigger a normalization interrupt (soft patch) if any of the operands contains scalar single-precision denormal data, as described in *Section 3.2.8 Handling of Denormal Single-Precision Values in Double-Precision Format* on page 69.

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 1 of 11)

Mnemonic	Operands	Description
fcfid	FRT,FRB	Floating Convert From Integer Doubleword
fcfid.	FRT,FRB	Floating Convert From Integer Doubleword and Record
fcfids	FRT,FRB	Floating Convert From Integer Doubleword Single
fcfids.	FRT,FRB	Floating Convert From Integer Doubleword Single and Record
fcfidu	FRT,FRB	Floating Convert From Integer Doubleword Unsigned
fcfidu.	FRT,FRB	Floating Convert From Integer Doubleword Unsigned and Record
fcfidus	FRT,FRB	Floating Convert From Integer Doubleword Unsigned Single
fcfidus.	FRT,FRB	Floating Convert From Integer Doubleword Unsigned Single and Record
fcmpo	BF,FRA,FRB	Floating Compare Ordered
fcmpu	BF,FRA,FRB	Floating Compare Unordered
fmrgew	FRT,FRA,FRB	Floating Merge Even Word
fmrgow	FRT,FRA,FRB	Floating Merge Odd Word
ftdiv	BF,FRA,FRB	Floating Test for software Divide
ftsqrt	BF,FRB	Floating Test for software Square Root
mtfsf	FLM,FRB,L,W	Move To FPSCR Fields
mtfsf.	FLM,FRB,L,W	Move To FPSCR Fields and Record
dadd	FRT,FRA,FRB	DFP Add
dadd.	FRT,FRA,FRB	DFP Add and Record
daddq	FRTp,FRAp,FRBp	DFP Add Quad
daddq.	FRTp,FRAp,FRBp	DFP Add Quad and Record
dcffix	FRT,FRB	DFP Convert From Fixed
dcffix.	FRT,FRB	DFP Convert From Fixed and Record
dcffixq	FRTp,FRB	DFP Convert From Fixed Quad
dcffixq.	FRTp,FRB	DFP Convert From Fixed Quad and Record
dcmpo	BF,FRA,FRB	DFP Compare Ordered
dcmpoq	BF,FRAp,FRBp	DFP Compare Ordered Quad
dcmpu	BF,FRA,FRB	DFP Compare Unordered
dcmpuq	BF,FRAp,FRBp	DFP Compare Unordered Quad
dctdp	FRT,FRB	DFP Convert To 64
dctdp.	FRT,FRB	DFP Convert To 64 and Record
dctfix	FRT,FRB	DFP Convert To Fixed
dctfix.	FRT,FRB	DFP Convert To Fixed and Record
dctfixq	FRT,FRBp	DFP Convert To Fixed Quad
dctfixq.	FRT,FRBp	DFP Convert To Fixed Quad and Record
dctqpq	FRTp,FRB	DFP Convert To 128
dctqpq.	FRTp,FRB	DFP Convert To 128 and Record

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 2 of 11)

Mnemonic	Operands	Description
ddedpd	SP,FRT,FRB	DFP Decode DPD To BCD
ddedpd.	SP,FRT,FRB	DFP Decode DPD To BCD and Record
ddedpdq	SP,FRTp,FRBp	DFP Decode DPD To BCD Quad
ddedpdq.	SP,FRTp,FRBp	DFP Decode DPD To BCD Quad and Record
ddiv	FRT,FRA,FRB	DFP Divide
ddiv.	FRT,FRA,FRB	DFP Divide and Record
ddivq	FRTp,FRAp,FRBp	DFP Divide Quad
ddivq.	FRTp,FRAp,FRBp	DFP Divide Quad and Record
denbcd	S,FRT,FRB	DFP Encode BCD To DPD
denbcd.	S,FRT,FRB	DFP Encode BCD To DPD and Record
denbcdq	S,FRTp,FRBp	DFP Encode BCD To DPD Quad
denbcdq.	S,FRTp,FRBp	DFP Encode BCD To DPD Quad and Record
diex	FRT,FRA,FRB	DFP Insert Biased Exponent
diex.	FRT,FRA,FRB	DFP Insert Biased Exponent and Record
diexq	FRTp,FRA,FRBp	DFP Insert Biased Exponent Quad
diexq.	FRTp,FRA,FRBp	DFP Insert Biased Exponent Quad and Record
dmul	FRT,FRA,FRB	DFP Multiply
dmul.	FRT,FRA,FRB	DFP Multiply and Record
dmulq	FRTp,FRAp,FRBp	DFP Multiply Quad
dmulq.	FRTp,FRAp,FRBp	DFP Multiply Quad and Record
dqua	FRT,FRA,FRB,RMC	DFP Quantize
dqua.	FRT,FRA,FRB,RMC	DFP Quantize and Record
dquai	TE,FRT,FRB,RMC	DFP Quantize Immediate
dquai.	TE,FRT,FRB,RMC	DFP Quantize Immediate and Record
dquaiq	TE,FRTp,FRBp,RMC	DFP Quantize Immediate Quad
dquaiq.	TE,FRTp,FRBp,RMC	DFP Quantize Immediate Quad and Record
dquaq	FRTp,FRAp,FRBp,RMC	DFP Quantize Quad
dquaq.	FRTp,FRAp,FRBp,RMC	DFP Quantize Quad and Record
drdpq	FRTp,FRBp	DFP Round To 64
drdpq.	FRTp,FRBp	DFP Round To 64 and Record
drintn	R,FRT,FRB,RMC	DFP Round To FP Integer Without Inexact
drintn.	R,FRT,FRB,RMC	DFP Round To FP Integer Without Inexact and Record
drintnq	FRTp,FRBp,RMC	DFP Round To FP Integer Without Inexact Quad
drintnq.	FRTp,FRBp,RMC	DFP Round To FP Integer Without Inexact Quad and Record
drintx	R,FRT,FRB,RMC	DFP Round To FP Integer With Inexact
drintx.	R,FRT,FRB,RMC	DFP Round To FP Integer With Inexact and Record
drintxq	FRTp,FRBp,RMC	DFP Round To FP Integer With Inexact Quad

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 3 of 11)

Mnemonic	Operands	Description
drintxq.	FRTp,FRBp,RMC	DFP Round To FP Integer With Inexact Quad and Record
drrnd	FRT,FRA,FRB,RMC	DFP Reround
drrnd.	FRT,FRA,FRB,RMC	DFP Reround and Record
drrndq	FRTp,FRA,FRBp,RMC	DFP Reround Quad
drrndq.	FRTp,FRA,FRBp,RMC	DFP Reround Quad and Record
drsp	FRT,FRB	DFP Round To 32
drsp.	FRT,FRB	DFP Round To 32 and Record
dscli	FRT,FRA,SH	DFP Shift Coefficient Left Immediate
dscli.	FRT,FRA,SH	DFP Shift Coefficient Left Immediate and Record
dscliq	FRTp,FRAp,SH	DFP Shift Coefficient Left Immediate Quad
dscliq.	FRTp,FRAp,SH	DFP Shift Coefficient Left Immediate Quad and Record
dscri	FRT,FRA,SH	DFP Shift Coefficient Right Immediate
dscri.	FRT,FRA,SH	DFP Shift Coefficient Right Immediate and Record
dscriq	FRTp,FRAp,SH	DFP Shift Coefficient Right Immediate Quad
dscriq.	FRTp,FRAp,SH	DFP Shift Coefficient Right Immediate Quad and Record
dsub	FRT,FRA,FRB	DFP Subtract
dsub.	FRT,FRA,FRB	DFP Subtract and Record
dsubq	FRTp,FRAp,FRBp	DFP Subtract Quad
dsubq.	FRTp,FRAp,FRBp	DFP Subtract Quad and Record
dtstdc	BF,FRA,DCM	DFP Test Data Class
dtstdcq	BF,FRAp,DCM	DFP Test Data Class Quad
dtstdg	BF,FRA,DGM	DFP Test Data Group
dtstdgq	BF,FRAp,DGM	DFP Test Data Group Quad
dtstex	BF,FRA,FRB	DFP Test Exponent
dtstexq	BF,FRAp,FRBp	DFP Test Exponent Quad
dtstsf	BF,FRA,FRB	DFP Test Significance
dtstsfq	BF,FRAp,FRBp	DFP Test Significance Quad
dxex	FRT,FRB	DFP Extract Biased Exponent
dxex.	FRT,FRB	DFP Extract Biased Exponent and Record
dxexq	FRT,FRBp	DFP Extract Biased Exponent Quad
dxexq.	FRT,FRBp	DFP Extract Biased Exponent Quad and Record
mfvsrd	RA,XS	Move From VSR Doubleword
mfvsrwz	RA,XS	Move From VSR Word and Zero
mtvscr	VB	Move To VSCR
vaddcuw	VT,VA,VB	Vector Add and Write Carry-Out Unsigned Word
vaddfp	VT,VA,VB	Vector Add Single-Precision
vaddsbs	VT,VA,VB	Vector Add Signed Byte Saturate

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 4 of 11)

Mnemonic	Operands	Description
vaddshs	VT,VA,VB	Vector Add Signed Halfword Saturate
vaddsws	VT,VA,VB	Vector Add Signed Word Saturate
vaddubm	VT,VA,VB	Vector Add Unsigned Byte Modulo
vaddubs	VT,VA,VB	Vector Add Unsigned Byte Saturate
vadduhm	VT,VA,VB	Vector Add Unsigned Halfword Modulo
vadduhs	VT,VA,VB	Vector Add Unsigned Halfword Saturate
vadduwm	VT,VA,VB	Vector Add Unsigned Word Modulo
vadduws	VT,VA,VB	Vector Add Unsigned Word Saturate
vaddudm	VT,VA,VB	Vector Add Unsigned Doubleword Modulo
vand	VT,VA,VB	Vector Logical AND
vandc	VT,VA,VB	Vector Logical AND with Complement
vavgsb	VT,VA,VB	Vector Average Signed Byte
vavgsh	VT,VA,VB	Vector Average Signed Halfword
vavgsw	VT,VA,VB	Vector Average Signed Word
vavgub	VT,VA,VB	Vector Average Unsigned Byte
vavguh	VT,VA,VB	Vector Average Unsigned Halfword
vavguw	VT,VA,VB	Vector Average Unsigned Word
vbrd	VT,VB	Vector Byte Reverse Doubleword (microcode)
vbrdxor	VT,VA,VB	Vector Byte Reverse Doubleword Xor (microcode)
vbrw	VT,VB	Vector Byte Reverse Word (microcode)
vbrwxor	VT,VA,VB	Vector Byte Reverse Word Xor (microcode)
vctsxs	VT,VB,UIM	Vector Convert Single-Precision to Signed Fixed-Point Word Saturate
vctuxs	VT,VB,UIM	Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate
vcipher	VT,VA,VB	Vector AES Cipher
vcipherlast	VT,VA,VB	Vector AES Cipher Last
vclzb	VT,VB	Vector Count Leading Zero Byte
vclzh	VT,VB	Vector Count Leading Zero Halfword
vclzw	VT,VB	Vector Count Leading Zero Word
vcmpbfp	VT,VA,VB	Vector Compare Bounds Single-Precision
vcmpbfp.	VT,VA,VB	Vector Compare Bounds Single-Precision and Record
vcmpeqfp	VT,VA,VB	Vector Compare Equal To Single-Precision
vcmpeqfp.	VT,VA,VB	Vector Compare Equal To Single-Precision and Record
vcmpequb	VT,VA,VB	Vector Compare Equal To Unsigned Byte
vcmpequb.	VT,VA,VB	Vector Compare Equal To Unsigned Byte and Record
vcmpequh	VT,VA,VB	Vector Compare Equal To Unsigned Halfword
vcmpequh.	VT,VA,VB	Vector Compare Equal To Unsigned Halfword and Record
vcmpequw	VT,VA,VB	Vector Compare Equal To Unsigned Word

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 5 of 11)

Mnemonic	Operands	Description
vcmpequw.	VT,VA,VB	Vector Compare Equal To Unsigned Word and Record
vcmpequd	VT,VA,VB	Vector Compare Equal To Unsigned Doubleword
vcmpequd.	VT,VA,VB	Vector Compare Equal To Unsigned Doubleword and Record
vcmpgefp	VT,VA,VB	Vector Compare Greater Than or Equal To Single-Precision
vcmpgefp.	VT,VA,VB	Vector Compare Greater Than or Equal To Single-Precision and Record
vcmpgtfp	VT,VA,VB	Vector Compare Greater Than Single-Precision
vcmpgtfp.	VT,VA,VB	Vector Compare Greater Than Single-Precision and Record
vcmpgtsb	VT,VA,VB	Vector Compare Greater Than Signed Byte
vcmpgtsb.	VT,VA,VB	Vector Compare Greater Than Signed Byte and Record
vcmpgtsh	VT,VA,VB	Vector Compare Greater Than Signed Halfword
vcmpgtsh.	VT,VA,VB	Vector Compare Greater Than Signed Halfword and Record
vcmpgtsw	VT,VA,VB	Vector Compare Greater Than Signed Word
vcmpgtsw.	VT,VA,VB	Vector Compare Greater Than Signed Word and Record
vcmpgtsd	VT,VA,VB	Vector Compare Greater Than Signed Doubleword
vcmpgtsd.	VT,VA,VB	Vector Compare Greater Than Signed Doubleword and Record
vcmpgtub	VT,VA,VB	Vector Compare Greater Than Unsigned Byte
vcmpgtub.	VT,VA,VB	Vector Compare Greater Than Unsigned Byte and Record
vcmpgtuh	VT,VA,VB	Vector Compare Greater Than Unsigned Halfword
vcmpgtuh.	VT,VA,VB	Vector Compare Greater Than Unsigned Halfword and Record
vcmpgtuw	VT,VA,VB	Vector Compare Greater Than Unsigned Word
vcmpgtuw.	VT,VA,VB	Vector Compare Greater Than Unsigned Word and Record
vcmpgtud	VT,VA,VB	Vector Compare Greater Than Unsigned Doubleword
vcmpgtud.	VT,VA,VB	Vector Compare Greater Than Unsigned Doubleword and Record
vcfsx	VT,VB,UIM	Vector Convert Signed Fixed-Point Word to Single-Precision
vcfux	VT,VB,UIM	Vector Convert Unsigned Fixed-Point Word to Single-Precision
vgbbd	VT,VB	Vector Gather Bits by Bytes by Doubleword
vexptefp	VT,VB	Vector 2 Raised to the Exponent Estimate Single-Precision
vlogefp	VT,VB	Vector Log Base 2 Estimate Single-Precision
vmaddfp	VT,VA,VC,VB	Vector Multiply-Add Single-Precision
vmaxfp	VT,VA,VB	Vector Maximum Single-Precision
vmaxsb	VT,VA,VB	Vector Maximum Signed Byte
vmaxsh	VT,VA,VB	Vector Maximum Signed Halfword
vmaxsw	VT,VA,VB	Vector Maximum Signed Word
vmaxsd	VT,VA,VB	Vector Maximum Signed Doubleword
vmaxub	VT,VA,VB	Vector Maximum Unsigned Byte
vmaxuh	VT,VA,VB	Vector Maximum Unsigned Halfword
vmaxuw	VT,VA,VB	Vector Maximum Unsigned Word

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 6 of 11)

Mnemonic	Operands	Description
vmaxud	VT,VA,VB	Vector Maximum Unsigned Doubleword
vmhaddshs	VT,VA,VB,VC	Vector Multiply-High-Add Signed Halfword Saturate
vmhraddshs	VT,VA,VB,VC	Vector Multiply-High-Round-Add Signed Halfword Saturate
vminfp	VT,VA,VB	Vector Minimum Single-Precision
vminsb	VT,VA,VB	Vector Minimum Signed Byte
vminsh	VT,VA,VB	Vector Minimum Signed Halfword
vminsw	VT,VA,VB	Vector Minimum Signed Word
vminsd	VT,VA,VB	Vector Minimum Signed Doubleword
vminub	VT,VA,VB	Vector Minimum Unsigned Byte
vminuh	VT,VA,VB	Vector Minimum Unsigned Halfword
vminuw	VT,VA,VB	Vector Minimum Unsigned Word
vminud	VT,VA,VB	Vector Minimum Unsigned Doubleword
vmladduhm	VT,VA,VB,VC	Vector Multiply-Low-Add Unsigned Halfword Modulo
vmrgew	VT,VA,VB	Vector Merge Even Word
vmrghb	VT,VA,VB	Vector Merge High Byte
vmrghh	VT,VA,VB	Vector Merge High Halfword
vmrghw	VT,VA,VB	Vector Merge High Word
vmrglb	VT,VA,VB	Vector Merge Low Byte
vmrglh	VT,VA,VB	Vector Merge Low Halfword
vmrglw	VT,VA,VB	Vector Merge Low Word
vmrgow	VT,VA,VB	Vector Merge Odd Word
vmsummbm	VT,VA,VB,VC	Vector Multiply-Sum Mixed Byte Modulo
vmsumshm	VT,VA,VB,VC	Vector Multiply-Sum Signed Halfword Modulo
vmsumshs	VT,VA,VB,VC	Vector Multiply-Sum Signed Halfword Saturate
vmsumubm	VT,VA,VB,VC	Vector Multiply-Sum Unsigned Byte Modulo
vmsumuhm	VT,VA,VB,VC	Vector Multiply-Sum Unsigned Halfword Modulo
vmsumuhs	VT,VA,VB,VC	Vector Multiply-Sum Unsigned Halfword Saturate
vmulesb	VT,VA,VB	Vector Multiply Even Signed Byte
vmulesh	VT,VA,VB	Vector Multiply Even Signed Halfword
vmuleub	VT,VA,VB	Vector Multiply Even Unsigned Byte
vmuleuh	VT,VA,VB	Vector Multiply Even Unsigned Halfword
vmulosb	VT,VA,VB	Vector Multiply Odd Signed Byte
vmulosh	VT,VA,VB	Vector Multiply Odd Signed Halfword
vmuloub	VT,VA,VB	Vector Multiply Odd Unsigned Byte
vmulouh	VT,VA,VB	Vector Multiply Odd Unsigned Halfword
vncipher	VT,VA,VB	Vector AES Inverse Cipher
vncipherlast	VT,VA,VB	Vector AES Inverse Cipher Last

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 7 of 11)

Mnemonic	Operands	Description
vnmsubfp	VT,VA,VC,VB	Vector Negative Multiply-Subtract Single-Precision
vnor	VT,VA,VB	Vector Logical NOR
vor	VT,VA,VB	Vector Logical OR
vperm	VT,VA,VB,VC	Vector Permute
vpermxor	VT,VA,VB,VC	Vector Permute Xor
vpkpx	VT,VA,VB	Vector Pack Pixel
vpksdss	VT,VA,VB	Vector Pack Signed Doubleword Signed Saturate
vpksdus	VT,VA,VB	Vector Pack Signed Doubleword Unsigned Saturate
vpkshss	VT,VA,VB	Vector Pack Signed Halfword Signed Saturate
vpkshus	VT,VA,VB	Vector Pack Signed Halfword Unsigned Saturate
vpkswss	VT,VA,VB	Vector Pack Signed Word Signed Saturate
vpkswus	VT,VA,VB	Vector Pack Signed Word Unsigned Saturate
vpkudum	VT,VA,VB	Vector Pack Unsigned Doubleword Unsigned Modulo
vpkudus	VT,VA,VB	Vector Pack Unsigned Doubleword Unsigned Saturate
vpkuhum	VT,VA,VB	Vector Pack Unsigned Halfword Unsigned Modulo
vpkuhus	VT,VA,VB	Vector Pack Unsigned Halfword Unsigned Saturate
vpkuwum	VT,VA,VB	Vector Pack Unsigned Word Unsigned Modulo
vpkuwus	VT,VA,VB	Vector Pack Unsigned Word Unsigned Saturate
vpmsumb	VT,VA,VB	Vector Polynomial Multiply-Sum Byte
vpmsumd	VT,VA,VB	Vector Polynomial Multiply-Sum Doubleword
vpmsumh	VT,VA,VB	Vector Polynomial Multiply-Sum Halfword
vpmsumw	VT,VA,VB	Vector Polynomial Multiply-Sum Word
vrefp	VT,VB	Vector Reciprocal Estimate Single-Precision
vrfim	VT,VB	Vector Round to Single-Precision Integer toward -Infinity
vrfin	VT,VB	Vector Round to Single-Precision Integer Nearest
vrfip	VT,VB	Vector Round to Single-Precision Integer toward +Infinity
vrfiz	VT,VB	Vector Round to Single-Precision Integer toward Zero
vrlb	VT,VA,VB	Vector Rotate Left Byte
vrld	VT,VA,VB	Vector Rotate Left Doubleword
vrlh	VT,VA,VB	Vector Rotate Left Halfword
vrlw	VT,VA,VB	Vector Rotate Left Word
vrsqrtefp	VT,VB	Vector Reciprocal Square Root Estimate Single-Precision
vsbox	VT,VA	Vector Sbox (AES)
vsel	VT,VA,VB,VC	Vector Select
vshasigmad	VT,VA	Vector SHA-512 Sigma Doubleword
vshasigmaw	VT,VA	Vector SHA-256 Sigma Word
vsl	VT,VA,VB	Vector Shift Left

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 8 of 11)

Mnemonic	Operands	Description
vslb	VT,VA,VB	Vector Shift Left Byte
vsld	VT,VA,VB	Vector Shift Left Doubleword
vsldoi	VT,VA,VB,SHB	Vector Shift Left Double by Octet Immediate
vslh	VT,VA,VB	Vector Shift Left Halfword
vslo	VT,VA,VB	Vector Shift Left by Octet
vslw	VT,VA,VB	Vector Shift Left Word
vspltb	VT,VB,UIM	Vector Splat Byte
vsplth	VT,VB,UIM	Vector Splat Halfword
vspltw	VT,VB,UIM	Vector Splat Word
vsr	VT,VA,VB	Vector Shift Right
vsrab	VT,VA,VB	Vector Shift Right Algebraic Byte
vsrad	VT,VA,VB	Vector Shift Right Algebraic Doubleword
vsrah	VT,VA,VB	Vector Shift Right Algebraic Halfword
vsraw	VT,VA,VB	Vector Shift Right Algebraic Word
vsrb	VT,VA,VB	Vector Shift Right Byte
vsrd	VT,VA,VB	Vector Shift Right Doubleword
vsrh	VT,VA,VB	Vector Shift Right Halfword
vsro	VT,VA,VB	Vector Shift Right by Octet
vsrw	VT,VA,VB	Vector Shift Right Word
vsubcuw	VT,VA,VB	Vector Subtract and Write Carry-Out Unsigned Word
vsubfp	VT,VA,VB	Vector Subtract Single-Precision
vsubsbs	VT,VA,VB	Vector Subtract Signed Byte Saturate
vsubshs	VT,VA,VB	Vector Subtract Signed Halfword Saturate
vsubsws	VT,VA,VB	Vector Subtract Signed Word Saturate
vsububm	VT,VA,VB	Vector Subtract Unsigned Byte Modulo
vsububs	VT,VA,VB	Vector Subtract Unsigned Byte Saturate
vsubuhm	VT,VA,VB	Vector Subtract Unsigned Halfword Modulo
vsubuhs	VT,VA,VB	Vector Subtract Unsigned Halfword Saturate
vsubuwm	VT,VA,VB	Vector Subtract Unsigned Word Modulo
vsubuws	VT,VA,VB	Vector Subtract Unsigned Word Saturate
vsubudm	VT,VA,VB	Vector Subtract Unsigned Doubleword Modulo
vsum2sws	VT,VA,VB	Vector Sum across Half Signed Word Saturate
vsum4sbs	VT,VA,VB	Vector Sum across Quarter Signed Byte Saturate
vsum4shs	VT,VA,VB	Vector Sum across Quarter Signed Halfword Saturate
vsum4ubs	VT,VA,VB	Vector Sum across Quarter Unsigned Byte Saturate
vsumsws	VT,VA,VB	Vector Sum across Signed Word Saturate
vupkhpx	VT,VB	Vector Unpack High Pixel

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 9 of 11)

Mnemonic	Operands	Description
vupkhsb	VT,VB	Vector Unpack High Signed Byte
vupkhsh	VT,VB	Vector Unpack High Signed Halfword
vupkhsw	VT,VB	Vector Unpack High Signed Word
vupklpx	VT,VB	Vector Unpack Low Pixel
vupklsb	VT,VB	Vector Unpack Low Signed Byte
vupklsh	VT,VB	Vector Unpack Low Signed Halfword
vupklsw	VT,VB	Vector Unpack Low Signed Word
vxor	VT,RA,RB	Vector Logical XOR
xscmpodp	BF,XA,XB	VSX Scalar Compare Ordered Double-Precision
xscmpudp	BF,XA,XB	VSX Scalar Compare Unordered Double-Precision
xscvspdp	XT,XB	VSX Scalar Convert Single-Precision to Double-Precision (p=1)
xscvsxddp	XT,XB	VSX Scalar Convert Signed Fixed-Point Doubleword to Double-Precision
xscvsxdsp	XT,XB	VSX Scalar Convert and Round Signed Integer Doubleword to Single-Precision Format
xscvuxddp	XT,XB	VSX Scalar Convert Unsigned Fixed-Point Doubleword to Double-Precision
xscvuxdsp	XT,XB	VSX Scalar Convert and Round Unsigned Integer Doubleword to Single-Precision Format
xsmaxdp	XT,XA,XB	VSX Scalar Maximum Double-Precision
xsmindp	XT,XA,XB	VSX Scalar Minimum Double-Precision
xstdivdp	BF,XA,XB	VSX Scalar Test for Software Divide Double-Precision
xstsqrtdp	BF,XB	VSX Scalar Test for Software Square Root Double-Precision
xvabssp	XT,XB	VSX Vector Absolute Value Single-Precision
xvaddsp	XT,XA,XB	VSX Vector Add Single-Precision
xvcmpeqdp	XT,XA,XB	VSX Vector Compare Equal To Double-Precision
xvcmpeqdp.	XT,XA,XB	VSX Vector Compare Equal To Double-Precision and Record
xvcmpeqsp	XT,XA,XB	VSX Vector Compare Equal To Single-Precision
xvcmpeqsp.	XT,XA,XB	VSX Vector Compare Equal To Single-Precision and Record
xvcmpgedp	XT,XA,XB	VSX Vector Compare Greater Than or Equal To Double-Precision
xvcmpgedp.	XT,XA,XB	VSX Vector Compare Greater Than or Equal To Double-Precision and Record
xvcmpgesp	XT,XA,XB	VSX Vector Compare Greater Than or Equal To Single-Precision
xvcmpgesp.	XT,XA,XB	VSX Vector Compare Greater Than or Equal To Single-Precision and Record
xvcmpgtdp	XT,XA,XB	VSX Vector Compare Greater Than Double-Precision
xvcmpgtdp.	XT,XA,XB	VSX Vector Compare Greater Than Double-Precision and Record
xvcmpgtsp	XT,XA,XB	VSX Vector Compare Greater Than Single-Precision
xvcmpgtsp.	XT,XA,XB	VSX Vector Compare Greater Than Single-Precision and Record
xvcpsgnsp	XT,XA,XB	VSX Vector Copy Sign Single-Precision
xvcvspdp	XT,XB	VSX Vector Convert Single-Precision to Double-Precision (p=1)
xvcvspsxds	XT,XB	VSX Vector Convert Single-Precision to Signed Fixed-Point Doubleword Saturate

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 10 of 11)

Mnemonic	Operands	Description
xvcvspsxws	XT,XB	VSX Vector Convert Single-Precision to Signed Fixed-Point Word Saturate
xvcvspuxds	XT,XB	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Doubleword Saturate
xvcvspuxws	XT,XB	VSX Vector Convert Single-Precision to Unsigned Fixed-Point Word Saturate
xvcvsxddp	XT,XB	VSX Vector Convert Signed Fixed-Point Doubleword to Double-Precision
xvcvsxdsp	XT,XB	VSX Vector Convert Signed Fixed-Point Doubleword to Single-Precision
xvcvsxwdp	XT,XB	VSX Vector Convert Signed Fixed-Point Word to Double-Precision
xvcvsxwsp	XT,XB	VSX Vector Convert Signed Fixed-Point Word to Single-Precision
xvcvuxddp	XT,XB	VSX Vector Convert Unsigned Fixed-Point Doubleword to Double-Precision
xvcvuxdsp	XT,XB	VSX Vector Convert Unsigned Fixed-Point Doubleword to Single-Precision
xvcvuxwdp	XT,XB	VSX Vector Convert Unsigned Fixed-Point Word to Double-Precision
xvcvuxwsp	XT,XB	VSX Vector Convert Unsigned Fixed-Point Word to Single-Precision
xvdivsp	XT,XA,XB	VSX Vector Divide Single-Precision
xvmaddasp	XT,XA,XB	VSX Vector Multiply-Add Type-A Single-Precision
xvmaddmsp	XT,XA,XB	VSX Vector Multiply-Add Type-M Single-Precision
xvmaxdp	XT,XA,XB	VSX Vector Maximum Double-Precision
xvmaxsp	XT,XA,XB	VSX Vector Maximum Single-Precision
xvmindp	XT,XA,XB	VSX Vector Minimum Double-Precision
xvminsp	XT,XA,XB	VSX Vector Minimum Single-Precision
xvmsubasp	XT,XA,XB	VSX Vector Multiply-Subtract Type-A Single-Precision
xvmsubmsp	XT,XA,XB	VSX Vector Multiply-Subtract Type-M Single-Precision
xvmulsp	XT,XA,XB	VSX Vector Multiply Single-Precision
xvnabssp	XT,XB	VSX Vector Negative Absolute Value Single-Precision
xvnegsp	XT,XB	VSX Vector Negate Single-Precision
xvnmaddasp	XT,XA,XB	VSX Vector Negative Multiply-Add Type-A Single-Precision
xvnmaddmsp	XT,XA,XB	VSX Vector Negative Multiply-Add Type-M Single-Precision
xvnmsubasp	XT,XA,XB	VSX Vector Negative Multiply-Subtract Type-A Single-Precision
xvnmsubmsp	XT,XA,XB	VSX Vector Negative Multiply-Subtract Type-M Single-Precision
xvresp	XT,XB	VSX Vector Reciprocal Estimate Single-Precision
xvrspi	XT,XB	VSX Vector Round to Single-Precision Integer
xvrspic	XT,XB	VSX Vector Round to Single-Precision Integer using Current rounding mode
xvrspim	XT,XB	VSX Vector Round to Single-Precision Integer toward -Infinity
xvrspip	XT,XB	VSX Vector Round to Single-Precision Integer toward +Infinity
xvrspiz	XT,XB	VSX Vector Round to Single-Precision Integer toward Zero
xvrsqrtesp	XT,XB	VSX Vector Reciprocal Square Root Estimate Single-Precision
xvsqrtsp	XT,XB	VSX Vector Square Root Single-Precision
xvsubsp	XT,XA,XB	VSX Vector Subtract Single-Precision
xvtdivdp	BF,XA,XB	VSX Vector Test for software Divide Double-Precision

Table 3-5. Instructions that Trigger an Exception on Consumption of a Scalar SP Denorm (Sheet 11 of 11)

Mnemonic	Operands	Description
xvtdivsp	BF,XA,XB	VSX Vector Test for software Divide Single-Precision
xvtsqrtdp	BF,XB	VSX Vector Test for software Square Root Double-Precision
xvtsqrtsp	BF,XB	VSX Vector Test for software Square Root Single-Precision
xxland	XT,XA,XB	VSX Logical AND
xxlandc	XT,XA,XB	VSX Logical AND with Complement
xxlnor	XT,XA,XB	VSX Logical NOR
xxlor	XT,XA,XB	VSX Logical OR
xxlxor	XT,XA,XB	VSX Logical XOR
xxmrghw	XT,XA,XB	VSX Merge High Word
xxmrglw	XT,XA,XB	VSX Merge Low Word
xxpermdi	XT,XA,XB,DM	VSX Permute Doubleword Immediate
xxsel	XT,XA,XB,XC	VSX Select
xxsldwi	XT,XA,XB,SHW	VSX Shift Left Double by Word Immediate
xxspltd	XT,XA	VSX Splat Doubleword (microcode)
xxspltw	XT,XB,UIM	VSX Splat Word

Table 3-6 lists BFP and VSX store instructions that cause a soft patch request if the address is unaligned and LSU flushes to microcode. For example, a store crosses 4K or the address crosses an 8-byte boundary for doubleword (or smaller) operations and a 16-byte boundary for quadword operations in DAWR mode (DWARX0 for POWER8).

Table 3-6. Soft Patch Instruction on Unaligned Stores

Mnemonic	Operands	Description
stfd	FRS,D(RA)	Store Float Double
stfdu	FRS,D(RA)	Store Float Double with Update
stfdux	FRS,RA,RB	Store Float Double with Update Indexed
stfdx	FRS,RA,RB	Store Float Double Indexed
stfiwx	FRS,RA,RB	Store Float as Integer Word Indexed
stxsdx	XS,RA,RB	Store VSR Scalar Doubleword Indexed
stxsiwx	XS,RA,RB	Store VSX Scalar Integer Word Indexed
stxvd2x	XS,RA,RB	Store VSR Vector Doubleword*2 Indexed (only in LE mode)
stxvw4x	XS,RA,RB	Store VSR Vector Word*4 Indexed (only in LE mode)

3.6 Book II - Virtual Environment Architecture

3.6.1 Classes of Instructions

3.6.1.1 Optional Instructions

None.

3.6.2 Storage Model

The POWER8 core supports:

- Coherence block size of 128 bytes
- Weakly consistent access ordering of stores and loads per the Power ISA Book II, unless the special WIMG encoding designating all accesses to specific pages as being strongly ordered (SAO) is used to translate the access.
- Transactional memory accesses are performed and ordered as described in the Power ISA.

3.6.3 Cache

The POWER8 chip contains three levels of cache hierarchy. All the caches (L1 I-cache, L1 D-cache, and the L2 and L3 caches) are dynamically shared among the eight threads. A cache block might be installed by one thread and used by the other threads. The basic coherence block size for the POWER8 core is 128 bytes.

The POWER8 chip automatically maintains the coherency of all data cached in these caches. The L1 cache employs Harvard cache organization, with separate L1 I-cache and L1 D-cache. L2 and L3 caches are unified. Because some levels of the cache hierarchy contain both instructions and data, when an instruction cache reload request is serviced by the L2 and/or the L3 caches, it is done so in a coherent manner.

The processor keeps the instruction storage consistent with the data storage. All cache lines in the L1 I-cache and L1 D-cache are also present in the L2 cache (inclusive property maintained).

The L1 I-cache is 8-way set associative and is indexed with five effective address bits (EA[51:55]). A particular physical block of memory with a given real address can be found in one of two positions in the L1 I-cache. The tag comparison associated with lookups in this cache (as well as all other caches in the system) are done using physical addresses, so that there are no synonym or alias hazards that must be explicitly handled by the system software.

The L1 D-cache is 8-way set associative and is indexed with six effective address bits (EA[51:56]). A particular physical block of memory with a given real address can only be found at a particular location in the L1 D-cache. On each access, the tag comparison is done with the physical address. On a cache miss, the cache reload mechanism searches the other seven related sets to determine if the required real address block is located elsewhere in the cache, and if so, it appropriately eliminates these copies.

In addition to maintaining caches, the POWER8 chip also includes several types of queues that act as logical predecessors and extensions to the caches. In particular, the machine contains store queues for holding store data above the caches, cast-out queues for holding modified data that has been pushed out of the caches (by the replacement algorithm, cache control instructions, or snoop requests), and others. All of these queues are maintained coherent by the hardware. In general, their presence should not be observable by either software or system hardware.

3.6.4 Data Prefetch

The POWER8 core provides an aggressive hardware-based data prefetching engine that is designed to work well for stride-one technical workloads with up to 16 streams. The 16 streams can be dynamically shared among the eight threads, with a sharing policy described elsewhere in more details. The POWER8 core implements enhanced data prefetching (**edcbt** instruction). For enhanced data prefetching, each thread can employ up to sixteen software initiated streams in ST mode. In SMT2 mode, the sixteen entries are shared between the two threads. In SMT4 mode, there are two groups of eight entries, and each group is shared between two threads. In SMT8 mode, there are four groups of four entries, and each group is shared between two threads

The POWER8 core also supports instruction cache block touch (**icbt**) by mapping it to **dcbtst** to prefetch into the L2 cache.

Another feature of the POWER8 core is the DSCR[58] bit that turns off hardware load-stream prefetching that can be accessed in problem state.

3.6.5 Effect of Operand Placement on Performance

In general, the POWER8 core provides excellent performance for storage accesses to naturally aligned boundaries (that is, to the boundaries defined by the operand size). In addition, the POWER8 core provides excellent hardware support for most unaligned storage accesses cases. In particular, cacheable load operations within 128-byte aligned sections of memory that hit in the L1 D-cache are performed at full speed independent of address alignment. Note that many storage accesses that cross page boundaries and segment boundaries are performed by the hardware. See *Section 3.1.4 Storage Access Alignment Support Overview* on page 42 for details.

3.6.6 Storage Model

3.6.6.1 Atomicity

The POWER8 core is fully compliant with the architectural requirement for single-copy atomicity on naturally aligned storage accesses. This includes the quadword data atomicity associated with the **lq, lqarx, stq**, and **stqcx.** instructions. Furthermore, transactional mode accesses executed by a given thread appear to execute atomically to all other threads in the system (assuming the transaction succeeds).

3.6.6.2 Transactional Memory

The POWER8 processor supports Transactional Memory as described in the Power ISA.

The Transaction Exception and Status Register (TEXASR) is used to record various failure conditions that are described in the Power ISA. In addition, TEXASR[15] is used to specify various implementation-specific transaction failure causes that are not architected. The POWER8 processor sets TEXASR[15]='1' for the following reasons (implementation-specific transactional failure causes):

- Instruction fetch to caching-inhibited page in transactional mode.
- Recovery in transactional or suspend mode.
- Quiesce request in transaction or suspend mode.

The persistent bit is set to '0' for all of the above cases.

3.6.6.3 Storage Access Ordering

The architecture defines a weakly ordered storage model for most types of storage access scenarios. For these cases, the POWER8 core takes advantage of this relaxed requirement to achieve better performance through out-of-order instruction execution and out-of-order bus transactions. As a result, if strongly-ordered storage accesses are required, software must use the appropriate synchronizing instruction (**sync**, **ptesync**, **eieio**, or **lwsync**) to enforce order explicitly, or perform these accesses to regions marked with attributes that require the hardware to enforce strong ordering.

The POWER8 core employs the Real Mode Storage Control (RMSC) facility. Therefore, stores to storage marked as *non-guarded*, can be executed out-of-order, if the access is to well-behaved memory, as specified in the Real Mode Storage Control facility. Otherwise, stores to storage marked as *guarded*, cannot be executed out-of-order.

3.6.7 Atomic Updates and Reservations

The reservation granule size in the POWER8 core is 128 bytes.

There is at most one reservation per thread at any point in time.

3.6.8 Storage Control Instructions

3.6.8.1 Overview of Key Aspects of Storage Control Instructions

In the POWER8 core, all cache control instructions operate on aligned 128-byte sections of storage. *Table 3-7* summarizes many of the key aspects of the storage control instructions.

Table 3-7. Storage Control Instructions

Aspect	Book II Cache Instructions					
	icbi	**dcbt**	**dcbtst**	**dcbz**	**dcbst**	**dcbf/dcbfl**
Granularity	128 bytes	128 bytes	128 bytes	128 bytes	128 bytes	128 bytes
Semantic checking	load (DSI on storage exception)	load (NOP on storage exception)	load (NOP on storage exception)	store (DSI on storage exception)	load (DSI on storage exception)	load (DSI on storage exception)
"r" bit update	yes	yes	yes	yes	yes	yes
"c" bit update	no	no	no	yes	no	no
L1 I-cache effect	see *Section 3.6.8.2* on page 86	none	none	none	none	none
L1 D-cache effect	none	see *Section 3.6.8.4* on page 86	see *Section 3.6.8.4* on page 86	as define in architecture	NOP	as define in architecture
L2 cache effect	none	see *Section 3.6.8.4* on page 86	see *Section 3.6.8.4* on page 86	see *Section 3.6.8.7* on page 87	see *Section 3.6.8.8* on page 88	see *Section 3.6.8.9* on page 88
L3 cache effect	none	see *Section 3.6.8.4* on page 86	see *Section 3.6.8.4* on page 86	see *Section 3.6.8.7* on page 87	see *Section 3.6.8.8* on page 88	see *Section 3.6.8.9* on page 88
TLB effect	reload as required	reload as required	reload as required	reload as required	reload as required	reload as required
SLB effect	reload as required	None (NOP if miss)	None (NOP if miss)	reload as required	reload as required	reload as required

3.6.8.2 Instruction Cache Block Invalidate (icbi)

The instruction cache block size for **icbi** on the POWER8 core is 128 bytes.

The POWER8 core implements a split I/D L1 cache, and the I-cache is kept coherent with the L2/L3 cache. All I-cache lines are present in the L2/L3 (L2/L3 inclusive). When any modification is made to cache lines contained in the L2 cache, the L2 invalidates copies in L1 I-caches. However, to perform self or cross-thread code modification and ensure the modifications take effect in a synchronized manner, software must follow the Power ISA requirements for performing self-modifying code, as specified in the Instruction Storage section of Power ISA Book II.

3.6.8.3 Instruction Cache Synchronize (isync)

As a performance optimization, the POWER8 core internally tracks and scoreboards **icbi** instructions that are required to be synchronized by the **isync** instruction. When the **isync** instruction is executed, this scoreboard bit is checked to see whether or not the machine must *flush and refetch* the instructions following the **isync**.

3.6.8.4 Data Cache Block Touch (dcbt and dcbtst)

The data cache block size for **dcbt** and **dcbtst** on the POWER8 core is 128 bytes.

The **dcbtst** instruction operates exactly the same way as the **dcbt** instruction.

These instructions act as a touch for the D-cache hierarchy and the TLB. If data translation is enabled (MSR[DR] = 1), and an SLB miss results, the instruction will be treated as a *NOP*. If a TLB miss results, the instruction reloads the TLB (and sets the reference bit). Once past translation, if the page protection attributes prohibit access, the page is marked *cache inhibited*, the page is marked *guarded*, or the processors' D-cache is disabled (via the bits in the HID4 Register), the instruction is finished as a *NOP* and does not reload the cache. Otherwise, the instruction checks the state of the L1 D-cache, and if it is not present, it initiates a reload. Note that this might also reload the L2 cache and/or the L3 cache with the addressed block if it is not already present in these caches. If the cache block is already present in the L1 D-cache, the cache content is not altered. Note that if the **dcbt** or **dcbtst** instruction does reload cache blocks, it affects the state of the cache replacement algorithm bits.

The POWER8 core does implement the optional extension to the **dcbt** instruction that allows software to directly engage a data stream prefetch from a particular address.

3.6.8.5 Data Cache Block Touch - No Access Needed Anymore (TH = '10001')

The POWER8 core supports this as specified in the Power ISA.

3.6.8.6 Data Cache Block Touch - Transient (TH = '10000')

The POWER8 core implements the load and store version of these transient touch instructions (**dcbtct**, **dcbtds**, **dcbtt**, **dcbtstct**, and **dcbtstt**). The transient property of a cache line is retained in the L3 cache for both the load and store version of the transient touch instructions. The transient property of a cache line is retained in the L2 cache for the load version of the transient touch instruction for the case that the line is loaded but not stored into. In this transient state, the transient line becomes the most likely cache line in its congruence class to be replaced next, thus preserving the other cache lines in that congruence class. This behavior is useful if it is known that a set of lines will be loaded or stored with a low probability for temporal cache reuse and it is desirable that they be as minimally intrusive to the cache as possible (for example, displacing as few lines in the cache as possible). Reading or writing a large array with the help of transient touch instructions only impacts one of the eight sets in the L3 cache, and reading a large array with the help of transient touch instructions only impacts one of the eight sets of the L2 cache.

3.6.8.7 Data Cache Block Zero (dcbz)

The data cache block size for **dcbz** on the POWER8 core is 128 bytes.

The function of **dcbz** is performed in the L2 cache. As a result, if the block addressed by the **dcbz** is present in the L1 D-cache, the block is invalidated before the operation is sent to the L2 cache logic for execution. The L2 cache gains exclusive access to the block (without actually reading the old data) and performs the zeroing function in a broadside manner.

If the cache block specified by the **dcbz** instruction contains an error (even one that is not correctable with ECC), the contents of all locations within the block are set to zeros in the L2 cache. If the specified block in the L2 cache does not contain a hard fault, a subsequent load from any location within the cache block returns zeros and does not cause a machine check interrupt.

If the block addressed by the **dcbz** instruction is in a memory region marked *cache inhibited*, or if the L1 D-cache or L2 cache is disabled (using the bits in the HID registers), then the instruction causes an alignment interrupt to occur.

3.6.8.8 Data Cache Block Store (dcbst)

The data cache block size for ***dcbst*** on the POWER8 core is 128 bytes.

The **dcbst** instruction has no direct effect on the L1 D-cache (because it is store-through, it never contains modified data). It also has no direct effect on the L2 cache or L3 cache (both of these are kept coherent with memory and I/O, so that nothing special needs to be done). As a result, the instruction simply goes through address translation, reports any errors, and is completed. The instruction is not sent to the storage subsystem, and consequently it does not broadcast any transactions onto the inter-processor SMP interconnect.

3.6.8.9 Data Cache Block Flush (dcbf, dcbfl and dcbflp)

The data cache block size for **dcbf**, **dcbfl** and **dcbflp** on the POWER8 core is 128 bytes.

The POWER8 core supports **dcbf** (L = 0), **dcbfl** (**dcbf** with L = 1) and **dcbflp** (**dcbf** with L = 3) as specified in the Power ISA.

3.6.8.10 Load and Reserve and Store Conditional Instructions

The reservation granule size for the POWER8 core is 128 bytes.

An attempt to execute a non-halfword aligned **lharx or sthcx.,** or a non-word aligned **lwarx** or **stwcx.**, or a non-doubleword aligned **ldarx** or **stdcx.**, or a non-quadword aligned **lqarx** or **stqcx.** to a cacheable address causes an alignment interrupt.

An attempt to execute a **lbarx, lharx, lwarx**, **ldarx**, **lqarx**, **stbcx.**, **sthcx., stwcx.**, **stdcx.**, or a **stqcx.** instruction to storage marked cache inhibited causes a data storage interrupt independent of the address alignment.

There are separate reservations per thread.

3.6.8.11 sync Instruction

The POWER8 design achieves high performance by exploiting speculative out-of-order instruction execution. The *heavyweight sync* (**hwsync)** instruction, as defined in the architecture, acts as a serious barrier to this type of aggressive execution and therefore, can have a dramatic effect on performance. Although the POWER8 core has optimized the performance of **hwsync** to some degree, care should be exercised in the indiscriminate use of this instruction. As a performance consideration, software should attempt to use the lightweight version of **sync** (often referred to as **lwsync** in this document) whenever possible. Unless otherwise stated, **sync** refers to **hwsync**.

The POWER8 core implements the **ptesync** for use in synchronizing page table updates.

See the Power ISA Books II and III-S for a complete description of the different forms of the **sync** instruction.

The POWER8 core does not support the optional elemental memory barriers described in the Power ISA. Instead, the Power8 design ignores the EB field of the **sync** instruction and the L-field solely determines which version of the **sync** instruction (**sync**/**lwsync**/**ptesync**) is performed by the hardware.

3.6.8.12 eieio Instruction

The POWER8 core implements **eieio** as described in the Power ISA.

In the POWER8 nest logic, the store queues above the L2 cache attempt to gather sequential both cacheable and cache-inhibited store operations to improve bandwidth. If this behavior is not desired, software must insert either an **eieio** (preferable for performance) or a **sync** to prevent it.

3.6.8.13 miso Instruction

The POWER8 core implements the **miso** instruction as a NOP. It has no effect on the stores.

3.6.8.14 Transactional Memory Instructions

The POWER8 core implements the transactional memory instructions as described in the Power ISA.

Table 3-8 lists how certain cache and TLB management instructions affect transactions.

Table 3-8. Cache and TLB Management Instruction Effects on Transactional Accesses (Sheet 1 of 2)

Mode			
Instruction	TM State	Fails Transaction	TEXASR bit set
tlbie	T	Always	8 - disallowed
tlbie	S	When the virtual address it is attempting to invalidate hits in the bloom filter for the current transaction	14 - translation invalidation conflict
tlbiel	T	Always	8 - disallowed
tlbiel	S	Never	N/A
dcbt (any TH)	T	Never (unless it causes a castout of the TM footprint)	10 - footprint overflow
dcbt (any TH)	S	Never (unless it causes a castout of the TM footprint)	10 - footprint overflow
dcbst	T	Always	8 - disallowed
dcbst	S	Never (**dcbst** is treated as a NOP in this case)	11 - self-induced conflict
dcbf (L=0,1)	T	Always	8 - disallowed
dcbf (L=0,1)	S	When the block (line) being pushed out of the cache is part of the TM footprint	11 - self-induced conflict
dcbf (L=3)	T	Always	8 - disallowed
dcbf (L=3)	S	Never	N/A
dcbz	T	Never (unless **dcbz** causes a castout of the TM footprint)	N/A
dcbz	S	Case 1: When the block (line) being zero'ed is part of the TM footprint Case 2: When the **dcbz** causes a castout of the TM load or store footprint	Case 1: 11 - self-induced conflict Case 2: 11 - footprint overflow
dcbtst	T	Never (unless **dcbtst** causes a castout of the TM footprint)	10 - footprint overflow
dcbtst	S	Never (unless **dcbtst** causes a castout of the TM footprint)	10 - footprint overflow
icbi	T	Always	8 - disallowed

Table 3-8. Cache and TLB Management Instruction Effects on Transactional Accesses (Sheet 2 of 2)

Mode			
Instruction	TM State	Fails Transaction	TEXASR bit set
icbi	S	Never (**icbi** is treated as NOP with regards to transaction failure in suspend mode)	N/A
icbt	T	Never (unless **icbt** causes a castout of the TM footprint)	10 - footprint overflow
icbt	S	Never (unless **icbt** causes a castout of the TM footprint)	10 - footprint overflow

3.6.9 Timer Facilities

3.6.9.1 Time Base

Time base is designed to tick at the rate of time-of-day (TOD). In other words, bit 59 of the Time Base Register increments at the 32 MHz clock. There is one time base per processor core that is shared by all the threads, running on a core. There is one decrementer per thread.

The POWER8 core implements two time-base modes: POWER8 time-base mode and non-POWER8 time-base mode. They are selectable by setting a mode bit in the Time Facility Management Register (TFMR).

Time Facility Management Register

The Time Facility Management Register (TFMR) is an SPR that is accessible only in the hypervisor state. Executing a move to or move from TFMR in a nonhypervisor state causes a privileged interrupt. There is one TFMR per processor core that is shared among the threads. The TFMR is used as both a status and control register.

POWER8 Time-Base Mode

The time-base function uses an external time-of-day (TOD) clock, which is independent of the processor frequency. This is required to support dynamic frequency variation for power management. The external TOD oscillator can be 16 MHz or 32 MHz. The external TOD oscillator is sampled to provide a 32 MHz step signal, which is distributed to all processors in the system.

Bits 0:59 of the TB are incremented at the 32 MHz frequency as provided by the distributed step signal. Bits 60:63 of the TB are incremented at a fixed frequency of 500 MHz. If the value of bits 60:63 is '1111' (saturated), it is held until the 32 MHz step signal causes bit 59 to change. At that time, bits 60:63 are allowed to change to '0000'.

To support multi-node configurations across multiple oscillator domains, error detection and recovery, and concurrent maintenance, the POWER8 core uses the following means of synchronizing the time bases across all processors.Each multicore processor chip contains a Time-of-Day (TOD) Register. The chip TOD registers are first synchronized across all the processor chips. Then, the time-base registers in each processor core are synchronized to the chip TOD. Also encoded on the step signal is a synchronization pulse which is used for synchronization and error checking. The synchronization mechanism requires system operations to complete within a sync interval. The sync interval can be set via the TFMR bits to be, for example, 1µs, 2 µs (default, corresponds to TB bit 53), 4 µs, or 8 µs.

Error checking includes parity checks on all registers, and functional checking such as missing step signal detection and synchronization errors. The step signal rate is defined in the POWER8 mode to be 32 MHz, and the logic checks for the correct number of steps for each synchronization signal (which is selected by TFMR). After the TB is operational, the hardware also detects a missing step signal, which requires specifying in the TFMR the maximum number of processor cycles allowed without seeing a 32 MHz step signal, for the fastest allowable operating frequency. The TFMR maximum cycle step time-out should be specified as (2×31.25 ns) / (minimum processor cycle time in ns $\times$ 4).

The initial synchronization requires some software sequencing, which is performed by writing values to the TFMR (via **mtspr**). The TFMR also indicates the status of the various time facilities. The status bits in the TFMR are read-only, not modified by **mtspr** to the TFMR. The time facility logic implements error detection for hardware and also for invalid software sequencing. Because synchronized time is critical to a system, writes to the time base or the TFMR that would break synchronization cause the logic to enter an error state and trigger a hypervisor maintenance interrupt.

To initially set the time and synchronize the time base values, software must synchronize all processors in the system and choose one processor to perform updates to the TFMR via a read-modify-write operation to preserve the other bits. This sequence assumes the external TOD oscillator distribution is already running.

After the chip TOD is running on all chips and the TB is running on the processor that drove this sequence, software must then release the remaining processors to synchronize their TB registers to their corresponding chip TOD.

3.6.10 Hypervisor Decrementer (HDEC)

There is one hypervisor decrementer register per thread. HDEC decrements every time TB bit 63 is incremented.

3.6.11 Decrementer (DEC)

There is one decrementer register per thread. DEC decrements every time TB bit 63 is incremented.

3.6.12 Book II Invalid Forms

The results of executing an invalid form of an instruction in Book II or an instance of such an instruction for which the architecture specifies that some results are undefined, are described here for the cases in which executing an instruction does not cause an exception. Only results that differ from those specified by the architecture are described in the following list.

- Instruction with reserved fields
  Bits in reserved fields are ignored; the results of executing an instruction in which one or more reserved bits are '1' is the same as if the bits were '0'.
- Transactional Memory instructions and Store Conditional instructions (bit 31 is ignored)
  Bit 31 of **tbegin.**, **tend.**, **tabort.**, **tabortwc.**, **tabortdc.**, **tabortwci.**, **tabortdci.**, **tsr.**, **treclaim.**, **stbcx.**, **sthcx.**, **stwcx.**, **stdcx.** and **stqcx.** is ignored. Bit 31 = '1' or bit 31 = '0' is treated as the same given that other x-form instructions, which implicitly set CR and have no "non-record" from variant. Ignoring bit 31 is an acceptable way to handle this invalid form.
- **mftb** instructions
  This instruction produces the same result as the **mfspr** instruction. For a complete description on the

associated invalid forms, see *Section 3.1.4.9 Move To/From Special Purpose Register (SPR) Instructions* on page 63.

3.7 Book III - Operating Environment Architecture

3.7.1 Classes of Instructions

3.7.1.1 Storage Control Instructions

The POWER8 core does provide support for the following instructions:

- **tlbie** - TLB invalidate entry (large and small page)
- **tlbiel** - Processor local form of TLB invalidate entry (large and small page, and the IS field)
- **tlbsync** - TLB synchronize
- **slbmte**- Segment Lookaside Buffer Move To Entry
- **slbmfev**- Segment Lookaside Buffer Move From Entry VSID
- **slbmfee**- Segment Lookaside Buffer Move From Entry ESID
- **slbfee.**- Segment Lookaside Buffer Find Entry ESID
- **slbie** - SLB invalidate entry
- **slbia** - SLB invalidate all
- **mtsr** - Move to segment register (Bridge Facility)
- **mtsrin** - Move to segment register indirect (Bridge Facility)
- **mfsr** - Move from segment register (Bridge Facility)
- **mfsrin** - Move from segment register indirect (Bridge Facility)
- **mtmsr** - Move to Machine State Register (32-bit)
- **mtmsrd -** Move to Machine State Register (64-bit)
- **sc -** System Call
- **rfid -** Return From Interrupt Doubleword
- **hrfid -** Hypervisor Return From Interrupt Doubleword

The POWER8 core does not provide support for the following optional or obsolete instructions (attempted use of these results in a hypervisor emulation assistance interrupt):

- **tlbia** - TLB invalidate all
- **tlbiex** - TLB invalidate entry by index (obsolete)
- **slbiex** - SLB invalidate entry by index (obsolete)
- **dcba** - Data cache block allocate (Book II; obsolete)
- **dcbi** - Data cache block invalidate (obsolete)
- **rfi** - Return from interrupt (32-bit; obsolete)

The following instruction variants are implemented:

- **ptesync** - Page table synchronize
- **hwsync** - Heavyweight synchronize
- **lwsync**- Lightweight synchronize

3.7.1.2 Reserved Instructions

The architecture breaks the reserved instruction class down into several categories as described in the *Reserved Instructions* appendix of the Power ISA. The POWER8 processor core behaves in the following manner with respect to these categories:

- Primary opcode equals zero.
- POWER Architecture instructions not in the Power ISA. The POWER8 core takes a hypervisor emulation assistance interrupt. See the complete list in the Power ISA appendices.
- Service processor "Attention" instruction.
- In addition, there are several implementation-specific registers available for access through the **mtspr** and **mfspr** instructions. These are described in *Section 3.7.3.4 Move To/From Special Purpose Register Instructions* on page 96.

3.7.2 Branch Processor

3.7.2.1 SRR1 Register

In the POWER8 processor core, the SRR1 is implemented per the Power ISA.

3.7.2.2 MSR Register

In the POWER8 processor core, the MSR is implemented per the Power ISA

3.7.2.3 Branch Processor Instructions

Branch Processor Instruction with Undefined Results

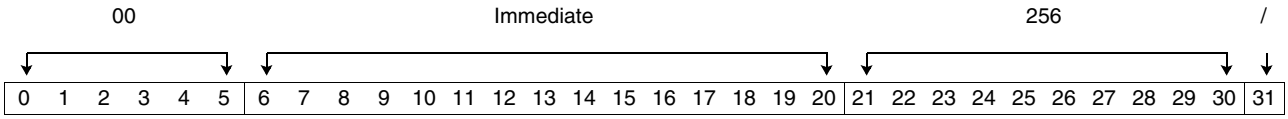
The results of executing an invalid form of a branch instruction or an instance of a branch instruction for which the architecture specifies that some results are undefined are described as follows.

System Call SC-Form

In the POWER8 processor core, the system call (**sc**) instruction is implemented per the Power ISA.

Support Processor Attention Instruction

The POWER8 processor core supports a special, implementation-dependent instruction for signaling an attention to the support processor:



The immediate field (I) has no effect on the operation of this instruction in the POWER8 processor core.

If HID0[31] = '1' (support-processor attention enable bit is set), this instruction causes all preceding instructions to run to completion, the machine to quiesce, and the assertion of the support processor attention signal. Instruction execution does not resume until the support processor signals it to do so.

If HID0[31] = '0' (support processor attention enable not set), this instruction causes a hypervisor emulation assistance interrupt.

3.7.2.4 Current Instruction Address Breakpoint (CIABR)

The POWER8 processor core supports the CIAB Register as implemented per the Power ISA.

3.7.2.5 Instruction Effective to Real Address Translation Cache (I-ERAT)

The POWER8 processor core includes a 64-entry, 64-way set associative instruction effective-to-real address translation (I-ERAT) for fast translation of instruction effective addresses into physical (real) addresses. The ERAT is dynamically shared between all eight threads. The ERAT is implemented as a CAM that supports page sizes of 4 KB, 64 KB, and 16 MB. Instruction fetches to 16 GB pages are installed in the I-ERAT as multiple 16 MB page entries as needed. Likewise, instruction fetches to 1 MB pages are installed as multiple 64 KB pages as needed. Entries for hypervisor relocate off can be shared between all eight threads (regardless of LPAR mode), further increasing the capacity of the ERAT.

Because addresses associated with non-hypervisor real mode accesses are translated differently than those associated with virtual-mode accesses, the IERAT must keep the MSR[IR] and MSR[HV] bits (along with various bits of translation information) in each entry. This allows the I-ERAT to distinguish between translations that are valid for the various modes of operation. Because the content of each I-ERAT entry is the result of a page table search based on the contents of an SLB entry, to maintain consistency with the SLB (or segment registers), the following instructions cause all entries in the I-ERAT that belong to the thread executing the instruction to be invalidated. The ERAT compares as many effective address bits as are available in the various invalidate operations.

- **mtsr** or **mtsrin** instructions - used for segment register changes in 32-bit operating systems
- **slbia**
- **mtiamr**

The **slbie** instruction causes invalidation of a D-ERAT entry belonging to the thread (no impact to the other thread) only if there is a perfect address match (that is, for invalidation effective address bits, EA[0:35] are matched for **slbie** small 256 MB segment, EA[0:23] are matched for **slbie** 1 TB segment).

The **tlbie** instruction (or the detection of snooped-tlbie operations) invalidates all D-ERAT entries (irrespective of the thread) in the D-ERAT that have a perfect match. In other words, the entry is invalidated only if:

- EA[36:51] are matched for tlbie small page
- EA[36:47] are matched for tlbie 64 KB page
- EA[36:43] are matched for tlbie 1 MB page
- EA[36:39] are matched for tlbie 16 MB page

EA[24:29] are matched for tlbie 16 GB page

Upon power-on, each I-ERAT entry is set to the invalid state.

I and G bit setting in the I-ERAT is done based on *Table 3-9*.

Table 3-9. I-ERAT I and G Bit Setting

Condition								Resulting Action
MSR [IR]	MSR [HV]	LPCR[0]	HID4[33]	First access I=1 Fetch	HID4[0:4] RMSC (above line)	I	G	
1	x	x	x	x	x	PTE(I)	PTE(G)	If G='0', the page is written into the I-ERAT using the I-bit value and page size determined from the PTE and described above. If G='1', an ISI is taken.
0	0	1	x	x	x	PTE(I)	PTE(G)	Virtual real mode: If G = '0', the page is written into the I-ERAT using the I-bit value and page size determined from the PTE and described above. If G='1', an ISI is taken.
0	0	0	x	x	x	0	0	RMOR mode: set IG = '00'
0	1	x	0	yes	no	1	0	Legacy RMSC mode: If access is below the guarded line established by HID4[0:4], access is allowed and a 4K page is installed in I-ERAT as I='1', G = '0'.
0	1	x	0	no	no	0	0	Legacy RMSC mode. If access is below the guarded line established by HID4[0:4], access is allowed and a 4K page is installed in I-ERAT as I='0',G = '0'.
0	1	x	0	x	yes	N/A	N/A	Legacy RMSC mode. If access is above the guarded line established by HID4[0:4], access is not allowed resulting in an ISI due to an instruction fetch to guarded storage.
0	1	x	1	yes	x	1	N/A	Page-based RMSC mode. 16 M page is installed in I-ERAT with I = '1', G = '1'.
0	1	x	1	no	x	0	N/A	Page-based RMSC mode. 16 M page is installed in I-ERAT with I = '0', G = '0'.

3.7.3 Fixed-Point Processor

3.7.3.1 Processor Version Register (PVR)

The processor version number (PVR[0:15]) for the POWER8 processor in the SCM is x'004D'. In future versions of the POWER8 core, this section of the version number will only change if there are significant software-visible changes in the design.

The processor revision level (PVR[16:31]) starts at x'0100', indicating revision '1.0'. As revisions are made, bits [29:31] will indicate minor revisions. Similarly, bits [20:23] indicate major changes.

Example: The PVR value for POWER8 processor in the SCM for design level 1 is: x'004D0100'.

3.7.3.2 Processor ID Register (PIR)

The processor identification register (PIR) is a 32-bit register that holds a processor identification tag in the 9 least-significant bits [23:31]. This tag is used for tagging bus transactions and for processor differentiation in multiprocessor systems.

Bits	Field Name	Description
0:21	Reserved	Read as zeros
22:24	ChID	Chip ID
25:28	CoID	Core number
29:31	TID	Thread ID

The PIR is a read-only register. During power-on reset, PIR is set to a unique value for each processor in a multiprocessor system.

3.7.3.3 Chip Information Register (CIR)

The POWER8 processor implements the CIR per the Power ISA.

3.7.3.4 Move To/From Special Purpose Register Instructions

The POWER8 core supports special purpose registers listed in *Table 3-10*. Many of these SPRs are only accessible in hypervisor or privileged modes. A handful of these registers (for example, DSCR) are also user-mode accessible through a second SPR number.

To support multithreading, some of the SPRs are replicated in the POWER8 core, while the others are shared, as shown in the SMT column in *Table 3-10*. In the table column headers, Prob indicates problem state (HV = x, PR = 1), Priv indicates privileged state (HV = 0, PR = 0), and Hyp indicates Hypervisor state (HV = 1, PR = 0). In the SPR specific rows, Priv indicates that a privileged instruction type program interrupt will occur in that state for the attempted read or write of the SPR. Illegal indicates a hypervisor emulation assistance interrupt will occur. NOP indicates the instruction will be treated as a NOP. A blank under each column indicates the access will be performed normally.

Table 3-10. SPR Table (Sheet 1 of 5)

SPR Name	SPR [0:9]	Decimal	SMT	Length	Read (**mfspr**)			Write (**mtspr**)		
					Prob	Priv	Hyp	Prob	Priv	Hyp
XER	00000 00001	1	Replicated	64						
DSCR (FSCR [61] = 0)	00000 00011	3	Replicated	25	Priv			Priv		
DSCR (FSCR [61] = 0)	00000 00011	3	Replicated	25						
LR	00000 01000	8	Replicated	64						
CTR	00000 01001	9	Replicated	64						
AMR	00000 01101	13	Replicated	64						
DSCR (HFSCR[61]=0)	00000 10001	17	Replicated	25	Priv	Priv		Priv	Priv	
DSCR (HFSCR[61]=1)	00000 10001	17	Replicated	25	Priv	Priv		Priv	Priv	
DSISR	00000 10010	18	Replicated	32	Priv			Priv		
DAR	00000 10011	19	Replicated	64	Priv			Priv		
DEC	00000 10110	22	Replicated	32	Priv			Priv		
SDR1	00000 11001	25	Per LPAR	64	Priv	Priv		Priv	Priv	
SRR0	00000 11010	26	Replicated	64						
SRR1	00000 11011	27	Replicated	64						
CFAR	00000 11100	28	Replicated	64						
AMR	00000 11101	29	Replicated	64						
ACOP	00000 11111	31	Replicated	64						
PID	00001 10000	48	Replicated	32	Priv			Priv		
IAMR	00001 11101	61	Replicated	32	Priv			Priv		
TFHAR	00100 00000	128	Replicated	64						
TFIAR	00100 00001	129	Replicated	64						
TEXASR	00100 00010	130	Replicated	64						
TEXASRU	00100 00011	131	Replicated	32						
CTRL	00100 01000	136	Shared	32	bit 63 - Priv	bit 63 - Priv		Illegal	NOP	NOP
APSCR	00100 01010	138	Replicated	64						
APSCRU	00100 01011	139	Replicated	32						
CTRL	00100 11000	152	Shared	32	Priv	bit 63 - Priv		Illegal	NOP	NOP
FSCR	00100 11001	153	Replicated	64	Priv			Priv		
UAMOR	00100 11101	157	Replicated	64	Priv			Priv		
PSPB	00100 11111	159	Replicated	32	Priv			Priv		
DPDES	00101 10000	176	Per LPAR	8	Priv	Priv		Priv	Priv	

Table 3-10. SPR Table (Sheet 2 of 5)

SPR Name	SPR [0:9]	Decimal	SMT	Length	Read (**mfspr**)			Write (**mtspr**)		
					Prob	Priv	Hyp	Prob	Priv	Hyp
DHDES	00101 10001	177	Shared	8	Priv	Priv		Priv	Priv	
DAWR0	00101 10100	180	Replicated	64	Priv	Priv		Priv	Priv	
RPR	00101 11010	186	Per LPAR	64	Priv	Priv		Priv	Priv	
CIABR	00101 11011	187	Replicated	64	Priv	Priv		Priv	Priv	
DAWRX0	00101 11100	188	Replicated	32	Priv	Priv		Priv	Priv	
HFSCR	00101 11110	190	Replicated	64	Priv	Priv		Priv	Priv	
VRSAVE	01000 00000	256	Replicated	32						
SPRG3	01000 00011	259	Replicated	64	Illegal	NOP	NOP			
TB	01000 01100	268	Per LPAR	64	Illegal	NOP	NOP			
TBU	01000 01101	269	Per LPAR	64	Illegal	NOP	NOP			
SPRG0	01000 10000	272	Replicated	64	Priv			Priv		
SPRG1	01000 10001	273	Replicated	64	Priv			Priv		
SPRG3	01000 10011	275	Replicated	64	Priv			Priv		
SPRC	01000 10100	276	Replicated	64	Priv	Priv		Priv	Priv	
SPRD	01000 10101	277	N/A	64	Priv	Priv		Priv	Priv	
TBL	01000 11100	284	Per LPAR	32	Illegal	NOP	NOP	Priv	Priv	
TBU	01000 11101	285	Per LPAR	32	Illegal	NOP	NOP	Priv	Priv	
TBU40	01000 11110	286	Per LPAR	64	Illegal	NOP	NOP	Priv	Priv	
PVR	01000 11111	287	Shared	32	Illegal	NOP	NOP	Priv		
HSPRG0	01001 10000	304	Replicated	64	Priv	Priv		Priv	Priv	
HSPRG1	01001 10001	305	Replicated	64	Priv	Priv		Priv	Priv	
HDSISR	01001 10010	306	Replicated	32	Priv	Priv		Priv	Priv	
HDAR	01001 10011	307	Replicated	64	Priv	Priv		Priv	Priv	
SPURR	01001 10100	308	Replicated	64	Priv	Priv		Priv		
PURR	01001 10101	309	Replicated	64	Priv	Priv		Priv		
HDEC	01001 10110	310	Per LPAR	32	Priv	Priv		Priv	Priv	
RMOR	01001 11000	312	Per LPAR	64	Priv	Priv		Priv	Priv	
HRMOR	01001 11001	313	Shared	64	Priv	Priv		Priv	Priv	
HSRR0	01001 11010	314	Replicated	64	Priv	Priv		Priv	Priv	
HSRR1	01001 11011	315	Replicated	64	Priv	Priv		Priv	Priv	
MMCRH	01001 11100	316	Shared	64	Priv	Priv		Priv	Priv	
TFMR	01001 11101	317	Shared/ LPAR bit 26 and 45 replicated	64	Priv	Priv		Priv	Priv	
LPCR	01001 11101	318	Replicated	64	Priv	Priv		Priv	Priv	
LPIDR	01001 11111	319	Per LPAR	64	Priv	Priv		Priv	Priv	

Table 3-10. SPR Table (Sheet 3 of 5)

SPR Name	SPR [0:9]	Decimal	SMT	Length	Read (**mfspr**)			Write (**mtspr**)		
					Prob	Priv	Hyp	Prob	Priv	Hyp
HMER	01010 10000	336	Replicated	64	Priv	Priv		Priv	Priv	
HMEER	01010 10001	337	Shared	64	Priv	Priv		Priv	Priv	
PCR	01010 10010	338	Per LPAR	64	Priv	Priv		Priv	Priv	
HEIR	01010 10011	339	Replicated	32	Priv	Priv		Priv	Priv	
HPMC1	01010 11000	344	Shared	64	Priv	Priv		Priv	Priv	
HPMC2	01010 11001	345	Shared	64	Priv	Priv		Priv	Priv	
HPMC3	01010 11010	346	Shared	64	Priv	Priv		Priv	Priv	
HPMC4	01010 11011	347	Shared	64	Priv	Priv		Priv	Priv	
AMOR	01010 11101	349	Per LPAR	64	Priv	Priv		Priv	Priv	
TIR	01101 11110	446	Replicated	8	Priv			Illegal	NOP	NOP
SIER	11000 00000	768	Replicated	64				Illegal	NOP	NOP
MMCR2	11000 00001	769	Replicated	64				Illegal	NOP	NOP
MMCRA	11000 00010	770	Replicated	32				Illegal	NOP	NOP
PMC1	11000 00011	771	Replicated	32				Illegal	NOP	NOP
PMC2	11000 00100	772	Replicated	32				Illegal	NOP	NOP
PMC3	11000 00101	773	Replicated	32				Illegal	NOP	NOP
PMC4	11000 00110	774	Replicated	32				Illegal	NOP	NOP
PMC5	11000 00111	775	Replicated	32				Illegal	NOP	NOP
PMC6	11000 01000	776	Replicated	32				Illegal	NOP	NOP
MMCR0	11000 01011	779	Replicated	32				Illegal	NOP	NOP
SIAR	11000 01100	780	Replicated	64				Illegal	NOP	NOP
SDAR	11000 01101	781	Replicated	64				Illegal	NOP	NOP
MMCR1	11000 01110	782	Replicated	32				Illegal	NOP	NOP
SIER	11000 10000	784	Replicated	64	Priv			Priv		
MMCR2	11000 10001	785	Replicated	64	Priv			Priv		
MMCRA	11000 10010	786	Replicated	64	Priv			Priv		
PMC1	11000 10011	787	Replicated	32	Priv			Priv		
PMC2	11000 10100	788	Replicated	32	Priv			Priv		
PMC3	11000 10101	789	Replicated	32	Priv			Priv		
PMC4	11000 10110	790	Replicated	32	Priv			Priv		
PMC5	11000 10111	791	Replicated	32	Priv			Priv		
PMC6	11000 11000	792	Replicated	32	Priv			Priv		
MMCR0	11000 11011	795	Replicated	32	Priv			Priv		
SIAR	11000 11100	796	Replicated	64	Priv			Priv		
SDAR	11000 11101	797	Replicated	64	Priv			Priv		
MMCR1	11000 11110	798	Replicated	32	Priv			Priv		

Table 3-10. SPR Table (Sheet 4 of 5)

SPR Name	SPR [0:9]	Decimal	SMT	Length	Read (**mfspr**)			Write (**mtspr**)		
					Prob	Priv	Hyp	Prob	Priv	Hyp
IMC	11000 11111	799	Shared	64	Priv	Priv	Return zeros	Priv	Priv	Condi-tional based on HID1
BESCRS	11001 00000	800	Replicated	64						
BESCRSU	11001 00001	801	Replicated	32						
BESCRR	11001 00010	802	Replicated	64						
BESCRRU	11001 00011	803	Replicated	32						
EBBHR	11001 00100	804	Replicated	64						
EBBRR	11001 00101	805	Replicated	64						
Reserved	11001 01000	808	N/A		NOP	NOP	NOP	NOP	NOP	NOP
Reserved	11001 01001	809	N/A		NOP	NOP	NOP	NOP	NOP	NOP
Reserved	11001 01010	810	N/A		NOP	NOP	NOP	NOP	NOP	NOP
Reserved	11001 01011	811	N/A		NOP	NOP	NOP	NOP	NOP	NOP
TAR	11001 01111	815	Replicated	64						
OS Run Instruction	11010 10000	848	Replicated	64	Priv	Priv		Priv	Priv	
OS Virtualized TimeBase	11010 10001	849	Per LPAR	64	Priv	Priv		Priv	Priv	
LDBAR	11010 10010	850	Per LPAR	64	Priv	Priv		Priv	Priv	
MMCRC	11010 10011	851	Per LPAR	32	Priv	Priv		Priv	Priv	
PMICR	11010 10100	852	Shared	64	Priv	Priv		Priv	Priv	
PMSR	11010 10101	853	Shared	32	Priv	Priv		Priv	Priv	NOP
PMMAR	11010 10110	854	Per LPAR	64	Priv	Priv		Priv	Priv	
Reserved	11010 11011	859	Per Core	64	Priv	Priv		Priv	Priv	
Reserved	11010 11100	860	Per Core	64	Priv	Priv		Priv	Priv	
Reserved	11010 11101	861	Per Core	64	Priv	Priv	NOP	Priv	Priv	
Reserved	11010 11110	862	N/A		Priv	Priv		Priv	Priv	
Reserved	11010 11111	863	Shared	32	Priv	Priv		Priv	Priv	
TRIG0	11011 10000	880	Replicated	64	Priv			Iillegal	NOP	NOP
TRIG1	11011 10001	881	Replicated	64	Priv			Iillegal	NOP	NOP
TRIG2	11011 10010	882	Replicated	64	Priv			Iillegal	NOP	NOP
PMCR	11011 10100	884	Per LPAR	64	Priv	Priv		Priv	Priv	
RWMR	11011 10101	885	Per LPAR	64	Priv	Priv		Priv	Priv	
TACR	11011 11000	888	Shared	64	Priv			Priv		
TCSCR	11011 11001	889	Shared	64	Priv			Priv		
SPMC1	11011 11100	892	Replicated	32	Priv			Priv		
SPMC2	11011 11101	893	Replicated	32	Priv			Priv		

Table 3-10. SPR Table (Sheet 5 of 5)

SPR Name	SPR [0:9]	Decimal	SMT	Length	Read (**mfspr**)			Write (**mtspr**)		
					Prob	Priv	Hyp	Prob	Priv	Hyp
MMCRS	11011 11110	894	Replicated	32	Priv			Priv		
Reserved	11011 11111	895	Replicated	32	Priv			Priv		
PPR	11100 00000	896	Replicated	64						
PPR32	11100 00010	898	Replicated	32						
TSR	11100 00001	897	Replicated	64	Priv	Priv		Priv	Priv	
TSCR	11100 11001	921	Shared	32	Priv	Priv		Priv	Priv	
TTR	11100 11010	922	Shared	64	Priv	Priv		Priv	Priv	
TRACE	11111 01110	1006	Shared	64	Illegal	NOP	NOP	Priv	Priv	
HID0	11111 10000	1008	Shared	64	Priv	Priv		Priv	Priv	
HID1	11111 10001	1009	Shared (1bit replicated for lpar)	64	Priv	Priv		Priv	Priv	
HID4	11111 10100	1012	Shared (some replicated for lpar)	64	Priv	Priv		Priv	Priv	
HID5	11111 10110	1014	Shared (hid4 spill over repli-cated per lpar)	64	Priv	Priv		Priv	Priv	
CIR	11111 11110	1022	Shared	32	Priv			Illegal	NOP	NOP
PIR	11111 11111	1023	Replicated	32	Priv			Priv	NOP	NOP
Unsupported POWER MQ	00000 00000	0	N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Unsupported Power Real Time Upper SPR	00000 00100	4	N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Unsupported Power Real Time Lower SPR	00000 00101	5	N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Unsupported Power Decrementer SPR	00000 00110	6	N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Unsupported non-privileged, non-Power SPR	xxxxx 0xxxx		N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal
Unsupported-privileged SPR	xxxxx 1xxxx		N/A	N/A	Illegal	Illegal	Illegal	Illegal	Illegal	Illegal

3.8 HID Registers (HID0, HID1, HID4, and HID5)

The POWER8 processor core includes many implementation-dependent mode bits that allow various features of the chip to be enabled and disabled. These bits are included in the Hardware Implementation Dependent registers (HID0, HID1, HID4, and HID5). In general, the HID0 Register controls high-level functions of the POWER8 core. The HID1 Register contains additional mode bits that are related to the instruction fetch and instruction decode functions in the POWER8 core. The HID4 and HID5 Registers contain bits related to the load-store function in the POWER8 processor. All of these registers are only accessible in hypervisor mode. Reserved bits in the HID registers should not be set by software and may return either a zero or one value depending on the bit if set. Attempts to set some of these bits may enable functions that are no longer supported and thus could cause unpredictable behavior. Two values of the contents of each register are shown in the descriptions.

Initial state:
This is the state of the register after a normal scan-based POR. The actual and full POR sequence can set bits beyond the scan-based POR.

Preferred state:
This is the preferred state of the register for optimal performance and function.

1. The following sequence must be used when modifying HID0:
   sync
   mtspr HID0,Rx
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   mfspr Rx,HID0
   isync

   After modifying HID0, executing six **mfspr** instructions specifying HID0 as the source and specifying the same target GPR (Rx) in all six instructions is necessary to ensure that the modification is effective and the processor is in a valid state to continue executing subsequent instructions

2. The following sequence must be used when modifying HID1:
   mtspr HID1,Rx
   mtspr HID1,Rx
   isync

   Executing two **mtspr** instructions is necessary to ensure that updates to all portions of HID1 are completed before the **isync** instruction is completed.

3. The following sequence must be used when modifying HID4:
   sync
   mtspr HID4,Rx
   isync

   Because the ERATs contain entries for nonhypervisor real mode translations, when changing the RMOR value, the hypervisor must flush the ERATs (that is, by executing **slbia**) before passing control to the operating system. The en_rmsc (HID4[33]) bit must not be changed by the hypervisor, and its initial value must be included in the init file.

3.8.1 HID0 Register

Initial state: x'0005_0000_0000_0000'
Preferred state: x'0005_0000_0000_0000'

The HID0 Register is defined as follows.

Bit	Name	Description
0:1	reserved	Reserved.
2	4_lpar_mode	Core runs in the 4 LPAR mode where the core is partitioned into four logical partitions (sub-processors). Has precedence over 2_lpar_mode (bit 6). This is a read-only bit.
3:5	reserved	Reserved.
6	2_lpar_mode	Core runs in the 2 LPAR mode where core is partitioned into two logical partitions (sub-processors). This is a read-only bit.
7:10	reserved	Reserved.
11	1LPARto2LPAR	A write to this bit initiates a 1 LPAR to 2 LPAR dynamic mode switch. Cannot read this bit.
12	1LPARto4LPAR	A write to this bit initiates a 1 LPAR to 4 LPAR dynamic mode switch. Cannot read this bit.
13	dis_recovery	Disable processor recovery mechanism.
14	reserved	Reserved.
15	dyn_lpar_dis	Dynamic LPAR switching mode disabled.
16:30	reserved	Reserved.
31	en_attn	Enable support processor **attn** instruction.
32:63	reserved	Reserved.

3.8.2 HID1 Register

The HID1 Register contains additional mode bits that are related to the instruction fetch and instruction decode functions in the POWER8 processor core.

Initial state: x'0000_0000_0000_0000'
Preferred state: x'0000_0000_0000_0000'

The HID1 Register is defined as follows.

Bits	Field Name	Description
0	flush_ic	Flush the I-cache directory and the IEADIR on a transition from '0' to '1'.
1:4	reserved	Reserved.
5	dis_sp_itw	Disable speculative table walks. If this bit is set to a '1', ERAT miss handling is only carried out when the instruction with the ERAT miss is next-to-complete.
6:7	ierat_lru_mode(0:1)	I-ERAT size restrictor. These HID1 bits can be used to limit the number of I-ERAT entries to use. 00 Full I-ERAT (64 entries) - normal settings 01 Half I-ERAT (32 entries) 10 Quarter I-ERAT (16 entries) 11 Eighth I-ERAT (8 entries)
8	reserved	Reserved.
9	dis_dfp	Disable DFP. If this bit is set to '1,' the DFP instructions are treated as illegals.
10	reserved	Should always be set to '0'.
11	dis_user_priority_ verylow_lpar0	Disable user priority change to very low - LPAR0. 0 Allow user mode to set thread priority to very-low. 1 Do not allow user mode to set thread priority to very low.
12	dis_user_priority_ verylow_lpar1	Disable user priority change to very low - LPAR1 0 Allow user mode to set thread priority to very-low. 1 Do not allow user mode to set thread priority to very low.
13	dis_user_priority_ verylow_lpar2	Disable user priority change to very low - LPAR2 0 Allow user mode to set thread priority to very-low. 1 Do not allow user mode to set thread priority to very low
14	dis_user_priority_ verylow_lpar3	Disable user priority change to very low - LPAR3 0 Allow user mode to set thread priority to very-low. 1 Do not allow user mode to set thread priority to very low.
15	en_reduce_spec	Enable the reduce speculation mode in the IFU. This mode is used to reduce bandwidth beyond the L3 cache.
16:20	reserved	Reserved.
21:31	spare	Spare. These bits are implemented and thus a move-from returns the value of the last move-to.
32:63	Unimplemented spare	Unimplemented spares. These bits always return zeros.

3.8.3 HID4 Register

The HID4 Register controls the load-store functions in the POWER8 core.

Initial state: x'0000_0000_0000_0000'
Preferred state: x'0000_0004_4000_0000'

Bits	Field Name	Description
0:4	rmsc	Real mode storage control field used exclusively for "Legacy RMSC" mode (HID4[33]='0'). If this field is set to a value N, the first $256 \times 2^{(N-1)}$ MB, for 0 < N < 18, of real memory is considered well-behaved, and real memory accesses in this range are treated as nonguarded in MSR[IR, DR] = '0' mode. Beyond this range, accesses are treated as guarded for MSR[IR, DR] = '0' mode. '00000' = all MSR[IR, DR] = '0' accesses are treated as guarded. '00001' = MSR[IR, DR] = '0' accesses in the first 256 MB are nonguarded and beyond that guarded. '00010' = MSR[IR, DR] = '0' accesses in the first 512 MB are nonguarded and beyond that guarded. ..... '10000' = MSR[IR, DR] = '0' accesses in the first 8 TB are nonguarded and beyond that guarded. '10001' = MSR[IR, DR] = '0' accesses in the first 16 TB are nonguarded and beyond that guarded. >'10001' = all MSR[IR, DR] = '0' accesses are treated as guarded. An **slbia** instruction is needed after changing this bit.
5:6	reserved	Reserved.
7	dis_lpage	Disable large page support (only support 4 KB page).
8	dis_mpss	Disable mixed page segment support (4 KB and 64 KB, 4 KB and 16 MB, 64 KB and 16 MB).
9	dis_mpssx	Disable mixed page segment support extensions (4 KB and 16 MB, 64 KB and 16 MB).
10	dis_tspec	Disable speculative load tablewalks (stores are always nonspeculative).
11	dis_vpck	Disable virtual-page class keys exception.
12	en_hash2	Enable secondary hash tablewalks for speculative store instructions (if LPCR[TC] = '0').
13	tlbie_cc	Congruence class invalidation of TLB due to **tlbie** instruction. '0' Use the IS field of the **tlbie** instruction. '1' Override the IS field, and always invalidate the entire congruence class.
14	force_geq1	Force all load instructions to be treated as if they were to guarded (G = 1) storage.
15	nop_dcbt	**dcbt** is treated as a NOP (go through address translation, suppressed from going to Nest).
16	nop_dcbtst	**dcbtst** is treated as a NOP (go through address translation, suppressed from going to Nest).
17	reserved	Reserved.
18	rst_prf	Reset the data prefetch mechanism. Suppress subsequent prefetch requests and clear the stream detection logic, such that stream detection will not be affected by accesses performed before setting the bit back to '0'
19	sus_prf	Suspend the data prefetch mechanism Preserve the current state, but disable allocations, updates, and requests until the bit is set back to '0'.
20	dis_prf1	Disable L1 data prefetching.
21	dis_prf2a	Disable L1 data prefetching two lines ahead (only prefetch one line ahead, pertains to all streams).
22	dis_prf3l	Disable L3 load data prefetching.
23	dis_prf3s	Disable L3 store data prefetching.
24	dis_prfh	Disable data prefetching initiated by hardware (allow software streams).

Bits	Field Name	Description
25	dis_prfnh	Disable stride-n data prefetching initiated by hardware.
26	dis_prfns	Disable stride-n data prefetching initiated by software.
27	dis_shad	Disable data prefetch shadow register. 0 Prefetch stream direction always assumed 'up', but the PRQ can detect a confirmation in the 'down' direction. 1 Prefetch stream direction assumed 'up' if the address is in upper 3/4 of cache line, otherwise, down.
28	lmt_prf	Limit the number of available streams to four.
29	ex_dcbz	Exclude **dcbz** from initiating L3 store prefetches.
30:32	reserved	Reserved.
33	en_rmsc	Enable "Page-based RMSC mode" described in Power ISA Book III-S as "history blocks" implementation. **Note:** Setting this bit to '0' disables the page-based RMSC mode and enables the legacy RMSC mode, which divides real memory into a lower unguarded regions and an upper guarded region. The Legacy mode may not be supported in future POWER processors.
34	reserved	Reserved.
35	reserved	Reserved.
36	dis_rep_lp0	Control bit to disable the PTE time base and reference bit array updates for LPAR 0 0 Reference bit array update disabled. 1 Reference bit array update enabled.
37:39	sel_tb_lp0	Encode bits to select TB bit that provides low-order 12-bit TB value for LPAR 0 000 Reserved. 001 Select 0.5 second. 010 Select 1 second. 011 Select 2 seconds. 100 Select 4 seconds. 101 Select 8 seconds. 110 Select 16 seconds. 111 Select 32 seconds.
40	sel_pte_lp0	1 HCA (Hot / Cold Page Affinity) PTE format in use for LPAR 0. 0 Power ISA PTE format.
41	dis_rep_lp1	Control bit to disable PTE time base and ref-bit array updates for LPAR 1 0 Reference bit array update disabled. 1 Reference bit array update enabled.
42:44	sel_tb_lp1	Encode bits to select TB bit that provides low-order 12-bit TB value for LPAR 1 000 Reserved. 001 Select 0.5 second. 010 Select 1 second. 011 Select 2 seconds. 100 Select 4 seconds. 101 Select 8 seconds. 110 Select 16 seconds. 111 Select 32 seconds.
45	sel_pte_lp1	1 HCA (Hot / Cold Page Affinity) PTE format in use for LPAR 1. 0 Power ISA PTE format.
46	dis_rep_lp2	Control bit to disable PTE time base and reference bit array updates LPAR 2. 0 Reference bit array update disabled. 1 Reference bit array update enabled.

Bits	Field Name	Description
47:49	sel_tb_lp2	Encode bits to select TB bit that provides low-order 12-bit TB value LPAR 2. 000 Reserved. 001 Select 0.5 second. 010 Select 1 second. 011 Select 2 seconds. 100 Select 4 seconds. 101 Select 8 seconds. 110 Select 16 seconds. 111 Select 32 seconds.
50	sel_pte_lp2	1 HCA (Hot / Cold Page Affinity) PTE format in use for LPAR 2. 0 Power ISA PTE format.
51	dis_rep_lp3	Control bit to disable PTE time base and ref-bit array updates LPAR 3 0 Reference bit array update disabled. 1 Reference bit array update enabled. **Note:** In single LPAR mode, all four partitions (en_rep_lpx and sel_tb_lpx and sle_pte_lpx) must be set to the same value. In 2 LPAR mode: en_rep_lp[0,1] and sel_tb_lp[0,1] and sel_pte_lp[0,1] must be set to the same value; and en_rep_lp[2,3] and sel_tb_lp[2,3] and sel_pte_lp[2,3] must be et to the same value.
52:54	sel_tb_lp3	Encode bits to select TB bit that provides low-order 12-bit TB value LPAR 3. 000 Reserved. 001 Select 0.5 second. 010 Select 1 second. 011 Select 2 seconds. 100 Select 4 seconds. 101 Select 8 seconds. 110 Select 16 seconds. 111 Select 32 seconds.
55	sel_pte_lp3	1 HCA (Hot / Cold Page Affinity) PTE format in use for LPAR 0. 0 Power ISA PTE format.
56:63	reserved	Reserved.

3.8.4 HID5 Register

Initial state: x'0000_0000_0000_0000'
Preferred state: x'4000_0001_0000_0000'

The HID5 Register is defined as follows.

Bits	Field Name	Description
0:12	reserved	Reserved.
13:14	Throt_Dpf	Hypervisor memory bandwidth control for data prefetch. 00 Medium 01 Low 10 High 11 Reserved
15	reserved	Reserved.
16:29	reserved	Reserved.
30:31	pte_plus_n	PTE Plus n prefetching where n = 1 or 3. 00 Disable 10 PTE + 1 01 PTE + 3 11 Undefined
32:63	reserved	Reserved.

3.8.5 Real Mode Offset (RMO) Region Sizes

The following RMO sizes are available for the POWER8 processor. The RMLS[34:37] field in the LPCR defines the RMO sizes, as described below.

1000 - 32 MB
0011 - 64 MB
0111 - 128 MB
0100 - 256 MB
0010 - 1 GB
0001 - 16 GB
0000 - 256 GB

3.8.6 Hypervisor Real Mode Offset (HRMO) Register Update Sequence

Table 3-11. HRMOR Update Sequence

Master	Slave
Thread sync up point 1	Thread sync up point 1
Change HID4[0:4] = 0000 (default in POWER8)	
EA[0] = 1	EA[0] = 1
Thread sync up point 2	Thread sync up point 2
Change HRMOR	
Thread sync up point 3	Thread sync up point 3
isync	isync
slbia0	slbia0
isync	isync
Thread sync up point 4	Thread sync up point 4

3.8.7 Core-to-Core Trace SPR

The Trace SPR is used to access enhanced instruction trace information from the processor core trace logic. This 64-bit register is read only and has a privileged read access. There is a protocol associated with the use of this register to coordinate gathering instruction trace images from the other processor core.

3.8.8 Trigger Registers

Writes to the trigger registers, named TRIG0, TRIG1, and TRIG2, can be inserted in the instruction stream to cause triggers to the on-chip trace array debug logic. These are intended to be used for lab debug and bringup only and architecturally behave as a NOP.

3.8.9 IMC Array Access Register

The Instruction Match CAM (IMC) array facility is used for performance monitoring instrumentation and for the soft patch of instructions (this latter use is restricted for the support processor and is not available through the SPR access to this register array). The array has privileged write access and user-level read access via this SPR. Writes to the register array are used to configure the IMC, and reads return information about the availability of registers within the facility.

3.8.10 Performance Monitor Registers

The performance monitor counter registers (PMC1 - PMC6), the performance monitor control registers (MMCR0, MMCR1, MMCRA), and the sampled address registers (SIAR, SDAR) are supported in the POWER8 processor core. The performance monitor counter registers PMC7 and PMC8 are not implemented in the POWER8 processor core (an operation for these two performance counter registers will be treated as NOP).

3.8.11 Other Fixed-Point Instructions

The POWER8 processor core supports both the 32-bit **mtmsr** instruction and the 64-bit **mtmsrd** instruction.

The POWER8 processor core works to optimize the **mtmsr** instruction to help speed up the cases where little or no synchronization is required (such as, updates to the EE bit).

Software must avoid placing **mtmsr** and **mtmsrd** instructions that change the SF bit at address x'00000000FFFFFFFC' or x'FFFFFFFFFFFFFFFC'.

3.9 Storage Control

3.9.1 Virtual and Physical Address Ranges Supported

The POWER8 processor core supports a 68-bit virtual address and a 50-bit physical (real) address.

3.9.2 Data Effective-to-Real-Address Translation (D-ERAT)

The POWER8 processor core includes a primary ERAT and secondary ERAT. The primary ERAT has four copies of 48 entries. It is a fully-associative D-ERAT, each with binary LRU replacement policy (separate LRU for each half in SMT2/4/8 mode), data effective-to-real address translation (D-ERAT) cache for fast translation of data effective addresses into physical (real) addresses.

In the POWER8 core, each entry of the D-ERAT contains translation information for a 4 KB, 64 KB, or 16 MB block of effective storage, depending on the page size being used. Reference to a 1 MB or 16 GB page uses 64 KB or 16 MB entries in the D-ERAT, respectively.

In ST mode, 48 entries are available, and all four copies have the same content. In SMT2, SMT4, and SMT8 mode, 48 entries are available for threads on LS half 0, and 48 entries for LS half 1. The two ERATs on a half have the same contents.

D-ERAT entries are created for real mode (MSR[DR] = '0') and are shared by all threads in the same partition. Hypervisor real-mode entries are shared by all threads. Entries are invalidated via a CAM to clear the function in support of **slbia**, **slbie**, **tlbie**, and **mtsr** instructions. It is parity and multi-hit protected, reports to the FIR and completion on an error detect, hardware recovery via ABIST, and multi-hit is reported as a machine check interrupt.

The secondary D-ERAT is the second-level translation cache for data addresses. It has 256 entries managed in two 128-entry halves. It is a fully associative with FIFO replacement policy. In ST mode, 256 entries are available. In SMT2, SMT4, and SMT8 mode, 128 entries are available for threads on LS half 0, and 128 entries for LS half 1.

According to the Power ISA, aliasing the I-bit storage attribute is prohibited. In the POWER8 core, due to the caching of pages in the ERATs, software should avoid accessing the same real page with different values for the I-bit storage attribute. Failure to follow this restriction may result in a cache paradox or other boundedly undefined behavior.

D-ERAT I and G Bit Setting

The G and I bits are set based on *Table 3-12*.

Table 3-12. D-ERAT I and G Bit Setting (Sheet 1 of 2)

Condition								Resulting Action
MSR [DR]	MSR [HV]	LPCR[0]	HID4[33]	First access HV CI Instruction	HID4[0:4] RMSC (above line)	I	G	
1	x	x	x	x	x	PTE(I)	PTE(G)	An entry is created with the I and G values set from the PTE.
0	0	1	x	x	x	PTE(I)	PTE(G)	Virtual real mode. An entry is created with the I and G values set from the PTE.
0	0	0	x	x	x	0	0	RMOR mode. Set IG = '00'
0	1	x	0	yes	no	1	0	Legacy RMSC mode. If the first access is caused by a hypervisor CI load or store (for example, **ldcix**, **stdcix**, and so on) is below the guarded line established by HID4[0:4], storage is G = '0' and CI load/store access causes a DSI and set DSISR(62) = '1'.
0	1	x	0	no	no	0	0	Legacy RMSC mode. If the first access is caused by any instruction other than a hypervisor CI load or store is below the guarded line established by HID4[0:4], storage is G = '0' and an entry is established as I = '0' and G = '0'.
0	1	x	0	no	yes	0	1	Legacy RMSC mode. If the first access is caused by any instruction other than a hypervisor CI load or store is above the guarded line established by HID4[0:4], storage is G = '1' and an entry is established as I = '0' and G = '1'.

Table 3-12. D-ERAT I and G Bit Setting (Sheet 2 of 2)

Condition								Resulting Action
MSR [DR]	MSR [HV]	LPCR[0]	HID4[33]	First access HV CI Instruction	HID4[0:4] RMSC (above line)	I	G	
0	1	x	0	yes	yes	0	1	Legacy RMSC mode. If the first access is caused by a hypervisor CI load or store (for example, **ldcix**, **stdcix**, and so on) is above the guarded line established by HID4[0:4], storage is G = '1' and an entry is established as I = '1' and G = '1'.
0	1	x	1	yes	x	1	1	Page-based RMSC mode. If the first access is caused by a hypervisor CI load or store (for example, **ldcix**, **stdcix**, and so on), an entry is established as I = '1' and G = '1'.
0	1	x	1	no	x	0	0	Page-based RMSC mode. If the first access is caused by any instruction other than a hypervisor CI load or store, storage is G = '0' and an entry is established as I = '0' and G = '0'.

In addition to the conditions shown in *Table 3-12*, the G bit is forced to a '1' for the following cases:

- HID4[33] = 0, HV = 1, and the referenced address is greater that 16 TB.
- HID4[33] = 0, HV = 1, and the RMSC value is '00000' or greater than '10001' (Illegal value).
- HV = 1, PR = 0, EA(0) = 1, and EA(14 to 33) = '11111111111111111'.

The I bit is forced to a '1' for the following case:

- HV = 1, PR = 0, EA(0) = 1, and EA(14 to 33) = '11111111111111111'.

Caching-Inhibited Paradox Cases

If a caching-inhibited load instruction hits in the L1 data cache, the load data is serviced from the L1 data cache and no request is sent to the NCU.

If a caching-inhibited store instruction hits in the L1 data cache, the store data is written to the L1 data cache and sent to the NCU. Note that the L1 data cache and L2 cache are no longer coherent.

These scenarios are true both for instructions marked as caching-inhibited by the PTE I-bit and for the hypervisor caching-inhibited load and store instructions.

The POWER8 core supports the page-based real-mode storage control (RMSC) mechanism that allows speculative access to DR = 0 space, if there is real memory there.

Because the content of each D-ERAT entry is the result of a page table search based on the contents of an SLB entry, to maintain consistency with the SLB (or segment registers), the following instructions will cause all entries that belong to the thread in the D-ERAT to be invalidated (these instruction do not invalidate D-ERAT entries of the other thread).

- **mtsr** or **mtsrin** instructions - used for segment register changes in 32-bit operating systems
- **slbia**

The **slbie** instruction causes invalidation of a D-ERAT entry belonging to the thread (no impact to the other thread) only if there is a perfect address match: EA[0:35] are matched for **slbie** for a small (256 MB) segment and EA[0:23] are matched for **slbie** for a big (1 TB) segment.

The **tlbie** instruction (or the detection of snooped-tlbie operations) invalidates all D-ERAT entries (irrespective of the thread) in the D-ERAT that have a perfect match. In other words, the entry is invalidated only if:

- EA[36:51] are matched for tlbie small page
- EA[36:47] are matched for tlbie 64 KB page
- EA[36:43] are matched for tlbie 1 MB page
- EA[36:39] are matched for tlbie 16 MB page
- EA[24:29] are matched for tlbie 16 GB page

Upon power-on, each D-ERAT entry is set to the invalid state.

3.9.3 Translation Lookaside Buffer (TLB)

The POWER8 core contains a unified (combined for both instruction and data), 2048-entry, 4-way set-associative TLB (LRU-based replacement algorithm). In addition, the POWER8 core contains one 64-entry, fully-associative I-ERAT (single-level effective-to-real translation) and four 48-entry, fully-associative D-ERATs. The TLB is a cache of recently-used page table entries, and the ERATs are caches that contain translations derived from information in the page table and SLB. The TLB and ERATs are loaded and managed by hardware.

In the POWER8 core, the TLB entry stores the logic partition ID (LPID) in each TLB entry to indicate which partition loaded that TLB entry. Because the virtual and real address space are the same for all software threads within a logical partition, the TLB, which keeps the mapping from virtual-to-real address space, are completely shared by the threads within a partition and there is no thread-ID bit needed in the TLB to identify which entry belongs to which thread. Different partitions have different mappings from virtual-to-real address space, and, because of this, TLB entries cannot be shared between partitions. A given entry in the TLB can be used by all the threads within a partition at the same time (however, a given entry in I-ERAT or D-ERAT, cannot be shared by the two threads at the same time because the ERATs also contains information from the effective to virtual address mapping), unless the entry is created by bypassing the SLB (that is, IR = 0 or DR = 0 addresses). Threads in different partitions are not able to access TLB entries from another partition.

In the POWER8 core, TLB is indexed with the following hashed address (this includes reads from the TLB, writes to the TLB, tlbie instructions, and snooped tlbie transactions).

Table 3-13. 256 MB Segments

Page Size	Index
4 KB	[VSID(45:49) XOR EA(43:47)] || EA(48:51)
64 KB	[VSID(45:49) XOR EA(39:43)] || EA(44:47)
1 MB	[VSID(45:49) XOR EA(36:40)] || EA(40:43)
16 MB	VSID(45:49) || EA(36:39)
16 GB	Does not exist

Table 3-14. 1 TB Segments

Page Size	Index
4 KB	[VSID(33:37) XOR EA(43:47)] || EA(48:51)
64 KB	[VSID(33:37) XOR EA(39:43)] || EA(44:47)
1 MB	[VSID(33:37) XOR EA(35:39)] || EA(40:43)
16 MB	[VSID(33:37) XOR EA(31:35)] || EA(36:39)
16 GB	[VSID(33:37) XOR EA(24:28)] || EA(26:29)

The POWER8 core supports hardware update of the storage access recording bits (reference and change) into the memory-based page table.

The Power ISA has been enhanced to add an L bit, to indicate large page in the page table entry (PTE). This bit is compared with the L bit of the SLB in translation mode for a hit during a table walk to reload the TLB on a TLB miss.

The POWER8 core supports TLB hit under miss and four table concurrent table walks. The POWER8 core also supports two outstanding I-ERAT misses (from the eight threads) and four outstanding D-ERAT misses at the same time.

The POWER8 core supports lockless TLBIE operations.The architectural requirement that only one thread at a time can execute **tlbie**/**tlbsync** instructions during a page table modification (section Page Table Updates of Power ISA Book III) need not be followed. This was traditionally implemented with a single global lock for the entire page table modification sequences. The term "lock-less" TLBIEs refers to the POWER8 core's ability to manage concurrent **tlbie**/**tlbsync** sequences from multiple threads without this global lock.

However, software must still ensure that concurrent, conflicting, racing PTE updates from more than one thread do not occur (the hardware performs the updates in some fashion, but the end results is undefined due to the racing of the updates to the same PTE entry) and therefore, software locks or some other synchronization discipline are still required to prevent these collisions as necessary."

The execution of **tlbie** instructions or the detection of **snooped-tlbie** operations off the bus cause an index-based invalidate to occur in the TLB, if there is a match. In other words, an entry is invalidated only if there is a perfect match of the effective address supplied by the tlbie operation and the content of the TLB entry.

The POWER8 core does not support the **tlbia** instruction.

Upon power-on, the POWER8 core initializes each TLB entry to the invalid state.

3.9.4 Large-Page Support

In addition to the normal 4 KB page size, the POWER8 core provides support for 64 KB, 1 MB, 16 MB, and 16 GB pages. Translation information for all these page sizes is kept in the TLB. Irrespective of the page size, a given page takes up only one entry in the TLB.

If a virtual address is mapped into a small (large) page and then later on mapped into a large (small) page without invalidating TLB entries between changing page size, a machine check interrupt can result with an indication that a parity error occurred when the TLB was accessed to translate an effective address. The error condition can be corrected by invalidating the entire TLB and SLB.

POWER8 also supports Multiple Page Sizes Per Segment (MPSS) as described in the Power ISA. Specifically, POWER8 supports mixing page size in a single segment with the following combinations only:

4K base / 64K actual
4K base / 16M actual
64K base / 16M actual

Table 3-15 shows the correspondence between PTE[L LP] values and SLBE[L LP] values.

Table 3-15. PTE and SLBE Correspondence

Entry Number	PTE			SLBE		Base Page Size	Actual Virtual Page Size	Notes
	L	LP		L	LP			
1	0	rrrr	rrrr	0	00	4 KB	4 KB	1
2	1	0000	0000	1	00	16 MB	16 MB	
3	1	rrrr	0001	1	01	64 KB	64 KB	2
4	1	0000	0010	1	11	1 MB	1 MB	
5	1	0000	0011	1	10	16 GB	16 GB	
6	1	rrrr	0111	0	00	4 KB	64 KB	1
7	1	0000	1000	1	01	64 KB	16 MB	2
8	1	0011	1000	0	00	4 KB	16 MB	1

1. Entries 1, 6, and 8 all use SLB[L LP] = 000 encoding for base page size 4 KB but have unique PTE[L LP] encodings for actual page size.
2. Entries 3 and 7 both use SLB[L LP] = 101 encoding for base page size of 64 KB but have unique PTE[L LP] encodings for actual page size.
3. Unimplemented SLB page size encodings are treated the same as the '000' case.
4. If the SLB page size is '110' (16 GB) and the segment size is small (256 MB), hardware treats the SLB page size the same as the '000' case.
5. The 'r' bits are part of the real page number for actual pages sizes less than 1 MB. They can be any value.

3.9.5 PTE Prefetching

The POWER8 core supports the prefetching of PTE entries. This feature is enabled by setting the values in HID5[30:31] as follows:

00 PTE prefetching is disabled.

01 PTE prefetching prefetches three additional PTE entries.

10 PTE prefetching prefetches one additional PTE.

11 Undefined.

This PTE prefetch feature works as follows. When a tablewalk is initiated to bring in a new PTE entry, the VPN of that tablewalk request is incremented by one and a request is sent to bring the PTE group associated with the incremented VPN into the L3 cache. Only the PTE groups associated with primary table walks are requested. The VPN is only incremented in the lower 2 bits. If the incremented VPN would have crossed this boundary, the lower two bits are set to '0'. If PTE prefetching is configured for prefetching three additional PTE groups and the boundary is crossed, the prefetching logic continues to increment the lower 2 bits until all three prefetches are sent.

3.9.6 Segment Lookaside Buffer (SLB)

The POWER8 core contains a unified (combined for both instruction and data), 32-entry, fully associative SLB per thread (eight per processor core). Information derived from the SLB can also be cached in the I-ERAT or the D-ERAT along with information from the TLB. As a result, many of the SLB management instructions have effects on the ERATs as well as on the SLB itself. The POWER8 core supports the 1 TB segment size, in addition to the usual 256 MB segments. Bit 0 of the SLB[B] field is ignored by POWER8 and should be always set to '0'b per the Power ISA for unimplemented segment size encodings.

Because the SLB is managed by software (the operating system), it is possible that multiple entries can be incorrectly set up to provide translations for the same effective address. If an effective address is translated by more than one SLB entry (that is, the ESID fields of the entries are identical), a machine check interrupt results with an indication that a parity error occurred when the SLB was accessed. When this happens the hardware logically OR's the data in the conflicting entries. The machine check handler can look at the SLB contents to try to determine if conflicting entries have been provided. When a parity error occurs not due to multiple entries, the entire SLB must be reloaded because the DAR does not contain an address indicating which entry caused the parity error. If the source of the error was due to multiple entries, the conflicting entries must be corrected for the translation to proceed, which might also be accomplished by reloading the entire SLB with good entries.

3.9.7 Address Space Register

The Address Space Register (ASR) has been removed from the Power ISA.

3.9.8 Support for 32-Bit Operating Systems

The POWER8 core supports the optional bridge facilities and instructions for 64-bit implementations described in the Bridge to SLB Architecture section of the *Power ISA Book III-S version 2.07*.

Associated with this support, the following optional instructions are supported:

- **mtsr** - Move to segment register
- **mtsrin** - Move to segment register indirect
- **mfsr** - Move from segment register
- **mfsrin** - Move from segment register indirect
- **mtmsr** - Move to machine state register (32-bit)

3.9.9 Reference and Change Bits

The POWER8 hardware performs *reference* and *change* bit updates to the page table.

The W and M bits in the PTE are assumed to be '01' respectively. If the Change bit is updated, the W and M bit in the PTE are set to '01' respectively.

The POWER8 core provides a mode bit for determining whether or not speculative load instructions should reload the TLB in the event of a miss (HID4[10]). If this mode bit is set to allow this behavior, the reference bit can be set on behalf of speculative loads (that is, ones that never actually complete from the programs perspective). Under no circumstances will the POWER8 processor core speculatively set the page table change bit.

In the POWER8 core, load instructions that have a TLB miss but are denied access by storage protection, cause a DSI, cause a DAWR DSI, cause an I = 1 DSI, or cause an alignment interrupt that still causes the reference bit for the subject page to be set. Similarly, store type instructions that cause a DSI, cause a DAWR DSI, cause an I = 1 DSI, or cause an alignment interrupt, set the change bit. On the other hand, the change bit is not set, if a store instruction is denied access by page protection exception.

3.9.10 Storage Protection

The architecture defines whether the instruction fetch is permitted from a page marked "no access" as implementation dependent. In the POWER8 processor core, these instruction fetches are permitted to continue without signaling an exception. The POWER8 core supports storage protection modes in the Power ISA, with 32 virtual-page class keys. AMOR and UAMOR are implemented as 32-bit SPRs.

3.9.11 Block Address Translation

Although this facility existed in earlier versions of the architecture, it is no longer part of the Power ISA. As a result, the POWER8 core does not support block address translation.

3.9.12 Real Mode Storage Control

The POWER8 core supports the ability to control cacheability of data and instruction accesses while in real mode.

For the D-side and I-side, in HV = 1, the translation is loaded with G = 0 and 16 MB page size.

The POWER8 core supports RMSC for data storage and instruction storage. The legacy RMSC behavior is also available under a HID4 bit control. The real memory in a system is often noncontiguous and the hypervisor data and instruction storage accesses can be scattered across the address space. The page-based RMSC architecture and implementation allows speculative access safely in system memory. The first time the access is made in DR = 0 and IR = 0 mode, it is done nonspeculatively. After the first access, a proper D-ERAT entry and I-ERAT entry is established. Subsequent accesses to such a D-ERAT and I-ERAT entry ensure that the access is made to system memory and therefore, can be done speculatively, providing higher performance.

To change to or from the legacy RMSC behavior, an **slbia** is needed after the HID4 Register update.

3.9.13 Storage Access Modes - WIMG Bits

The POWER8 core always assumes W = 0 and M = 1 independent of the value of these bits in the page table. Furthermore, when the hardware is performing a change bit update, it writes the W and M bits as W = 0 and M = 1. Per the Power ISA, accessing a page as both I = 0 and I = 1 is boundedly undefined. Software should avoid aliasing the I-bit on a page basis.

Table 3-16 summarizes the treatment of the WIMG bits in the POWER8 core.

Table 3-16. WIMG Bits

WIMG	Description
x1xx (except 1110)	Treated as WIMG = 0111
x0x1	Treated as WIMG = 0011
x0x0	Treated as WIMG = 0010
1110	Treated as WIMG = 0010 and accesses are strongly ordered

For the noncacheable unit (NCU), the IG combination has the following meaning in the POWER8 core to control the store ordering and store gathering.

Table 3-17. IG Bits

IG	Description
11	No gather, no reorder in NCU is allowed
10	Gather, reorder in NCU is allowed

In IG = '11' mode, cache-inhibited loads cannot be reordered relative to loads, and cache-inhibited stores cannot be reordered relative to cache-inhibited stores. Cache-inhibited loads can be reordered relative to cache-inhibited stores and vice-versa (if it is necessary to maintain ordering between loads and stores, barrier instructions must be used). There is no defined ordering between cache-inhibited load or store operations from different threads.

In IG = '11' mode, gathering is not permitted for either load or store operations within or between threads.

In IG = '10' mode, cache-inhibited loads or stores from a given thread can be gathered and can be reordered. This mode allows for higher performance with a certain loss of control of the order in which the operations are completed or whether operations are gathered (barriers can be used where necessary to re-establish order). There is no defined ordering between cache-inhibited load or store operations from different threads.

3.9.14 Speculative Storage Accesses

The POWER8 processor core can execute load instructions to nonguarded storage speculatively. This can occur when a load instruction is encountered on a predicted branch path, or when a logically preceding instruction causes an interrupt. As a result, it is possible for a speculative load that misses in the on-chip cache hierarchy to initiate an external storage request even if that load instruction is not actually executed as part of the true instruction stream.

3.9.15 mtsr, mtsrin, mfsr, and mfsrin Instructions

Most of the optional Power ISA bridge support for 32-bit operating systems is supported by the POWER8 core. As part of that support, the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions are supported.

3.9.16 TLB Invalidate Entry (tlbie and tlbiel) Instructions

The POWER8 processor core implements the **tlbie** and **tlbiel** instruction described in the architecture. Both of these instructions support small and large pages. The **tlbiel** instruction is never sent outside the processor core.

The **tlbie** (large or small page) instruction performs an index-based (congruence class) invalidate of the TLB (if there is a perfect match of the effective address of the tlbie operation and the content of the TLB entry). Both I-ERAT and D-ERAT entries are invalidated when there is a perfect match.

The **tlbsync** instruction is used to synchronize the completion of the **tlbie** instruction. Only one **tlbsync** instruction is required to synchronize the completion of a group of **tlbie** instructions.

The POWER8 core also uses the appropriate address bits from the TLBIE transaction to index into both ERATs and then invalidates an entry (if any), that matches the effective address.

The POWER8 core also supports two groups of pages TLBIE instructions: one for eight consecutive 4 KB pages TLBIE with AP encode RB[56:58] = '110' and another one for eight consecutive 64 KB pages TLBIE with LP encode RB[46:51] = '001010'.

The **tlbie** is a hypervisor-only instruction. An attempt to execute it while not in hypervisor mode causes a privileged type of program interrupt. In the POWER8 core, the **tlbiel** is a privileged instruction. Any attempt to execute it while in the problem state causes a privileged type of program interrupt.

Table 3-18 shows the legal segment size and page size specifications for **tlbie** and **tlbiel** for the POWER8 core when L = '0'.

*Table 3-18. Segment Size and Page Size Specifications for **tlbie** and **tlbiel** (L = 0)*

RB[49:51]	RB{54:55] Segment Size	RB[63] L	RB[56:58] AP (Same as SLB[L|LP] Encoding)	Actual Page Size to be Invalided
vvv	00	0	000	4 KB
vvv	00	0	101	64 KB
vvv	00	0	100	16 MB
000	00	0	110	8 consecutive 4K pages aligned on 32K boundary
vvv	01	0	000	4 KB
vvv	01	0	101	64 KB
vvv	01	0	100	16 MB
000	01	0	110	8 consecutive 4K pages aligned on 32K boundary

1. All other values must not be used when L = '0'.
2. RB[54:55] = '00' corresponds to a 256 MB segment size and RB[54:55] = '01' corresponds to 1 TB segment size.
3. 16 GB page with a small segment (RB(54:55]= '00') is *not* a permitted combination.

Table 3-19 shows the legal segment size and page size specifications for **tlbie** and **tlbiel** for the POWER8 core when L = '1'.

*Table 3-19. Segment Size and Page Size Specifications for **tlbie** and **tlbiel** (L = 1)*

RB[54:55] Segment Size	RB[63] L	RB[56:58] AP (same as SLB[L|LP] encoding)	Base Page Size	Actual Page Size to be Invalided
00	1	0000 0000	16 MB	16 MB
00	1	VVVV 0001	64 KB	64 KB
00	1	0000 0010	1 MB	1 MB
00	1	0000 1000	64 KB	16 MB
00	1	v000 1010	64KB	8 consecutive 64K pages on 512K boundary
01	0	0000 0000	16 MB	16 MB
01	1	VVVV 0001	64 KB	64 KB
01	1	0000 0010	1 MB	1 MB
01	1	0000 0011	16 GB	16 GB
01	1	0000 1000	64 KB	16 MB
01	1	v000 1010	64 KB	8 consecutive 64K pages on 512K boundary

1. All other values must not be used when L = '1'.
2. 'v' corresponds to AVA (AVPN) bits (and thus can be any value).
3. RB[54:55] = '00' corresponds to 256 MB segment size and RB[54:55] = '01' corresponds to 1 TB segment size.
4. 16 GB page with a small segment (RB[54:55] = '00') is *not* a permitted combination.

3.9.17 TLB Invalidate All (tlbia) Instruction

The **tlbia** instruction is not implemented in the POWER8 core and if detected causes a hypervisor emulation assistance interrupt. The effects of the instruction can easily be emulated by executing a series of **tlbiel** instructions (512 in the POWER8 core, for the 512 congruence classes) by incrementing the effective address bits [43:51] through their full range, and by setting the IS field of the **tlbiel** instruction to the appropriate values as described in the Power ISA. To invalidate all entries irrespective of the LPAR ID, MSR[HV] must equal '1'. A special HID4 bit can be used to force the core to ignore the IS field and always invalidate the entire congruence class.

3.9.18 TLB Synchronize (tlbsync) Instruction

On a given thread, the **tlbsync** instruction forces any previous **tlbie** instructions to complete before the **tlbsync** is allowed to complete. The instruction is implemented as described in the Power ISA.

3.9.19 Page Replacement Policy

The POWER8 core supports an optional PTE format, which must be used when the optional page replacement policy (hot/cold page affinity) feature is enabled by setting HID4 bits: 40 (LPAR0), 45 (LPAR1), 50 (LPAR2), and 55 (LPAR3). A reference-history array in memory contains Reference-bit values for recent sampling periods. On the first TLB miss for a page in the sampling period, the reference-bit array is right shifted (with zero fill) by the number of sample periods since the last TLB miss, and then the high-order bit is set to a one.

The PTE entry has been reorganized to support this function.

- B bits in PTE DW0 bits [0:1] move to PTE DW1 bits [4:5]
- REF_ARRAY enable bit in PTE DW1 bit 6
- 12 bits PTE_UPD_TIME in PTE DW0 bits [0:11]

The LSU examines PTE_UPD_TIME stamp during tablewalk TLB reload. A 12-bit CPU_UPD_TIME is generated from timebase + real address bits to compare against the PTE_UPD_TIME.

HID4[37:39] define sample periods from 0.5 second to 32 second.

- 000 - Reserved
- 001 - 0.5 second
- 010 - 1 second
- 011 - 2 seconds
- 100 - 4 seconds
- 101 - 8 seconds
- 110 - 16 seconds
- 111 - 32 seconds

HID4[36] controls the PTE time base and reference-bit array updates; '1' enables a reference-bit array update and '0' disables a reference-bit array update. The HID4[40] selects new or old PTE format; '1' for new PTE format and '0' for old PTE format.

Timebase bits, TB[18:43], are shadowed in the LSU. TB[35] is the 0.5 second bit and TB[29] is the 32 second bit. The high-order 12-bit CPU_UPD_TIME is generated from the 20-bit adder as follows.

- 0.5 second - TB[24:43] + [000000000000||Real Addr(40:47)]
- 1 second - TB[23:42] + [000000000000||Real Addr(40:47)]
- 2 second - TB[22:41] + [000000000000||Real Addr(40:47)]
- 4 second - TB[21:40] + [000000000000||Real Addr(40:47)]
- 8 second - TB[20:39] + [000000000000||Real Addr(40:47)]
- 16 second - TB[19:38] + [000000000000||Real Addr(40:47)]
- 32 second - TB[18:37] + [000000000000||Real Addr(40:47)]

If CPU_UPD_TIME does not equal PTE_UPT_TIME, the LSU sends the 12-bit CPU_UPD_TIME with PTE address to the nest through the store port to update both the PTE and Reference-bit arrays. The TTYPE = '100000' is for PTE update and TTYPE = '100011' is for Reference-bit array update.

Table 3-20 defines how the Reference-bit array is updated.

Table 3-20. Reference-Bit Array Update

HID4[40] Page Based PTE Format	HID4[36] Enable Reference-Bit Array Function	PTE DW1[6] Enable Reference-Bit Array Update	CPU_UPD_TIME = PTE_UPT_TIME	Reference-Bit Array Update
0	0	X	X	Not Updated
0	1	X	X	Not Updated
1	0	X	X	Not Updated
1	1	0	X	Not Updated
1	1	1	1	Not Updated
1	1	1	0	Updated

3.9.20 Support for Store Gathering

The POWER8 core performs gathering of cacheable stores to reduce the store traffic into the L2 cache. For cacheable stores, the gathering occurs in L2 store queues that sit above the L2 cache. The store queue is shared by the threads. The store queue is comprised of two banks of sixteen 64-byte wide, fully-associative entries or gather stations. Stores can be gathered while architecturally permitted (that is, there is no intervening barrier operation) and the matching address is valid in the store queue. The conditions for pushing the store queue data into the L2 cache are not visible to the programmer.

Gathering of cache-inhibited stores is also supported and can be disabled with a mode bit in the noncacheable unit (NCU) configuration register. There are sixteen 64-byte gather stations in the NCU.

3.9.21 Cache Coherency Paradoxes

Accesses to a given cache line as both cacheable and caching inhibited are not supported in either the Power ISA nor the POWER8 chip. Because the value of the I-bit is cached by the ERATs inside the processor core, cacheable accesses may be performed speculatively and thus, software should avoid alias the I-bit (that is, caching-inhibited bit) on a per page basis. Failure for software to adhere to this restriction can lead to cache corruption.

3.9.22 Handling Parity Error, Multi-Hit, and Uncorrectable Errors

3.9.22.1 Parity Error

If there is a parity error in the D-cache, I-cache, D-ERAT, I-ERAT, TLB or several other register files, SRAM dataflow or control structures (but not the SLB), the POWER8 core sets the relevant FIR bit and initiates the instruction retry and recovery (IRR) process to "clean up" all the architected states and flush the caches, ERATs, and TLB, but keep the SLB as is. Software restores the SLB. After the recovery process, a hypervisor maintenance interrupt (HMI) is generated. On a successful recovery, the HMER indicates a successful recovery.

If the same parity error occurs several times and reaches a threshold, the hypervisor can decide that the core is nonfunctional. The threshold counter is maintained by the hypervisor in software.

HID0[13] must be set to '0', otherwise processor recovery will not work.

Note: The instruction IRR process is engaged for detection of any recoverable parity error in the core or due to the firing of a control checker.

There is a separate FIR bit and FIR extension bits for a parity error in the I-cache, D-cache, SLB, D-ERAT, I-ERAT, TLB, and a few other structures. For all the other register files, there is one shared FIR bit to indicate parity error.

3.9.22.2 Multi-Hit

If there is a multi-hit in the D-ERAT, TLB, or SLB, the core finishes the operation with a machine check interrupt and sets the proper DSISR bit to indicate where the multi-hit was detected.

A multi-hit in the D-ERAT and SLB can occur due to a hardware failure. Multi-hit means more than one entry matched the EA in the D-ERAT (ESID in the case of an SLB). Due to their CAM structure, the result is a "bitters logical or" of the RA of the multiple entries (VSID in case of SLB). Because of this "bit-wise logical or", multi-hit is very likely to generate a parity error as well.

Because the SLB is managed by software with the Power ISA, a software bug can result in a multi-hit in SLB structures. There is no known case of multi-hit in I-ERAT that can produce a wrong result.

There are separate FIR bits for a multi-hit in the D-ERAT, TLB and SLB.

3.9.22.3 Both Multi-Hit and Parity Error

If both multi-hit and parity errors happen in the D-ERAT or TLB, the processor core initiates an instruction retry and recovery (IRR) process. No machine check is presented, but, as usual, after the recovery operation the processor core provides an HMI interrupt.

For SLB, any error causes the processor to take a machine check interrupt. The FIR bit setting indicates both multi-hit and parity error.

3.9.22.4 Uncorrectable Error Handling

If there is an uncorrectable error (UE) for a translate or a load operation, the instruction will finish with a machine check indication to the ISU. The instruction is flushed and re-executed without generating any machine check, and a counter is maintained to see how many UEs occurred. If the UE occurs more than a threshold, a MC interrupt is taken. For caching-inhibited load operation, a MC interrupt is taken on the first occurrence of the UE.

For the instruction side (I-side), if an instruction is executed and in the nonspeculative path, only then is it treated as a UE. Otherwise, the I-side UE handling mechanism is similar to the D-side.

The core provides the EA of the LSU operation that caused the UE in the DAR register. For a UE detected by the IFU for instruction fetches, SRR0 is set to the EA.

Table 3-21 summarizes how the POWER8 processor handles parity, multi-hit, and unrecoverable errors.

Table 3-21. Summary of POWER8 Behavior on Parity Error, Multi-Hit, and Uncorrectable Error

	Parity Error	Multi-Hit	Both Parity Error and Multi-Hit	Uncorrectable Error (UE)
SLB: I-side translation	MC, SRR1, SRR0	MC, SRR1, SRR0	MC, SRR1, SRR0	N/A
SLB: D-side translation, SLBFEE, MFSLB	MC, DSISR, DAR	MC, DSISR, DAR	MC, DSISR, DAR	N/A
TLB: I-side translation	IRR, HMI	MC, SRR1, SRR0	IRR, HMI	N/A
TLB: D-side translation, MFTLB	IRR, HMI	MC, DSISR, DAR	IRR, HMI	N/A
D-ERAT	IRR, HMI	MC, DSISR, DAR	IRR, HMI	N/A
Tablewalk: I-side initiated	IRR, HMI	N/A	N/A	MC, SRR1, SRR0
Tablewalk: D-side initiated	IRR, HMI	N/A	N/A	MC, DSISR, DAR
Load	IRR, HMI	N/A	N/A	MC, DSISR
CI Load	MC, DSISR	N/A	N/A	MC, DSISR
Store	IRR, HMI	N/A	N/A	MC, DSISR
Instruction fetch	IRR, HMI	N/A	N/A	MC, SRR1, SRR0
Any other structure (I-ERAT, other Regfile, I-cache, D-cache and other SRAMs, data-flow hardware control checker)	IRR, HMI	N/A	N/A	N/A

1. MC is a machine check interrupt, IRR is an instruction retry and recovery, HMI is a hypervisor maintenance interrupt, and SRR0, SRR1, DSISR, DAR are various SPRs set on a machine check interrupt. In the TLB, a multi-hit cannot generate a parity error, but a parity error can generate a multi-hit. In the SLB and D-ERAT, multi-hit probably generates a parity error.

3.9.22.5 TLB Parity Error and Multi-Hit Action

Parity = 0 and Multi-hit = 0: No action.

Parity = 1 and Multi-hit = 0: Parity error detected, IRR, followed by HMI (no machine check).

Parity = 0 and Multi-hit = 1: This case is probably caused by software setting up two TLB entries pointing to the same VSID.

Parity = 1 and Multi-hit = 1: Probably multiple bits flipped due to a soft-error that caused the parity error, but also made two VSIDs look the same. The POWER8 core does IRR and then HMI.

3.9.23 Interrupts

3.9.23.1 Interrupt Vectors

Exceptions implemented in the POWER8 processor are listed in *Table 3-22*.

Table 3-22. Interrupt Vector

Exception Type	Exception Value
System Reset	0X00100
Machine Check	0X00200
Data Storage Interrupt	0X00300
Data Segment Interrupt	0X00380
Instruction Storage Interrupt	0X00400
Instruction Segment Interrupt	0X00480
External Interrupt	0X00500
Alignment Interrupt	0X00600
Program Interrupt	0X00700
Floating-Point Unavailable	0X00800
Decrementer Interrupt	0X00900
Hypervisor Decrementer Interrupt	0X00980
Directed Privileged Doorbell Interrupt	0X00A00
Reserved	0X00B00
System Call	0X00C00
Trace Interrupt	0X00D00
Hypervisor Data Storage Interrupt	0X00E00
Hypervisor Instruction Storage Interrupt	0X00E20
Hypervisor Emulation Assistance Interrupt	0X00E40
Hypervisor Maintenance Interrupt	0X00E60
Directed Hypervisor Doorbell Interrupt	0X00E80
Performance SPMC Interrupt	0x00EE0
Performance Interrupt	0X00F00
VMX Unavailable Interrupt	0X00F20
VSX Unavailable Interrupt	0X00F40
Facility Unavailable Interrupt	0X00F60
Hypervisor Facility Unavailable Interrupt	0X00F80
Soft patch Interrupt	0X01500
Debug Interrupt	0X01600

3.9.23.2 Interrupt Definitions

Table 3-23 describes the interrupts that have been added to the POWER8 processor core.

Table 3-23. New Interrupt Vectors for POWER8

Exception Type	Exception Value
Directed Privileged Doorbell Interrupt	x'00A00'
Directed Hypervisor Doorbell Interrupt	x'00E80'
Facility Unavailable Interrupt	x'00F60'
Hypervisor Facility Unavailable Interrupt	x'00F80'

In addition to the interrupt types defined in the Power ISA, the POWER8 core supports several implementation-specific interrupt types. These are summarized in *Table 3-24* and described in more detail in the subsequent sections

Table 3-24. Implementation-Specific Interrupt Types

Exception Type	Exception Value
Performance SPMC Interrupt	x'00EE0'

Implemented MSR and SRR1/HSRR1 Bits

Table 3-25. Implementation MSR and SRR1/HSRR1 Bits (Sheet 1 of 2)

Bits	MSR	SRR1/HSRR1
0	SF	SF
1	Reserved	Reserved
2	Not Implemented	Not Implemented
3	HV	HV
4	Not Implemented	Not Implemented
5	SLE (Split Little Endian	SLE (Split Little Endian)
6:28	Not Implemented	Not Implemented
29:30	TS (Transactional State)	TS (Transactional State)
31	TM (Transactional Memory Available)	TM (Transactional Memory)
32	Not Implemented	Not Implemented
33	Not Implemented	Specific Interrupt Information
34	Not Implemented	Not Implemented
35:36	Not Implemented	Specific Interrupt Information
37	Not Implemented	Not Implemented
38	VMX	VMX
39	Not Implemented	Not Implemented
40	VSX	VSX
41	Not Implemented	Not Implemented
42:47	Not Implemented	Specific Interrupt Information

Table 3-25. Implementation MSR and SRR1/HSRR1 Bits (Sheet 2 of 2)

Bits	MSR	SRR1/HSRR1
48	EE	EE
49	PR	PR
50	FP	FP
51	ME	ME
52	FE0	FE0
53	SE	SE
54	BE	BE
55	FE1	FE1
56	US	US
57	Not Implemented	Not Implemented
58	IR	IR
59	DR	DR
60	Not Implemented	Not Implemented
61	PMM	PMM
62	RI	RI
63	LE	LE

3.9.23.3 System Reset Interrupt

The system reset interrupt is a non-maskable, asynchronous interrupt that is caused by an SCOM command for a soft reset.

Note: There is no explicit SRESET pin; SRESET must be invoked from the service processor.

Table 3-26. System Reset Interrupt

Register	Bits	Description
SRR0	0:63	Set to the effective address of the instruction that the processor would have.
SRR1	0:31	Implemented bits loaded from the MSR.
	32	Set to '0'.
	33	LPAR mode switch occurred while the thread was in power savings mode.
	35:36	Set to '0'.
	42:45	Interrupt caused by IFU detection of a hardware uncorrectable error (UE) 0000 Reserved by pervasive function. 0010 Interrupt caused by SCOM when not in power-saving mode or caused by back-to-back SRESET. 0011 Interrupt caused by hypervisor door bell. 0101 Interrupt caused by privileged door bell. 0100 Interrupt caused by SCOM when in power-saving mode. 0110 Interrupt caused by decrementer wake-up when in power-saving mode. 1000 Interrupt caused by external interrupt wake-up when in power-saving mode. 1010 Interrupt caused by HMI wake-up when in power saving mode. 1100 Interrupt caused by implementation-specific wake-up when in power-saving mode.
	46:47	Indicates if the interrupt occurs when the processor is in power-saving mode. 00 Interrupt did not occur while the processor was in power-saving mode. 01 Interrupt occurred while the processor was in power saving mode. The state of all resources was maintained as if the processor was not in power-saving mode 10 Interrupt occurred while the processor was in power-saving mode. The state of some resources was not maintained but the state of all hypervisor resources was maintained as if the processor was not in power-saving mode and the state of all other resources is such that the hypervisor can resume execution. 11 Interrupt occurred while the processor was in power-saving mode. The state of some resources was not maintained, and the state of some hypervisor resources was not maintained or the state of some resources is such that the hypervisor cannot resume execution.
	62	Loaded from MSR[62] if recoverable. Otherwise set to zero
	Others	Implemented bits loaded from MSR.

The POWER8 core implements a 1-deep queue to remember the reason of a subsequent system reset interrupt while a system reset interrupt is pending. The reason of the most important subsequent system reset interrupt is remembered per the following priority:

1. Hypervisor doorbell initiated system reset
2. Privileged doorbell initiated system reset
3. SCOM-initiated system reset
4. HMI-initiated system reset
5. External-initiated system reset
6. Decrementer-initiated system reset
7. Implementation-specific initiated system reset

3.9.23.4 Machine Check Interrupt

The are several possible causes of machine check interrupts in the POWER8 chip, some of which are generally recoverable and some of which are non-recoverable.

The following causes of machine check interrupts are precise and synchronous with the instruction that caused the operation which encountered the error (that is, SRR0 contains the address of the instruction that caused the operation).

1. The detection of either a parity error, or a multi-hit error, or both in the SLB during the execution of a load, store SLBFEE, or MFSLB instruction. If the interrupt is caused by a soft error, executing the appropriate sequence of instructions in the machine check handler program clears the error condition without causing any loss of state, permitting the interrupted program to be resumed if MSR[RI] was a '1' when the instruction that encountered the error was executed.
2. If there is a multi-hit in D-ERAT or TLB, the core finishes the operation with a machine check interrupt and sets the proper DSISR bit to indicate where the multi-hit occurred.
3. If there is an uncorrectable ECC error when a load instruction is executed or when the page table is being searched in the process of translating an address, the instruction finishes with a machine check indication to the instruction sequencing unit. The instruction is flushed and re-executed without generating any machine check. A counter is maintained to see how many UEs occurred. If the UE occurs more than a pre-established threshold, a machine check interrupt is taken.
4. For a caching-inhibited load operation, the machine check interrupt is taken on the first occurrence of the UE.
5. For the I-side, if an instruction is executed and the instruction is in the nonspeculative path, only then will it be treated as a UE. Otherwise, the I-side UE handling mechanism is similar to the D-side.

In the cases described in items (2), (3), (4) and (5), no state is lost in the processor, but recovery of the correct data might not be possible.

For more traumatic errors or hard errors, these characteristics cannot be reliably provided on a machine check because it is likely that the failure will prevent reliable execution. Additionally, a machine check interrupt that occurs when MSR[ME] = '0' results in a checkstop.

In the POWER8 core, there is no asynchronous machine check interrupt. A machine check interrupt is taken when the machine check input pin is asserted, if enabled by HID0[32] = '1'. The FIR, debug logic, and hang recovery logic can also be programmed to induce machine check interrupts for various error conditions. In general, the POWER8 core works hard to make these interrupts recoverable, but there are some scenarios where it cannot achieve this. Software can use the MSR[RI] bit to help identify the cases where the machine check interrupt is recoverable.

Information about the suspected source of the error condition is logged into either the SRR1 Register, the DSISR Register, or both as defined in *Table 3-27* for synchronous machine checks.

Table 3-27. Synchronous Machine Checks

Register	Bits	Description
SRR0	0:63	Effective address of the next instruction that would have executed if the machine check interrupt was not taken. For cases where this is a recoverable machine check due to a load that has surfaced an error, this will be the address of the load instruction itself. (The POWER8 core allows the instruction to execute to surface the error, but inhibits the commitment of the results.) For cases where this is a recoverable machine check due to an instruction fetch surfacing an error, this will be the address of an instruction that initiated the memory/cache access.
SRR1	42	Interrupt caused by load/store detection of error (see DSISR).
	36, 43:45	Interrupt caused by instruction fetch, indicated by the following encoding: 0000 Reserved. 0001 Interrupt caused by a hardware uncorrectable error detected while doing an instruction fetch (but not translation related). 0010 Interrupt caused by an SLB parity error while translating an instruction fetch address. 0011 Interrupt caused by an SLB multiple hit, while translating an instruction fetch address. Note: This condition occurs if the ESID fields of two or more SLB entries contain the same value. 0100 Interrupt caused by an I-ERAT multi-hit error. 0101 Interrupt caused by a TLB multiple-hit error detected while translating an instruction fetch address. Note: This condition occurs if an address is mapped to both a small and large page in the SLB. This condition can also occur due to a software bug, when a software-managed TLB mechanism is used. 0110 Interrupt caused by a hardware UE detected while doing a TLB reload for I-side. 0111 Reserved. 1000 Interrupt caused by an L2 abort on an instruction fetch due to foreign link time out. 1001 Interrupt caused by an L2 abort on an instruction tablewalk due to foreign link time out. 1011 Reserved. 11xx Reserved.
	62	Loaded from MSR[62] if recoverable. Otherwise, set to zero.
	others	Implemented bits loaded from MSR.

Table 3-27. Synchronous Machine Checks

Register	Bits	Description
DSISR	32:47	All zeros.
	48	Interrupt caused by a UE deferred error, but not for tablewalk (D-side only).
	49	Interrupt caused by a UE deferred error during a tablewalk (D-side).
	50	Nest abort.
	51	Nest abort for tablewalk.
	52	Interrupt caused by a D-ERAT multi-hit.
	53	Interrupt caused by a TLB multi-hit due to translation (D-side only) or MFTLB operation.
	54	Secondary D-ERAT multi-hit.
	55	Interrupt caused by a SLB parity error (translate lookup or mfslbfee) due to a translation (D-side only), SLBFEE, or MFSLB instruction.
	56	Interrupt caused by an SLB multi-hit (might not be recoverable) for translation (D-side only), SLBFEE, or MFSLB instruction.
	57	Zero.
	58:63	All zeros.
DAR	0:63	Effective address computed by a load or store instruction that caused the operation that encountered a parity error, or multi-hit, or both in the SLB, or which encountered a multi-hit in the TLB, or which encountered a multi-hit in D-ERAT, or which encountered an uncorrectable error (UE) while attempting to reload a TLB entry. For all other types of machine check interrupts, the DAR is undefined (including the case where the operand of the load instruction contains a UE). 1. SLB parity error, multi-hit, or both: DAR is loaded with the EA of the target of the load or store instruction that caused the error. 2. TLB multi-hit: DAR is loaded with the EA of the target of the load or store instruction that caused the error. 3. D-ERAT multi-hit: DAR is loaded with the EA of the target of the load or store instruction that caused the error. 4. UE on D-side table walk: DAR is loaded with the EA of the target of the load or store instruction. 5. UE on instruction fetch: DAR is undefined. 6. UE on I-side tablewalk: DAR is undefined. 7. UE on load or store instruction: DAR is undefined (EA is not available in LMQ for loads, so DAR cannot be loaded).

DSISR Implementation Note: All the bits have been implemented in hardware.

Machine Check Interrupt Handler Notes:

As mentioned previously, the machine check interrupt handler is expected to help hardware recover from certain types of D-ERAT, TLB, and SLB errors detected by the hardware. In general terms, the interrupt handler must check whether or not the machine check interrupt is recoverable (by looking at the state of the RI bit in SRR1). It must determine the type of error that caused the machine check (by looking at the state of the SRR1 and DSISR registers). It must flush the contents of the array that reported the detected error (this process is slightly different for each of the possible arrays). Finally, it must return to the interrupted process.

3.9.23.5 Hypervisor Maintenance Interrupt

The POWER8 hypervisor maintenance interrupt is implemented to replace the malfunction alert and thermal interrupt; and to provide support for recovery function. The HMER Register contains the sources of the interrupt, which can be masked by setting the HMEER enable bits to zero. For successful recovery, HMER setting indicates successful recovery.

3.9.23.6 External Interrupt

The POWER8 external interrupt is signaled by the assertion of the external interrupt input signal. The external interrupt signal must remain asserted until the processor has actually taken the interrupt. Failure to meet this requirement can lead the processor to not recognize the interrupt request.

3.9.23.7 Alignment Interrupt

See *Section 3.1.4.2* for details on when the POWER8 core takes alignment interrupts.

Table 3-28 shows how the DSISR and DAR are set for alignment interrupts.

Table 3-28. Alignment Interrupt

Register	Bits	Description
DSISR	32:63	Unchanged.
DAR	0:63	See *Table 3-2* and *Table 3-3* for how the DAR is set on alignment interrupts.

3.9.23.8 Trace Interrupt

The trace interrupt is taken when the single-step trace-enable bit (MSR[SE]) or the branch trace enable bit (MSR[BE]) is set and an instruction successfully completes. After a trace interrupt is taken, SRR0, SRR1, SIAR, and SDAR are set as shown in *Table 3-29*.

Table 3-29. Trace Interrupt (Sheet 1 of 2)

Register	Bits	Description
SRR0	0:63	Set as specified in the architecture.

Table 3-29. Trace Interrupt (Sheet 2 of 2)

Register	Bits	Description
SRR1	0:32	Implemented bits loaded from the MSR.
	33:34	'10'
	35	Set for a load instruction; otherwise, cleared.
	36	Set for a store instruction; otherwise, cleared.
	37:41	Loaded from the MSR.
	42	Set for a **lbarx/lharx/lwarx/ldarx/lqarx** or **stbcx./sthcx./stwcx./stdcx./stqcx.** instruction; otherwise, cleared.
	43	Set to a '1' if a CIABR trace.
	44:47	Set to '0'.
	48:63	Implemented bits loaded from the MSR.
	Note: Bit 35 and 36 are not set if an X-form Load String or Store String instruction specifies an operand length of 0.	
SIAR	0:63	Set to the effective address of the traced instruction; undefined if a CIABR trace.
SDAR	0:63	If the instruction that took the trace interrupt was a storage access instruction, the SDAR is set to the effective address of the storage access. SDAR is not set if an X-form Load String or Store String instruction specifies an operand length of 0; undefined if a CIABR trace.

The contents of SIAR and SDAR are undefined until a trace interrupt occurs.

3.9.23.9 Performance Monitor Interrupt

The performance monitor interrupt is signaled when the MSR[EE] bit is set, the MMCR0[PMAE] bit is set, and any of the performance monitor counters overflow (this includes the eight performance counters defined in the SPR space, as well as the counters defined in MMIO space for the nest).

After such an event is detected, the POWER8 core waits for previously dispatched instructions to complete, and then takes the interrupt.

3.9.23.10 SPMC Performance Monitor Interrupt

The SPMC performance monitor interrupt is an implementation-specific interrupt introduced on the POWER8 core. The SPMC performance monitor interrupt is signaled when an SPMC overflow exception occurs.

The MSR, SRR0, SRR1 interrupt settings are the same as the architected performance monitor interrupt as described by the POWER ISA. For the SPMC performance monitor interrupt execution resumes at the effective address x'0000_0000_0000_0EE0'.

3.9.23.11 Facility Unavailable Interrupt

The POWER8 core implements the facility unavailable interrupt as defined in the Power ISA.

3.9.24 Logical Partitioning (LPAR) Support

The following sections describe the POWER8 implementation aspects of the LPAR architecture. The POWER8 core supports one, two, and four LPARs referred to as 1 LPAR, 2 LPAR and 4 LPAR mode respectively.

3.9.24.1 Thread to LPAR mapping

Logical threads are tied to partitions as follows:

- 2 LPAR mode
  - LPAR0 - threads 0, 1, 2, and 3
  - LPAR1 - threads 4, 5, 6, and 7
- 4 LPAR mode
  - LPAR0 - threads 0 and 1
  - LPAR1 - threads 2 and 3
  - LPAR2 - threads 4 and 5
  - LPAR3 - threads 6 and 7

The snooped TLBIE in 4 LPAR mode only invalidates the D-ERAT and I-ERAT entries for the partitions that matched. If only one LPAR ID matches, the thread pair for that LPAR is used to invalidate the D-ERAT and I-ERAT. If two LPAR IDs match, two thread pairs for that LPAR are used for ERAT invalidation. If more than two LPAR IDs match, no thread bits are used for ERAT invalidation.

3.9.24.2 Dynamic LPAR Switching

In 2 LPAR or 4 LPAR mode, the POWER8 core runs in SMT8 mode and each LPAR is entitled to its share of execution bandwidth, irrespective of how many threads are actually active on the core. HID0 Register bits 11, 12, and 15 are used to control dynamic LPAR switching.

- HID0[DYN_LPAR_DIS]. If not set and the core is in either 2 LPAR or 4 LPAR mode, hardware switches the core to 1 LPAR mode when threads 1 - 7 are all in *nap* mode.
- HID0[1LPARto2LPAR]. A write to the bit initiates a dynamic switch from 1 LPAR mode to 2 LPAR mode.
- HID0[1LPARto4LPAR]. A write to the bit initiates a dynamic switch from 1 LPAR mode to 4 LPAR mode

3.9.25 Strong Access Ordering Mode (SAO)

The POWER8 core supports the SAO mode defined in Power ISA.

3.9.26 Graphics Data Stream Support

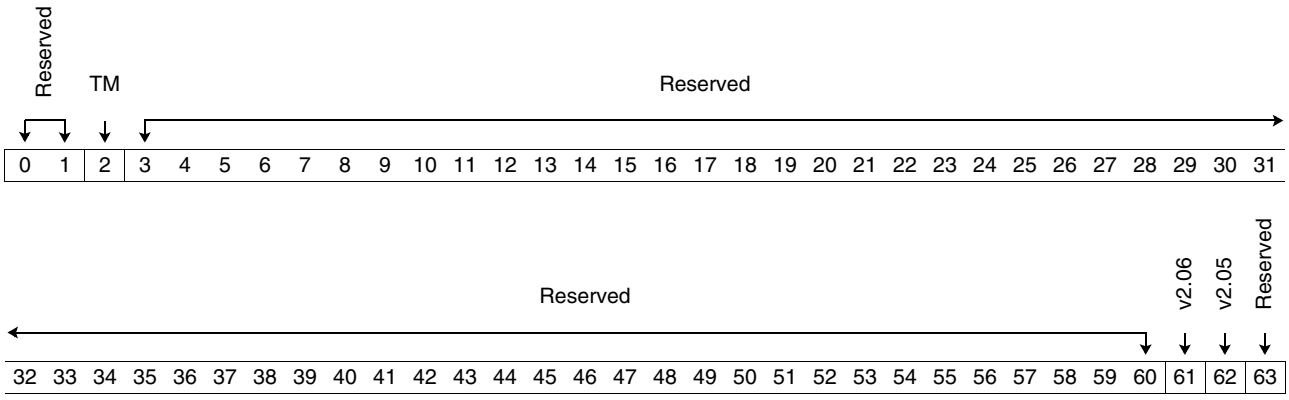
For cache-inhibited stores, the POWER8 core provides store gathering with an intentional stall to maximize the amount of gathering that can occur.

3.9.27 Performance Monitoring, Sampling, and Trace

Performance monitoring facilities have been incorporated into the POWER8 processor to enable the collection of performance related data and instruction traces. In general, the POWER8 core supports the recommended architecture for performance monitoring as described in the Power ISA.

3.9.28 Processor Compatibility Mode

The POWER8 core implements the Processor Compatibility Register (PCR) as described in the Power ISA to facilitate partition migration. Setting PCR[2] = '1' disables the transactional memory (TM) instructions in problem state. Setting PCR[61] = '1' disables all problem state instructions and facilities that were added in Version 2.07 of the Power ISA. Thus, setting this bit effectively makes a POWER8 core architecturally appear to problem state software as a Power ISA version 2.06 core. Likewise, setting PCR[62] = '1' disables all problem state instructions and facilities that were added in Version 2.06 of the Power ISA. Thus, to migrate a partition from a version 2.05 system to a POWER8 (version 2.07) system, PCR[61:62] must be set to '11'.



The POWER8 core does not follow the Power ISA for controlling the problem state code's ability to set thread priority to '001' based on the PCR[v2.06] bit. Instead, the POWER8 core controls this only by using the HID1[11:14] partition bits. See *Section 3.8.2 HID1 Register* on page 104.

4. Storage Subsystem

4.1 L2 Cache

The L2 cache is a unified cache that is accessed privately by a given core on the POWER8 processor. The L2 cache maintains full hardware coherency within the system and can supply cache intervention data to other cores on this die or on other POWER8 chips (for example, both on-chip and off-chip intervention). The L2 cache is a store-in cache that is fully inclusive of both the D-cache and I-cache for its private core (that is, the core has a store-through L1 D-cache). The L2 cache also supports private bus access to an 8 MB L3 cache region that is also private to this core.

4.1.1 L2 Cache Features

A summary of the L2 cache follows:

- 512 KB private cache per core
  - 128-byte line, 8-way set associative
  - Both I-side and D-side inclusive (ICBI is not required on the SMP interconnect)
  - Double-banked cache design interleaved on even or odd cache-line boundary
  - Can perform a read from one bank while writing to the other bank
  - Cache-array data protected by 8-byte SECDED ECC
- 8-way directory, dual-banked multiport
  - One processor read port, one snoop read port, and one write port per bank
  - The processor port into a given bank operates at ½ the processor clock rate (initiated on 2:1 clock boundary)
  - The snoop port into a given bank operates at ½ the processor clock rate (initiated on 2:1clock boundary) allowing for two snoops per 2:1clock across the two banks.
  - A bank can initiate up to two directory reads in a given 2:1 cycle (one on the snoop port, one on the processor port). A bank can initiate one write in a given pclk cycle (no reads allowed during a write pclk cycle)
  - Directory array data protected by SECDED ECC
- 512 $\times$ 13-bit LRU arrays (logical configuration); 2 $\times$ 4 LRU vector tracking tree with cache-invalidate state biasing
- Seven processor cycles latency beyond L1 data cache on an L2 hit through the fastpath
- Eight processor cycles latency beyond L1 instruction cache on an L2 hit
- Point of global coherency
- Reservation stations, one per processor thread
- Support for transactional memory (TM) transactions
- Support **icswx** for coprocessor initialization
- Dual snoop bus port split by even and odd cache-line address
- Support for micropartition prefetch assist logging
- Support strong access ordering (SAO)
- Hardware line delete for error tolerance

4.2 L3 Cache

The L3 cache controller services read and cast-out requests from the attached L2 cache and snoop requests from the fabric. It also provides a mechanism to prefetch cache lines into the L3 cache based on requests from the core. Data movement for L2 cast-outs (cast-ins for the L3 cache) are carried out using a private interface between the L2 and L3 cache. Data movement for L3 cast-outs, L3 interventions, and memory pushes are carried out by sending commands and data to the fabric interface. The L3 cache supports victim mode for others on chip L3 cache (referred to as L3.1 caches).

4.2.1 L3 Features, Queues and Resources

- Local 8 MB L3 cache region and shared L3.1
  - 128-byte cache line, 8-way set associative.
  - 8 cache banks (interleaved for access overlapping)
  - 64-byte reload and CI data buses.
  - 128-byte read or write per bank every 12 pclks.
  - Sixty-four 1Mb EDRAM macros configured in eight 8-byte dataflows.
  - 8-way directory: 4 banks, up to 4 reads or 2 reads and 2 writes every 2 pclks to differing banks, Physically 16 2048 x 2w x 50 bit SRAMs.
  - 26.5 pclock L1 data-cache load to use penalty.
  - Cache contents protected by 8-byte ECC.
  - Directory contents protected by ECC.
  - LRU algorithm with enhancements (for efficient L3.0/L3.1 victimization)
- Functionality
  - L3.0 management of victim lines from local L2 cache.
  - L3.1 management of victim lines from on-chip L3 caches.
  - Services load/store/l3 prefetch misses from local L2 cache.
  - Dual Snoop ports split by even and odd cache-line address.
  - Streaming data prefetch and cache-inject support.
  - Dual-class L3.0/L3.1 LRU support and L3.1 activity throttling.
  - Support for speculative memory transactions, including footprint tracking to detect collisions and mechanisms for conditional completion. Consists of fast flash 64 CL per thread and slow walk entire 8 MB available for one thread.
  - Fast broadcast on-chip fetch request to memory controllers and other L3 caches for L2 demand load miss L3.
- Entire L3 cache clocked at ½ core frequency.

4.3 NCU

The POWER8 noncacheable unit (NCU) is responsible for processing noncacheable operations (such as, load and store operations with I = '1') and certain other uncacheable operations such as TLBIE, various sync instructions, ptesync instructions, and so on. All of these instructions support the behavioral definitions given in the Power ISA documents. One NCU unit is instantiated per core and this NCU handles operations for all eight threads in the associated POWER8 core.

The POWER8 NCU provides one dedicated cache-inhibited load station per thread to process one outstanding cache-inhibited load per thread. In the POWER8 implementation, cache-inhibited loads (whether guarded or not) are not gathered and are not reordered in any fashion.

For stores, sixteen 64-byte store gather stations are provided and are shared across the eight core threads. A pair of 64-byte stations can "chain" together to gather up to full 128-byte lines. The POWER8 NCU supports gathering and reordering for stores in the IG = '10' space. In the IG = '11' space, stores are neither reordered nor gathered as required by the architecture. The POWER8 NCU only gathers 4-, 8-, and 16-byte stores. The stores must be naturally aligned, start at a given address, not overlap and be contiguous. This is designed to support the general gathering case of a set of stores of a given size starting at the beginning of a 128-byte block and continuing on to fill the entire 128-byte block.

4.3.1 NCU Characteristics

The NCU has the following characteristics:

- Store buffer
  - 16 × 64-byte store gather stations (chainable to 8 × 128-byte gathered lines).
  - The store buffers are shared across threads (LSU backoff mechanism prevents any thread from blocking any other thread).
- Store modes (IG = '1X')
  - IG = '11' mode; stores are done in-order and no gathering is allowed.
  - IG = '10' mode; stores can be gathered and reordered.
- Loads. One outstanding load per thread.

4.4 Memory Controller

The POWER8 memory controller function is split between the POWER8 memory controller (MC) unit, which provides an interface to the processor bus, and the POWER8 Memory Buffer chip, which provides memory command scheduling, a memory buffer cache, and memory diagnostic functions. The POWER8 MC unit is connected to the POWER8 Memory Buffer chip through the differential memory interface (DMI) channel interface.

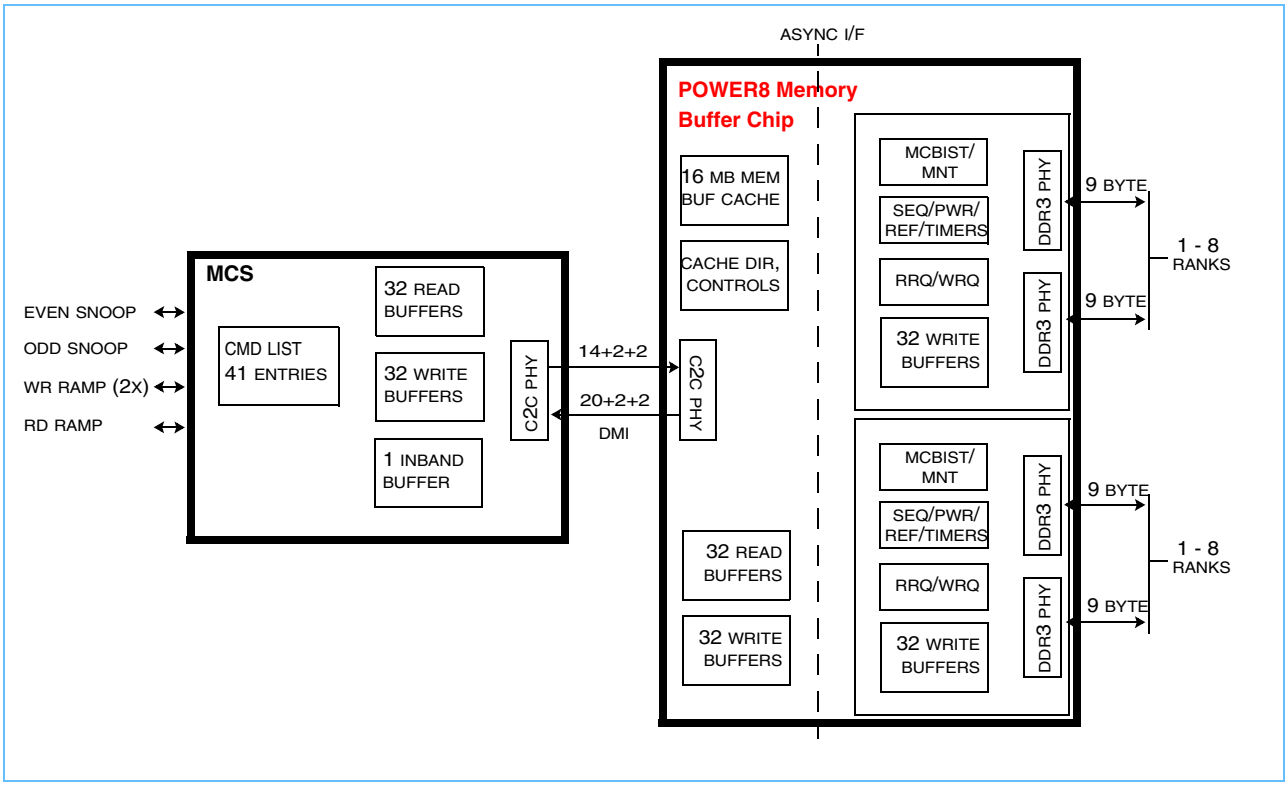
The POWER8 memory controller design has two memory controller units and 4 DMI ports per chip.

4.4.1 POWER8 Memory Stack Partitioning

Figure 4-1 shows a memory buffer chip connected to a memory controller synchronous (MCS) unit.

The POWER8 Memory Buffer chip contains four DDR3/4 memory ports that interface to Industry-Standard RDIMMs or LRDIMMs. Two of the four memory ports on the memory buffer operate in tandem to read or write a cache line of data using a burst length 8 (BL8) operation to the installed DIMMs.

Figure 4-1. Memory Stack Partitioning



4.4.2 POWER8 Chip Memory Controller Unit Features

- Physical Organization
  - POWER8 processor contains two memory controller units per chip.
  - Each memory controller unit contains two independent MCS units.
  - MCS units are accessed in pairs for selective memory mirroring.
  - Each MCS unit connects to a single DMI channel.
- Processor Bus Interface (per MCS unit)
  - Single 16-byte read data ramp, dual 16-byte write data ramp interfaces
  - 128-byte cache lines supported (cache line interleaving on a 32-byte basis)
  - Automatic maintenance of domain bits for multi-node systems
  - Speculative dispatch on a bandwidth-available basis
  - Delivery of critical octword of read data not gated by reading the entire cache line
- Memory Channel Interface (per MCS unit)
  - Downstream command/write data interface, up to 1 TB of memory addressing
  - Upstream tagged read data/status interface
  - CRC-protected
  - Error status and logging interface
  - Automatic hardware replay of soft channel faults
  - Automatic reconfiguration to use spare channel lanes to recover from hard failures
  - Hardware and software channel initialization capability
- Pervasive Interfaces (per MCS unit)
  - Performance monitor interface
  - SCOM interface
  - Debug bus interface (to shared trace array in processor bus logic)
- Resources (per MCS Unit)
  - Forty-one command list entries (32 memory entries, eight address-only entries, one inband configuration register entry)
  - Thirty-two 128-byte read data buffers
  - Thirty-two 128-byte write data buffers
  - One 128-byte configuration register read/write buffer
- Mirroring
  - Selective memory mirroring supported across MCS pairs. The pairs of MCS units must be within the same MC unit.
  - Mirrored memory accessed through unique memory address range (independent of memory extent).
- Clocking

  The memory buffer supports the following fixed clocking ratios and the maximum and minimum frequencies shown in *Table 4-1* and *Table 4-2* on page 142:
  - 1:1 with processor bus
  - 1:4 with DMI channel

Table 4-1. Maximum Frequencies

DRAM Speed (GHz)	POWER8 Memory Buffer DDR Frequency (GHz)	Processor Bus/MCS Frequency (GHz)	POWER8 Memory Buffer Cache Frequency (GHz)	DMI Channel Frequency (GHz)
1.33	1.33	2.4	1.2	9.6
1.6	1.6	2.4	1.2	9.6

Table 4-2. Minimum Frequencies

DRAM Speed (GHz)	POWER8 Memory Buffer DDR Frequency (GHz)	Processor Bus/MCS Frequency (GHz)	POWER8 Memory Buffer Cache Frequency (GHz)	DMI Channel Frequency (GHz)
1.33	1.33	2.0	1.0	8.0
1.6	1.6	2.0	1.0	8.0

4.4.2.1 Bandwidth

Table 4-3 lists the bandwidth per MCS unit per POWER8 Memory Buffer.

Table 4-3. Bandwidth (per MCS/POWER8 Memory Buffer)

Performance Parameter	Goal
Maximum downstream (write) DMI channel data bandwidth, 9.6 GHz channel	9.6 GB/s
Maximum upstream (read) DMI channel data bandwidth, 9.6 GHz channel	19.2 GB/s
Note: See the *POWER8 Memory Buffer User's Manual* for additional performance parameter goals.	

4.4.3 POWER8 Memory Controller Characteristics

Table 4-4 lists key characteristics of the POWER8 memory controller.

Table 4-4. POWER8 Memory Controller Characteristics (Sheet 1 of 2)

Parameter	POWER8
Processor Bus Interface	
Reflected command/partial response/combined response interfaces	Two per MCS unit
Read data ramp	One 16-byte read data ramp per MCS unit
Write data ramp	Two 16-byte write data ramp per MCS unit
Read buffers	Thirty-two 128-byte buffers per MCS unit
Write buffers	Thirty-two 128-byte buffers per MCS unit
Processor bus transfer size	32 bytes
Cache-line size	128 bytes
Command list entries	Thirty-two read/write, eight address-only, one configuration register per MCS unit
Address hashing	1, 2, or 4 MCS/group, 128-byte hashing

Table 4-4. POWER8 Memory Controller Characteristics (Sheet 2 of 2)

Parameter	POWER8
Memory Interface	
Memory channels	One per MCS unit
Channel speed/signaling	9.6 Gb/s maximum differential
Channel command/write data bus format	Combined command/data buses
Channel error protection	CRC, with retry
Read delay latency calculation	None, returned read data is tagged
Channel initialization/calibration	Dynamic
Channel lane sparing	Firmware (CRC syndrome capture), hardware
Memory fetch/store sizes	128 bytes
Channel speed ratios	4:1 (to SMP interconnect frequency)

5. Simultaneous Multithreading

5.1 Overview

The POWER8 processor core supports ST, SMT2, SMT4, and SMT8 modes. Any thread number can be run in any SMT mode, on any thread set. *Table 5-1* shows the SMT mode definitions.

Table 5-1. SMT Modes

Description	Number of Threads Enabled	Switch to this SMT mode when ...
SMT0	0	POR state
SMT1	1	1 thread
SMT2	1 - 2	2 threads
SMT4	1 - 4	3 - 4 threads
SMT8	1 - 8	5 - 8 threads

5.2 Partitioning of Resources in Different SMT Modes

Table 5-2 lists the resources that are partitioned in certain SMT modes.

Table 5-2. Front-End Execution Core Resource (Sheet 1 of 2)

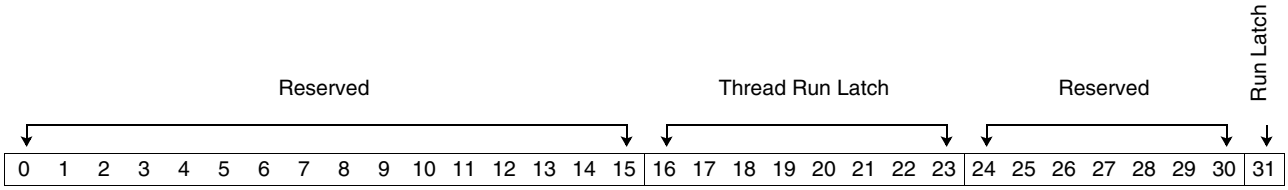
Resource	ST	SMT2	SMT4	SMT8
Fetch Bandwidth				
EAT Entries	24	24 per thread	12 per thread	6 per thread
Instruction Buffer Entries	64	64 per thread	32 per thread	16 per thread
Link Stack	32	32 per thread	16 per thread	8 per thread
D-ERAT Entries	48	48 per thread	48 per thread set	48 per thread set
Decode Bandwidth				
Dispatch Groups	6 nonbranch + 2 branch per thread	3 nonbranch + 1 branch per thread	3 nonbranch + 1 branch per thread	3 nonbranch + 1 branch per thread
GPR Entries	124	124 per thread	124 per thread set	124 per thread set
GPR in-flight renames	92	92 per thread	60 per thread set	60 per thread set
GPR Architected in First Level	32	32 per thread	64 per thread set	64 per thread set
GPR Architected in SAR				64 per thread set in SAR
VRF Entries	144	144 per thread	144 per thread set	144 per thread set
VRF in-flight renames	80	80 per thread	80 per thread set	80 per thread set
FP/VMX Architected in First Level	64	64 per thread	64 per thread set	64 per thread set
FP/VMX Architected in SAR			64 per thread set in SAR	192 per thread set in SAR
Steering	Toggle	Assign by thread set	Assign by thread set	Assign by thread set

Table 5-2. Front-End Execution Core Resource (Sheet 2 of 2)

Resource	ST	SMT2	SMT4	SMT8
Unified Issue Queue Entries	64	32 per thread	32 per thread set	32 per thread set
FXU, LSU, LU, VSU	2 each	1 each per thread	1 per thread set	1 per thread set
Completion Rates	1 group per cycle	1 (3+1) group per thread	1 (3+1) group per thread set	1 (3+1) group per thread set

5.3 Control Register

The Control Register (CTRL) is an architected 32-bit register. The bit assignment for the thread control bits in the CTRL supports up to eight threads. The POWER8 core supports threads 0 - 7. A bit in the CTRL Register represents the architected state for a particular thread.



Bits	Field Name	Description
0:15	Reserved	Reserved.
16:23	Thread Run Latch	Thread Run Latch bits (indirectly written supervisor and hypervisor). Threads 0 - 7.
24:30	Reserved	Reserved.
31	Run Latch	Run Latch for thread doing CTRL read/write (read only / rerouted supervisor or hypervisor write).

The CTRL Register can be read with the **mfspr** instruction using SPR 136 in user, supervisor, or hypervisor state.

The CTRL Register can be selectively written with the **mtspr** instruction using SPR 152 in the supervisor or hypervisor state.

The CTRL Register is initialized to 0x0000_0000 at power-on.

Even though a single CTRL Register is shared by the eight threads, there is no need to obtain a lock before updating the CTRL Register. There is only one bus that goes to the core pervasive unit, and the instruction issue logic serializes all **mtctrl** instructions. Updating the Run Latch bit must be done in hypervisor mode. When updating the Run Latch bit (in hypervisor mode), the software is recommended to set the Thread Enable bits to '00000000'. Setting the Thread Enable bit to '00000000' is not allowed. Therefore, this updates the run latch, but there is no effect to the Thread Enable bit and no thread will be killed or woken up.

CTRL[16:23] contain the Run Latches for threads 0 - 7. A **mtspr CTRL** instruction does not modify CTRL[16:23] based on GPR bits [48:55]. Instead, these bits are indirectly loaded by writing a value to CTRL[31]. The value written to CTRL[31] is loaded into CTRL[16] if thread 0 issued the move to CTRL and CTRL[17] if thread 1 issued the move to CTRL, and so on. A thread cannot update the thread Run Latch bit of another thread.

The run latch bit is only used by software for status and is sent to the performance monitor for performance analysis. For this purpose, the POWER8 processor core supports one run latch per thread. To use this function, if a thread is executing a dispatchable task, software must set the CTRL Run Latch for that thread to '1' by writing a '1' to CTRL[31]. If a thread is in a wait state, waiting for a dispatchable task, software must set the CTRL Run Latch for that thread to '0'.

Software can load the CTRL Register with a Run Latch value for its thread by writing the Run Latch value to CTRL[31]. Hardware routes data directed to CTRL[31] into either CTRL[16] - CTRL[23], depending on which thread is doing the write (see previous definition of CTRL[16:23]). When the CTRL Register is read, data driven on CTRL[31] comes from CTRL[16] - CTRL[23], depending on which thread is doing the read. CTRL[31] does not physically exist in hardware.

The data read on a **mfspr**(CTRL) is formatted differently based on the MSR[PR] and MSR[HV] bits and whether the core is in Big Core, 2 LPAR, or 4 LPAR mode. Bit 63 is always the Run Latch of the thread executing the **mfspr**. Bits 48:55 are formatted as shown in *Table 5-3*, where R0 equals run latch for thread 0, RT equals run latch of thread executing **mfspr**.

*Table 5-3. **mfspr** CTRL Data Formatting*

MSR[HV], MSR[PR]	Core Mode	Bits 48:55
00 - Privileged	Single LPAR	R0, R1, R2, R3, R4, R5, R6, R7
*1 - Problem	Single LPAR	RT, 0, 0, 0, 0, 0, 0, 0
10 - Hypervisor	Single LPAR	R0, R1, R2, R3, R4, R5, R6, R7
00 - Privileged	4 LPAR (Threads 0, 1)	R0, R1, 0, 0, 0, 0, 0, 0
00 - Privileged	4 LPAR (Threads 2, 3)	R2, R3, 0, 0, 0, 0, 0, 0
00 - Privileged	4 LPAR (Threads 4, 5)	R4, R5, 0, 0, 0, 0, 0, 0
00 - Privileged	4 LPAR (Threads 6, 7)	R6, R7, 0, 0, 0, 0, 0, 0
*1 - Problem	4 LPAR	RT, 0, 0, 0, 0, 0, 0, 0
10 - Hypervisor	4 LPAR	R0, R1, R2, R3, R4, R5, R6, R7
00 - Privileged	2 LPAR (Threads 0, 1, 2, 3)	R0, R1, R2, R3, 0, 0, 0, 0
00 - Privileged	2 LPAR (Threads 4, 5, 6, 7)	R4, R5, R6, R7, 0, 0, 0, 0
*1 - Problem	2 LPAR	RT, 0, 0, 0, 0, 0, 0, 0
10 - Hypervisor	2 LPAR	R0, R1, R2, R3, R4, R5, R6, R7

5.4 Thread Priority, Status, and Control Requirements

Thread priority, control, and status registers enable software to do the following:

- Give a large percentage of execution resources to critical tasks.
- Reduce the amount of resources and power used by low-priority work.
- Read foreground and background thread priority and status.
- Save and restore priority during interrupts.
- Provide a controlled way to allow supervisor/user code to change priority.
- Provide a means to kill or revive a thread.
- Avoid fine-grain livelock or deadlock situations between threads.

5.5 Thread Balance Control Requirements

The following mechanisms can be used to balance work between threads:

- Reduce ifetch priority of a thread using too many resources.
- Reduce decode priority of a thread using too many resources.
- Hold decode of thread with long latency events.
- Dispatch flush decode pipe to clean congested operations.
- Balance flush from next-to-complete plus one group and hold at IBUF until miss (and so on) resolves.

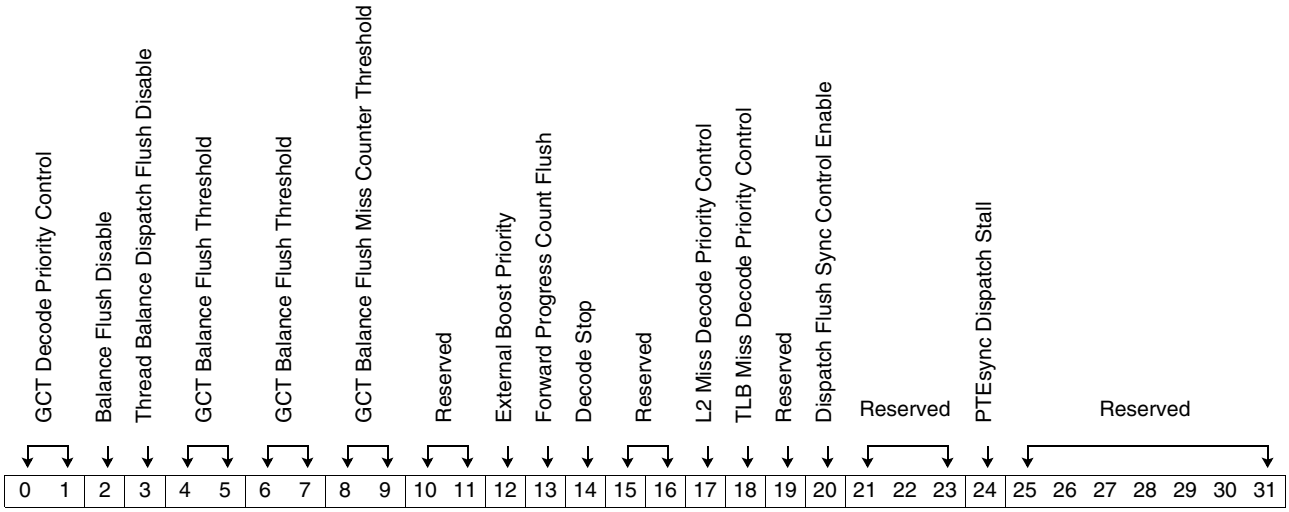
Table 5-4. Thread Balance Control (Balance Flush)

Indicator of Balance	Decode Priority	Decode Hold	Dispatch Flush	Balance Flush
GCT Utilization (covers UniQ utilization)	in plan	in plan	-	in plan[1]
UniQ Full	-	in plan[2]	in plan[2]	-
Sync operations[3]	-	in plan[4]	in plan	-
TLBIE	-	in plan[5]	in plan	-
Score Board Full	-	-	in plan	-
TLB or L2 miss	in plan	-	-	-
TLB or L3 miss	-	In plan	in plan[6]	in plan

1. Only used when a thread is stalled at dispatch and threads other than the stalled thread are consuming too many available resources.
2. If a particular thread or thread pair is using up its allocated UniQ resource, the thread pair is dispatched flushed and held at decode.
3. Synchronization instructions include: **sync**, **lwsync**, **ptesync**, **tlbsync**, **isync**, read/write of nonrenamed SPRs .
4. SYNC: After getting a dispatch flush, a sync instruction is held at IBUF(CLB) until its dispatch conditions are met. This is irrespective of thread priority.
5. TLBIE: After getting a dispatch flush, a **tlbie** instruction is held at the IBUF until the thread's GCT is empty. After the **tlbie** instruction is dispatched, the following instruction gets dispatched flushed and then held at IBUF until the "tlbie ack" is received from the GRS (through LSU). There is no effect on the **tlbiel** instruction. This is irrespective of thread priority
6. A dispatch flush is performed on the thread that is balanced flush only if that thread is stalled at dispatch and TSCR[3] is enabled.

5.6 Thread Switch Control Register (Hypervisor Access Only)

Thread priority controls are programmable. All bits are read/write. There is one Thread Switch Control Register per core. TSCR is initialized to x'0000_0000' at power-on.



Bits	Field Name	Description
0:1	GCT Decode Priority Control	GCT Decode Priority Control. If all threads are at the same software-set priority, decrease priority when a thread is utilizing more than the following number of GCT entries: 00 Function disabled (TSCR POR default). 01 Scan-only latch value programmable 1 - 28. Scanlatch POR default is: 13 (SMT2); 7 (SMT4); 4 (SMT8). 10 Scan-only latch value programmable 1 - 28. Scanlatch POR default is 14 (SMT2); 8 (SMT4); 5 (SMT8). 11 Scan-only latch value programmable 1 - 28. Scanlatch POR default is 15 (SMT2); 9 (SMT4); 6 (SMT8).
2	Balance Flush Disable	Balance Flush Disable 0 Enable NTCP1 balance flushes. 1 Disable NTCP1 balance flushes.
3	Thread Balance Dispatch Flush Disable	Thread Balance Dispatch Flush Disable 0 Enable dispatch flush for the thread that was chosen for a balance flush if that thread is stalled at dispatch. A dispatch flush is a lower-latency flush than a balance flush. 1 Disable. **Note:** Conditions for dispatch flush are the same as a balance flush.
4:5	GCT Balance Flush Threshold for L3/TLB miss	GCT Balance Flush Threshold for L3/TLB miss If a thread is stalled at dispatch with an L3 or TLB miss, consider the thread for balance flush when the thread is using more than the following number of GCT entries: 00 Function disabled (TSCR POR default). 01 Scan-only latch value programmable 1 - 28. Scan latch POR default is: 2 (SMT2); 2 (SMT4); 2 (SMT8). 10 Scan-only latch value programmable 1 - 28. Scan latch POR default is: 3 (SMT2); 3 (SMT4); 3 (SMT8). 11 Scan-only latch value programmable 1 - 28. Scan latch POR default is: 4 (SMT2); 4 (SMT4); 4 (SMT8).

Bits	Field Name	Description
6:7	GCT Balance Flush Threshold if no L3/TLB miss	GCT Balance Flush Threshold if no L3/TLB miss In SMT2 and higher, if a thread is stalled at dispatch with no L3 or TLB misses, consider thread for balance flush when the thread is using more than the following number of GCT entries: 00 Function disabled. Option available to have hardware disabled function in SMT2. 01 Scan-only latch value programmable. 1 - 28. Scan latch POR default is: 22 (SMT2); 14 (SMT4); 7 (SMT8). 10 Scan-only latch value programmable. 1 - 28. Scan latch POR default is: 24 (SMT2); 16 (SMT4); 8 (SMT8). 11 Scan-only latch value programmable. 1 - 28. Scan latch POR default is: 26 (SMT2); 18 (SMT4); 9 (SMT8); 8 (4 LPAR).
8:9	GCT Balance Flush Miss Counter Threshold	GCT Balance Flush Miss Counter Threshold If a thread is chosen to be balanced flushed based on TSCR[4:5] (GCT balance flush threshold), apply a CLB hold to that thread until the miss is resolved or until the following miss counter threshold is reached: 00 Function disabled (If thread is balanced flushed due to TSCR[4:5], CLB hold is applied until a Miss is resolved). 01 Scan-only latch value programmable, 10-bit LFSR; Scan latch POR default 256 cycles (x3C1 LFSR). 10 Scan-only latch value programmable, 10-bit LFSR; Scan latch POR default 384 cycles (x0E4 LFSR). 11 Scan-only latch value programmable, 10-bit LFSR; Scan latch POR default 512 cycles (x20F LFSR) 1023 cycles (x000 lfsr) is the maximum setting available.
10:11	Reserved	Reserved.
12	External Boost Priority	External Boost Priority. If '1' and an external interrupt request is active and the corresponding threads' priority is less than normal priority, set the threads' priority to normal. **Note:** This will *not* change the value in PPR[11:13] for the affected thread.
13	Enable Forward Progress Count Flush	Enable Forward Progress Count Flush. Note: This bit only enables/disables the flush from occurring. The forward progress timer does not stop decrementing when set to '0' SMT2 and higher: If one thread is not making progress, enable flushing the other active threads.
14	Decode Stop	Decode Stop. When set to 0, the forward progress timer (in PPR) is decremented even when the current thread is in decode stop state. When set to 1, the forward progress timer is not decremented when the current thread is in decode stop state.
15:16	Reserved	Reserved.
17	L2 Miss Decode Priority Control	L2 Miss Decode Priority Control. If all threads are at the same software set priority, then: 0 L2 miss disabled for use in adjusting decode priority. 1 L2 miss enabled for use in adjusting decode priority.
18	TLB Miss Decode Priority Control	TLB Miss Decode Priority Control. If all threads are at the same software set priority, then: 0 TLB miss disabled for use in adjusting decode priority. 1 TLB miss enabled for use in adjusting decode priority.
19	Reserved	Reserved.
20	Dispatch Flush Sync Control Enable	Dispatch Flush Sync Control Enable Stop decode if mode for thread with **sync** instruction is outstanding, (always on in shipping mode). Applies only to SMT2 and higher.
21:23	Reserved	Reserved.

Bits	Field Name	Description
24	PTEsync Dispatch Stall	PTEsync dispatch stall. Set to '1' to enable the following function for **ptesync** instruction. Hold the **ptesync** instruction at dispatch until GCT is empty, SRQ is empty, no I-side table walk pending, and wait for a minimum of 15 cycles. After these conditions are satisfied, dispatch the **ptesync**. Afterwards, wait for GCT empty and SRQ empty to continue with the dispatch of further instructions.
25:31	Reserved	Reserved.

The TSCR can be accessed with the **mtspr** and **mfspr** instructions using SPR 921.

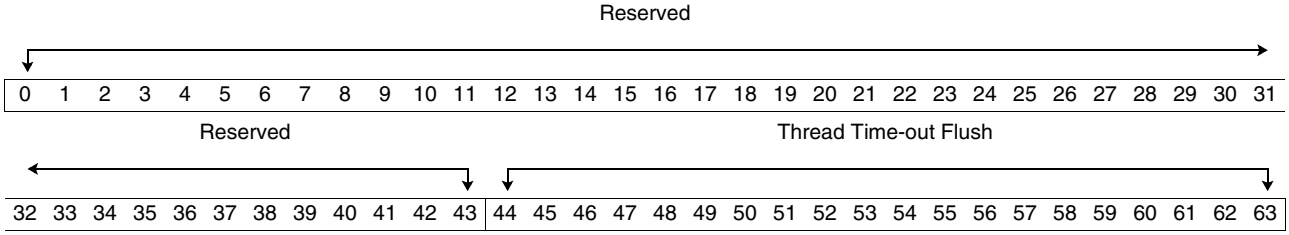
A SCOM latch is added to make this register as read-only for debug purposes.

TSCR is initialized to x'0000_0000' at power-on.

The preferred setting is: x'8ACC_6880'.

5.7 Thread Time-Out Register (Hypervisor only)

The Thread Time-Out Register (TTR) is used to ensure forward progress. There is one TTR per core. For more information see *Section 5.9 Forward Progress Timer* on page 153.



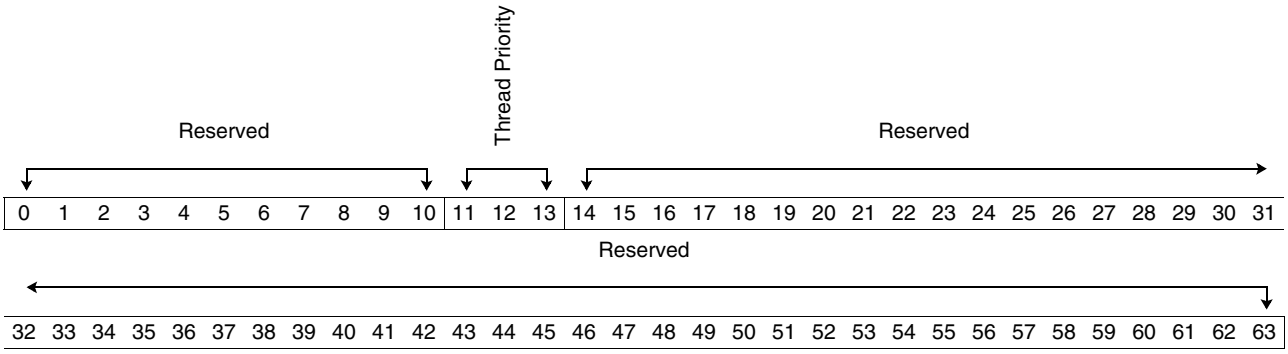
Bits	Field Name	Description
0:43	Reserved	Reserved.
44:63	Thread Time-out Flush Value.	Thread Time-out Flush Value. A x'00000' value generates a maximum count.

The TTR Register can be accessed with the **mtspr** and **mfspr** instructions using SPR 922.

The TTR Register is initialized to x'0000_0000_0000_0000' at power-on.

5.8 Program Priority Register (PPR)

Each thread has a 64-bit status register associated with it. Some bits are read-only, while other bits are read/write. There is one PPR per thread.



Bits	Fields Name	Description
0:10	Reserved	Reserved (not implemented).
11:13	Thread Priority	Thread Priority. 000 Not allowed 001 Very low 010 Low 011 Medium low 100 Normal 101 Medium high 110 High 111 Extra high Set to '100' on system reset interrupt.
14:63	Reserved	Reserved (not implemented).

The local PPR Register can be accessed with the **mtspr** or **mfspr** instructions using SPR 896.

The PPR Register for each thread is initialized to x'0010_0000_0000_0000' at power-on.

5.9 Forward Progress Timer

For the POWER8 core, the forward progress timer bits were moved out of the PPR Register. A non-architected latch bank holds these bits. PPR[44:63] are now reserved, non-implemented bits.

The forward progress latch bits are loaded from TTR[44:63] every time a group of instructions are retired on the current thread.

If the current thread is not in a decode stop state, the counter is decremented by '1' every time a group completes on another thread in the same thread set (see TSCR[14] in *Section 5.6 Thread Switch Control Register (Hypervisor Access Only)* on page 149).

Initialized by scan flush to x'00000' (maximum count)
Decrementer stops at x'00001' (minimum value)

For SMT2 and higher:

A flush of the other active threads in the same thread set occurs when:

- The timer count reaches x'00001'.
- The forward progress count flush is enabled TSC[13] = '1'.
- The group completes on another thread in the same thread set.

After the threads are flushed, no dispatch slots are given to the flushed thread until one group has completed for the current thread.

5.10 Thread Priority NOPs

The thread switch priority can be read or written by software using the **mfspr** and **mtspr** instruction to the Thread Status Register. Thread priority can also be altered by executing special forms of the **or x,x,x** NOP. The priority is changed upon completion of the operation, provided the function is enabled for the current privilege level.

- On the POWER8 core, problem-state programs can set their priority to very-low. TSCR[16] is reserved on the POWER8 core.
- Supervisor programs can set their thread priority to six different values, very-low through high. TSCR[15] is reserved on the POWER8 core.
- Hypervisor code can set all levels.

Table 5-5 shows how to set the thread priority NOPs.

Table 5-5. Thread Priority Nops (Sheet 1 of 2)

Priority nop/mtSPR	PPR[11:13]	Thread Priority	Required Privilege Level to Set Given Thread Priority Value
or 31,31,31 /mtPPR[11:13]	'001'	Very Low	Hypervisor, Supervisor, Problem
or 1,1,1 / mtPPR[11:13]	'010'	Low	Hypervisor, Supervisor, Problem
or 6,6,6 / mtPPR[11:13]	'011'	Medium Low	Hypervisor, Supervisor, Problem
or 2,2,2 /mtPPR[11:13]	'100'	Medium (Normal)	Hypervisor, Supervisor, Problem

1. See *Section 5.12 Priority Boosting to Medium-High in User Mode* on page 154

Table 5-5. Thread Priority Nops (Sheet 2 of 2)

Priority nop/mtSPR	PPR[11:13]	Thread Priority	Required Privilege Level to Set Given Thread Priority Value
or 5,5,5 / mtPPR[11:13]	'101'	Medium High	Hypervisor, Supervisor, Problem[1]
or 3,3,3 /mtPPR[11:13]	'110'	High	Hypervisor, Supervisor
or 7,7,7 / mtPPR[11:13]	'111'	Extra High	Hypervisor

1. See *Section 5.12 Priority Boosting to Medium-High in User Mode* on page 154

5.11 Thread Priority Boosting

Hardware typically does not change the thread priority value in the PPR, unless an **mtPPR** or one of the priority changing NOP instructions is committed. However, on the POWER8 core, problem-state programs can change the thread priority value to medium-high ('5') depending on the contents of the Problem-State Priority Boost Register (PSPBR). The problem-state boosting changes the contents of PPR[11:13].

The thread priority can be boosted internally by the hardware (in a software invisible manner) in certain cases (as described in *Section 5.13 Thread Priority Boosting on Asynchronous Interrupt* on page 155) to medium priority ('4'). The boosting of thread priority for pending asynchronous interrupts does not affect the actual architected thread priority value in the PPR. Therefore, if the software does a **mfPPR** at any time during the asynchronous boosting, it always gets the last priority value explicitly set by the software for that thread.

5.12 Priority Boosting to Medium-High in User Mode

The POWER8 core allows a problem-state program that is executing on a thread to temporarily change the PPR thread priority value to medium high ('5') by executing an **mtPPR** or priority NOP. The temporary thread priority boosting is controlled by a 32-bit privileged Problem-State Priority Boost (PSPB) Register. There is one PSBPR per thread, which is set by a move-to PSPB.

A problem state program is able to set the program priority to medium-high only when the PSPB of the thread contains a nonzero value. The maximum value to which the PSPB can be set must be a power of 2 minus 1. Bits that are not required to represent this maximum value must return '0's when read, regardless of what was written to them.

When the PSPB of the thread is set to a value less than its maximum value but greater than '0', its contents decrease monotonically at the same rate as the SPURR until its contents minus the amount it is to be decreased are '0' or less when a problem state program is executing on the thread at a priority of medium high.

When the contents of the PSPB minus the amount it is to be decreased are '0' or less, its contents are replaced by '0'. When the PSPB is set to its maximum value or '0', its contents do not change until it is set to a different value.

Whenever the priority of a thread is medium high and either of the following conditions exist, hardware changes the priority to medium:

- PSPB counts down to '0', or
- PSPB = 0 and the privilege state of the thread is changed to problem state (MSR[PR] = '1')

While in problem state at medium-high priority, there can be the potential of the PSPBR reaching '0' at the same time a priority NOP or **mtPPR** is trying to lower the thread priority to a value less than medium. If the attempted write to the PPR occurs in the same cycle, the priority NOP or **mtPPR** must update the PPR with its thread priority instead of allowing the PSPB reset to set the PPR to medium priority.

5.13 Thread Priority Boosting on Asynchronous Interrupt

If a thread has a priority less than medium ('4'), the priority of the thread is boosted on a pending asynchronous interrupt. This allows the interrupt to be serviced faster for a thread (that is waiting for the interrupt at a low-priority state). TSCR[12] is used to enable or disable priority boosting for any pending asynchronous interrupt. The boosting of thread priority does not affect the actual architected thread priority value in the PPR. Therefore, if the software does a move from PPR (**mfPPR**) at any time during asynchronous boosting, it always gets the last priority value explicitly set by the software for that thread.

5.13.1 When to Boost Thread Priority

Thread priority is boosted internally by the hardware on an asynchronous interrupt based on *Table 5-6* on page 155. After the priority is boosted, the hardware continues to treat the thread at medium ('4') priority, until there is an **mtPPR** or priority NOP instruction that changes the thread priority.

Table 5-6. Asynchronous Interrupt

Interrupt	MSR Bits
Debug	EE = 1 or HV = 0 or PR = 1
Hmaintenance	EE = 1 or HV = 0 or PR = 1
External	EE = 1
Perfmon	EE = 1
HDEC	EE = 1 or HV = 0 or PR = 1
DEC	EE = 1
System Reset	Always (ignores TSCR[12])

In other words, there is no thread priority boosting for:

- Debug, Hmaintenance, or HDEC interrupt, if in hypervisor with EE = '0'
- External, Perfmon, or Decrementer interrupt, if EE = '0'

Otherwise, boost the priority on a pending interrupt. The reasons for not boosting the priority in the previous cases include:

- Operating system and hypervisor: Spinning on a lock, the priority is low and MSR[EE] = '0'. Priority must not be boosted because nothing useful is going to happen until the lock is acquired. Before the **stdcx** can get the lock, high priority is asserted. Therefore, if a thread is holding a lock, its priority does not need to be boosted when an external interrupt becomes pending.
- Operating system interrupt handler running with MSR[EE] = '0'. Priority is already at the desired level as a result of the implicit or explicit boost. No additional boost is required by the hardware.

5.14 Thread Prioritization Implementation

5.14.1 Thread Switch Fetch Priority

The thread priority is used to apportion fetch cycles. For example, if two threads have a priority weighting that is different, the ratio of those two weights determines the relative number of cycles those two threads will be given access to the I-cache. If all threads have equal priority, the threads are accessed in a round-robin manner.

The instruction fetcher makes no priority provisions for an asymmetric SMT environment. For example, if one side of the core has one thread, and the other side has more then one thread, and all threads are of equal priority, then each thread gets an equal number of fetch cycles, even though there are more decode resources available for the thread that is on its own side.

If all of the threads have the same priority, the fetcher fetches the threads in an order that tries to swap from one core side to the other as much as it can. This is desirable, because fetching multiple cycles on the same core side increases the chance that no instructions are available to decode/dispatch on the other core side.

Normally a thread uses its fetch cycle if there is a chance that the fetch can result in a transfer. There are several cases where a thread relinquishes its fetch cycle and allows it to be skipped over. The cases where this happens are as follows:

- The thread is an I-cache miss pending or I-ERAT miss pending
- The IBuffer is full for eight cycles, such that it is unlikely that there is room in the IBuffer when the instructions are fetched from the I-cache.
- There is a hold fetch from either the ISU or the pervasive core unit that tells us we cannot fetch on that thread.

When all the highest priority threads give up their cycle, there are times when the base hardware algorithm is not able to assign another thread based on the priority. On the cycles when this occurs, the other threads that are able to be fetched take turns fetching, ignoring the thread priority.

If there is a flush on a thread and that thread is not already being selected, that thread is selected on the next IFM1 cycle, which is done to reduce the average latency on a flush.

5.14.1.1 SMT2 Fetch Pattern

In SMT mode, the target of a predicted taken branch can be fetched three cycles after the branch instruction is fetched. If threads are alternated in SMT2 mode, the earliest time that an instruction could be fetched would be allocated to the other thread, and thus the taken branch penalty goes from three to four cycles.

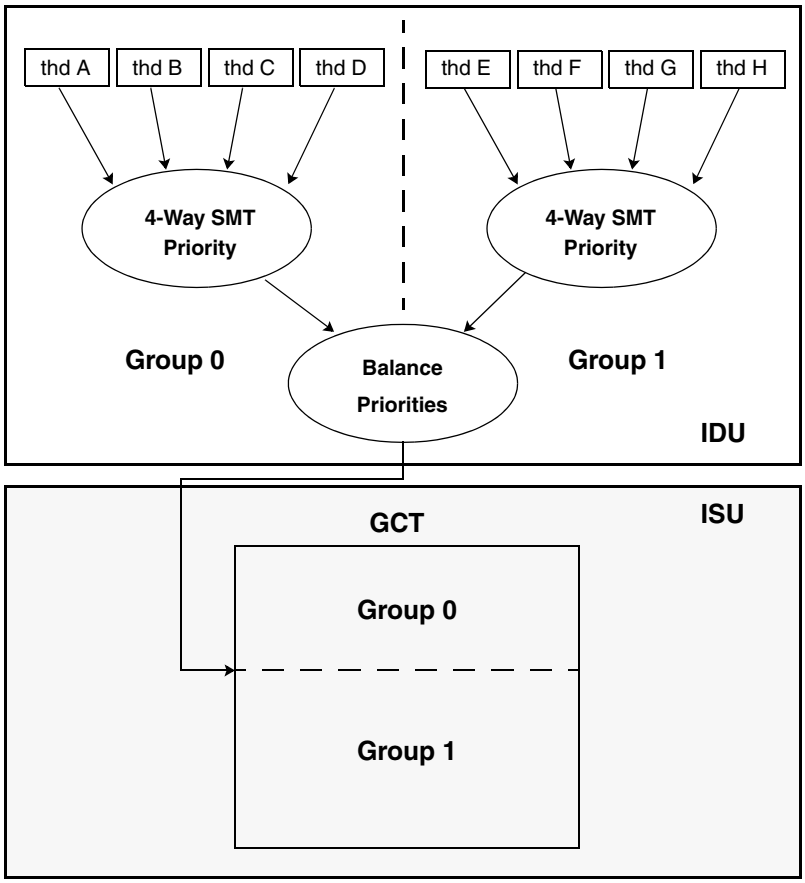
To reduce this effect, use a pattern in SMT2 mode that in most cases allows the same thread to be allocated every third cycle. The pattern implemented in SMT2 mode is '00100100 11011011'. This pattern causes the same thread to be assigned three cycles later 87.5% of the time.

5.14.2 Thread Switch Decode Priority

The instruction buffer, decode, and dispatch are divided into two parallel groups in SMT2, SMT4, and SMT8 modes. Each group gets half the dispatch width. In ST mode, the single thread gets the entire dispatch width. This drives the need for a pair of decode priority engines to run in parallel; one for the threads assigned to thread set 0 and one for those assigned to thread set 1. In SMT modes, each thread set operates in its own half of the dispatch group, independent of the other thread set.

To support two independent thread sets dispatching in parallel, two SMT4 thread priority engines are used. One controls the decode cycles of the threads assigned to thread set 0. The other controls thread set 1. Each engine can manage 1 - 4 threads, depending on how many are assigned to the thread set. Additionally, the decode cycle selections of the two engines are balanced by controlling the number of GCT entries each thread set has access to. The GCT limits are set based on the relative values of the sum of thread set 0 priorities and the sum of thread set 1 priorities. As one side of the GCT fills up, the ISU naturally throttles back that thread set on the dispatch hold interface. This achieves SMT2, SMT4, and SMT8 thread management at the system level, while allowing two thread sets to decode and dispatch in parallel.

Figure 5-1. Dual SMT4 Decode Priorities



In SMT mode, decode cycles (opportunities to form instruction groups out of the IBuffer) are given to a thread based on the following ordered criteria:

1. Thread enabled, no slots given to a stopped thread, CTRL[8:15].

2. Per thread decode stops for Decode Hold. See *Section 5.16 Controlling the Flow of Instructions in SMT* on page 161.
3. Instruction availability in the threads IBuffer. Used only if the priority in PPR[11:13] is equal for all enabled threads.
4. Software-set thread priority, controlled in Thread Status Register (PPR[11:13]). See *Section 5.14.3 Software-Set Thread Priority* on page 158.
5. Dynamically changing thread decode priority. See *Section 5.14.5 Dynamic Thread Priority* on page 159.

The first three criteria are only used to eliminate threads from consideration for the next decode cycle. If all available threads are eliminated based on those three criteria, no thread forms an instruction group next cycle. Otherwise, the remaining eligible threads are considered for the next decode cycle according to either their software-set thread priority or a dynamic thread priority algorithm using the current state of each eligible thread.

5.14.3 Software-Set Thread Priority

Software-set priority is used to determine the thread to receive the next decode cycle if at least one of the enabled threads has a different Thread Priority value in PPR[11:13] than the other enabled threads. The intention for the software-set priority algorithm is to divide decode cycles according to the relative values of the thread priority values.

POWER8 core control is given by allowing the user to define the weightings between the seven priorities. A 64-bit spr, Relative Priority Register (RPR), is provided for the user to set any 6-bit value (0 - 63), for each of the seven priority levels (very low, low, medium low, normal, medium high, high, extra high). Then, PPR[11:13] for each active thread determines which value to read from the RPR.

Each active thread receives a number of decode cycles, relative to the other threads, equal to their priority values. For example, within thread set 0, T0 has a relative priority of 17 (as defined by PPR[11:13] and the RPR), T3 has a relative priority of 6, and T7 has a relative priority of 45. Within a window of 17+6+45 = 68 decode cycles, T0 gets 17 cycles, T3 gets 6 cycles, and T7 gets 45 cycles. The pattern repeats every 68 decode cycles. Additionally, the cycles given to each thread are distributed as evenly as is reasonably possible within the pattern.

5.14.4 Low-Power Modes for Application

The POWER8 core slows the rate of group formation and decodes to reduce power whenever all enabled threads have a priority of '001', as set in each thread's PPR[11:13]. This has different effects depending on the SMT mode (CTRL[24:27]).

In ST mode, the single thread receives one decode cycle every 128 cycles.

In SMT2 mode, the threads in Thread Set 0 and Thread Set 1 each receive one decode cycle every 128 cycles. Thread Set 0 and Thread Set 1 are independent, and their decode cycles are not guaranteed to be concurrent or non-concurrent.

In SMT4 and SMT8 modes, one thread from Thread Set 0 receives one decode cycle every 128 cycles, similarly for Thread Set 1. As in SMT2, the two thread sets operate independently. Within a given thread set, the threads are queued in a modified round robin fashion. Every 128 cycles, a thread is selected from the queue and given a decode cycle. The thread at the head of the queue receives the next available decode cycle

unless its IBuffer is empty or it is otherwise removed from eligibility (ISU applies an IBuffer hold, and so on). If the thread is ineligible, the next thread in line is selected. After a thread receives a decode cycle, it is moved to the back of the queue.

5.14.5 Dynamic Thread Priority

If all enabled threads have the same thread priority value, a dynamic thread priority algorithm based on the state of the eligible threads determines which will get the next decode cycle. This algorithm uses a scoring system built on the number of GCT entries occupied by each thread, whether the thread has any outstanding L2 or TLB misses, and a final round-robin adder used only to break any ties between threads. This algorithm is implemented twice, once per thread set, where each algorithm manages 1 - 4 threads. The two thread sets are then balanced using the GCT as described in *Section 5.14.2 Thread Switch Decode Priority* on page 157.

Table 5-7 lists a summary of the scoring system.

Table 5-7. Scoring System Summary

Description	Score Adder
If a thread occupies fewer GCT entries than the threshold set in TSCR[0:1]	+10
If TSCR[0:1] = '00'	+0
If TSCR[17] = '1' and the thread has no outstanding L2 misses,	+5
If TSCR[18] = '1' and the thread does not have an outstanding TLB miss	+5
A final value is added based on the thread's position relative to a round-robin pointer:	
Thread being pointed to	+4
Thread that shares downstream resources with the pointed thread (thread0 shares with thread1, thread 2 with thread3)	+3
Thread with the same LSB as the pointer (thread0 with thread2, thread1 with thread3)	+2
Remaining thread	+1

The eligible thread with the highest overall score is given the next decode cycle. Note that the round-robin pointer only affects results in the event of a tie from the other three adders. To ensure fairness between threads when one or more threads are disabled, the round-robin pointer rotates between all threads available in the current SMT mode regardless of whether the thread is enabled.

5.15 Support for Multiple LPARs

In cloud computing mode, the core can be partitioned into 2 or 4 LPARs. When in 2 or 4 LPAR mode, the SMT mode is always forced to SMT8, so that most resources are allocated based on *Table 5-2 Front-End Execution Core Resource on page 145.*

5.15.1 Instruction Fetch

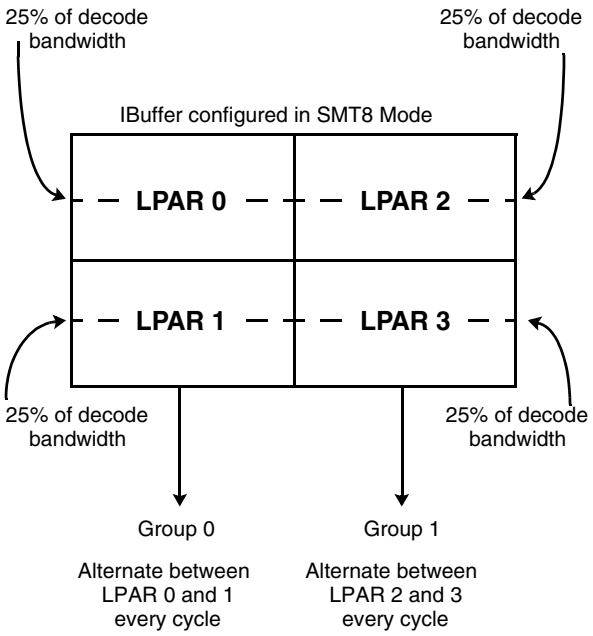
When the processor is configured in 2 or 4 LPAR mode, fetch cycles are allocated in a round robin manner to the partitions. This implies each partition will always get an equal percentage of the fetch cycles. If a partition does not have any threads that are ready, those cycles will be allocated to another partition, as long as other partitions have threads that are ready to fetch. If a partition contains multiple threads, the partition's fetch cycles for the threads in that partition are divided based on the relative software priority weighting.

5.15.2 Decode

In 2 LPAR mode, each partition gets 50% of the decode bandwidth. In 4 LPAR mode, each partition gets 25%. Each LPAR is given its 25% or 50% of the decode bandwidth, regardless of how many LPARs are active.

For 4 LPAR mode, one or two threads can be active. If an LPAR has two threads active, an SMT2 priority is followed as described in *Section 5.14.3 Software-Set Thread Priority* on page 158. Each thread is given cycles within its partitions time slice, relative to the other thread in its partition, based on their relative priorities. See *Figure 5-2* for details. For 2 LPAR mode, an SMT4 priority is followed.

Figure 5-2. Decode Priority in 4 LPAR Mode



5.15.3 Microcode Fairness

In multi-LPAR mode, the goal is to give each LPAR the same number of dispatch cycles. However, multi-cycle microcode instructions can cause an LPAR to consume multiple consecutive cycles. To remedy this, give the other LPAR sharing the dispatch bandwidth extra decode cycles to compensate for the loss of decode cycles. When microcode operations are in flight, each operation generates 32 groups (such as, load multiples).

The counter handles 32 × 12 = 384 catch-up cycles for either LPAR. A 10-bit counter handles up to 512 cycles for each LPAR.

5.15.4 Instruction Cache

I-cache allocation is altered for LPAR modes.

5.15.5 Thread Set Allocation

Threads 0 - 3 are allocated to side 0, and threads 4 - 7 to side 1, for both 2 LPAR or 4 LPAR mode. Thread set reconfiguration is disabled.

5.15.6 Data Cache

Threads 0 - 1 reload into sets 0 - 1, threads 2 - 3 reload into sets 2 - 3, threads 4 - 5 reload into sets 4 - 5, and threads 6 - 7 reload into sets 6 - 7

5.15.7 ERATs

Threads 0 - 3 share half of the primary/secondary ERAT, while threads 4 - 7 share the other half.

5.16 Controlling the Flow of Instructions in SMT

The ability to control the flow of instructions in an SMT processor is very important for the performance improvement due to SMT. When one thread is not making good progress (which can happen due to many reasons, such as an L2 miss, TLB miss, **sync** or other long latency operations), the other thread must be allowed to have as much of the machine resources as necessary for it to make progress. The following features are built into the POWER8 core for controlling instruction flow.

5.16.1 Dispatch Flush

A dispatch flush is a low-latency flush that flushes the decode pipe. **Sync**, **lwsync**, **ptesync**, **tlbsync**, **tlbie**, and instructions with the scoreboard bit set can cause a dispatch flush on the POWER8 core. Also, if enabled, a thread that was balanced flushed is dispatch flushed if the chosen thread is stalled at dispatch.

5.16.1.1 Dispatch Flush Rules

1. Dispatch flushes are disabled if the core is in single-thread mode, or if the core is in SMT2 or SMT4 mode and there is one or fewer than one threads active.
2. Dispatch flush only occurs if a thread shares a group with another thread. SMT2 mode should not dispatch flush if each thread has its own group.
3. If for some reason, both threads are on group0, (not balanced), dispatch flush can still occur.
4. Never dispatch flush a thread if it is in the middle of a microcode. Dispatch flushes related to smt-performance are never done if in the middle of a microcode dispatch. Other dispatch flushes can happen to microcode, however, such as quiesce, RAS, or a forward-progress timeout.
5. If a thread's virtual LRQ or virtual SRQ is full, it gets dispatch flushed, irrespective of its thread priority.
6. A **sync**, **lwsync**, **ptesync**, **tlbsync** instruction from thread A causes a dispatch flush of thread A (similarly, for other threads).
7. If the GCT thread is not empty, the **tlbie** instruction gets dispatch flushed. The instruction following the **tlbie** is dispatch flushed if the **tlbie** instruction has not received tlbie acknowledge from the Nest, through the LSU.
8. If thread A has its scoreboard bit set (such as, non-renamed **mtspr** followed by **mfspr**), thread A is dispatch flushed (similarly, for other threads).

9. The instruction following an instruction that stalls dispatch until GCT empty can cause a dispatch flush. Instructions that stall dispatch until GCT empty include: **mtsle**, **mtamr**, **treclaim**, and power mode instructions: doze, nap, sleep, rvwinkle, sp_attn, waitasec.
10. If TSCR[3] is enabled, dispatch flush the thread that was chosen for a balance flush if that thread is stalled at dispatch.
11. If a thread is stalled at dispatch because its unified queue half is full and the core is in SMT4 mode, then dispatch flush the stalled thread.

5.16.1.2 Stall at Dispatch

A thread can be *stalled at dispatch* due to unavailability of a shared resource that it needs for the next dispatch. When *stalled*, dispatch_hold is asserted to hold the decode pipe. The complete list of *stall* conditions follows:

- Required issue queue entries are not available
- Not enough renames are available for the entire group
- GCT is full

Note:

1. A 5-cycle delay is expected between counting the GCT entries and taking any action based on that count.
2. A 4-cycle delay is expected between the detection of a stall condition and causing a dispatch flush.

5.16.2 Decode Hold

1. After a dispatch flush, a **sync**, **lwsync**, **ptesync**, **tlbsync** instruction is held at IBUF until its dispatch conditions are met. This is done irrespective of thread priority.
2. After a dispatch flush, a **tlbie** instruction is held at IBUF until the GCT thread is empty. After the **tlbie** instruction is dispatched, the very next instruction gets dispatch flushed and then held at IBUF until GCT and the SRQ are empty. This is done irrespective of thread priority.
3. After a balance flush due to an L3 or TLB miss, instructions on the balance flush thread are held at IBUF until the miss is resolved or until the balance flush miss counter reaches the threshold value as determined by TSCR[8:9].

5.16.2.1 Balance Flush

A balance flush is an NTC+1 flush that flushes all instructions on a selected thread that are younger than the next-to-complete instruction group. It flushes the execution units, GCT, and EAT for the selected thread. Threads are considered for a balance flush only if a thread is stalled at dispatch. Balance flushes can be disabled using TSCR[2].

Criteria for Selecting a Thread to be Balanced Flushed

If the core is in SMT mode and more than one thread is active, then on a dispatch stall perform the following steps:

1. Select the threads with any number L3 or TLB misses, regardless of the balance flush miss counter value. If enabled by a debug switch, only select the threads with any number of L3 or TLB misses, if the balance flush miss counter for the thread is less than the counter threshold as described by TSCR[8:9]. If

the miss counter for the thread is greater than the threshold value, ignore the miss on that thread and do not consider the thread for a balance flush.

2. If only one thread has an L3 or TLB miss, select that thread to be balanced flushed. Raise the CLB hold on the thread that was chosen to be balanced flushed until either the miss has been resolved or the balance flush miss counter threshold has been reached as described by TSCR[8:9].
3. If in SMT4 or SMT8 and more than one thread is eligible to be balanced flushed based on having an L3 or TLB miss and a GCT count over the balance flush GCT threshold TSCR[4:5], select all eligible threads to be balanced flushed. Raise the CLB hold on the threads that were chosen to be balanced flushed until either the miss has been resolved or the balance flush miss counter threshold for that thread has been reached as described by TSCR[8:9].
4. If in SMT2 and both threads are eligible to be balanced flushed based on having an L3 or TLB miss and a GCT count over the balance flush GCT threshold TSCR[4:5], do not balance flush either thread. Do not raise the CLB hold.
5. If no threads have an L3 or TLB miss, select the threads with GCT counts over the TSCR[6:7] balance flush no-miss GCT threshold to be balanced flush. For the POWER8 processor, the default is to allow this option in SMT2, with a mode to disable it. It can be disabled for all SMT modes by setting TSCR[6:7] to '00'.
6. If the thread that is stalled at dispatch is also the thread that was chosen to be balanced flushed, then also do a dispatch flush on that thread if TSCR[3] is enabled. Otherwise, do only a balance flush on the chosen thread.

5.17 Dynamic Thread Transitioning

5.17.1 Overview

Any thread can be run in any SMT mode, on any thread set. Starting and stopping a thread, without crossing an SMT mode boundary (SMT1/2/4/8), can be done seamlessly without having to quiesce the core. However, if an SMT boundary is crossed, a quiesce (and optionally additional action) is required, as detailed in *Section 5.17.3 SMT Mode Boundary Crossings* on page 164. Note that the POWER8 processor treats the DOZE instruction as a NAP instruction.

5.17.2 Thread Set Definition

If only one thread is running, it is guaranteed to be on thread set 0. As threads are added, they are assigned to alternating thread sets, to strive to achieve balance in the number of threads per thread set. The GCT sets are tied to the thread sets and can complete one thread per cycle from each thread set. The dispatch groups are also tied to the thread sets. As threads are disabled, the thread set definition remains the same. Consequently, the remaining enabled threads can have imbalanced thread sets, which is addressed in *Section 5.17.4.1 Balancing* on page 164.

5.17.3 SMT Mode Boundary Crossings

When SMT mode boundaries (SMT1/2/4/8) are crossed, different resources throughout the core are reconfigured to maximize performance in a certain SMT configuration. All threads must be quiesced before any reconfiguration can occur.

Table 5-8 on page 164 shows how the different resources must be reconfigured in the various SMT mode boundary crossings. If multiple boundaries are crossed simultaneously, the action is the OR of the respective boundary-crossing functions listed in *Table 5-8*.

Table 5-8. SMT Mode Boundary Crossing Reconfigurations

From	To	Drain AMCs	Reconfigure IFU	Reconfigure IDU	Reconfigure LSU	Notes
SMT1	SMT2			Y	Y	
SMT2	SMT4		Y			
SMT4	SMT8		Y			
SMT8	SMT4		Y			
SMT4	SMT2		Y			
SMT2	SMT1	Y		Y	Y	Place remaining thread in thread set 0.

It is possible that, when an SMT mode is lowered, there might be too many threads in one of the thread sets, or the number of threads per thread set might be imbalanced. *Table 5-1 SMT Modes* on page 145 shows the maximum allowable number of threads per thread set. The corrective action is to reconfigure thread sets described in *Section 5.17.4 Thread Set Reconfiguration*.

5.17.4 Thread Set Reconfiguration

Threads can be moved to a different thread set, for a variety of reasons. To do so, all threads must be quiesced.

5.17.4.1 Balancing

There can be an imbalance in the number of threads per thread set, either after an SMT mode change, or as threads are disabled. *Table 5-9* shows the different scenarios where a rebalance can be triggered. Some of the scenarios can only occur after an SMT mode change because once in a given SMT mode, there is a maximum allowable number of threads per thread set, as shown in *Table 5-1 SMT Modes* on page 145. Otherwise, the scenario can occur at any time.

Table 5-9. Thread Balancing Scenarios (Sheet 1 of 2)

Mode	Threads Per Set	Rebalanced Threads Per Set
SMT2	2/0	1/1
SMT4	2/0	1/1
SMT4	3/0	2/1
SMT4	3/1 4/0	2/2
SMT8	2/0	1/1
SMT8	3/0	2/1

Table 5-9. Thread Balancing Scenarios (Sheet 2 of 2)

Mode	Threads Per Set	Rebalanced Threads Per Set
SMT8	3/1 4/0	2/2
SMT8	4/1	3/2
SMT8	4/2	3/3

5.17.4.2 Mixing

There can be an inequality in the processor resources being used per thread set. If one thread set is heavily using a given resource, but the other thread set is not, wasting its allotment of the resource, it might be advantageous to move a thread using the resource to the other thread set, to maximize the usage of the resources. A reconfiguration based on this thread "mixing" can only occur in SMT4 and SMT8 modes.

5.17.4.3 Action

When moving a thread to the opposite thread set, its AMC data must be drained, pushing that thread's GPR/VRF data out to the SAR, so that it can be reloaded into the opposite register file after it starts executing. If the action is triggered while executing, only perform the action after the trigger remains in the same state after a delayed amount of time, controlled by a programmable 20-bit LFSR counter (up to 1 million cycles). This delay is inserted because a reconfiguration is expensive in terms of latency. Therefore, it is important that the events that caused the trigger are in a relatively steady state. The link stack and D-ERAT must be rebuilt for the moved threads; therefore, performance initially suffers for those threads.

6. POWER8 SMP Interconnect

The POWER8 SMP interconnect fabric controller (FBC) is the underlying hardware used to create a scalable cache-coherent multiprocessor system. The POWER8 SMP interconnect controller provides coherent and noncoherent memory access, I/O operations, interrupt communication, and system controller communication. The FBC provides all of the interfaces, buffering, and sequencing of command and data operations within the storage subsystem. The FBC is integrated on the POWER8 chip, with twelve processor cores and an on-chip memory subsystem.

The POWER8 chip has up to three FBC links that can be used to connect to other POWER8 chips. The FBC link is a split-transaction, multiplexed command and data bus that can support up to four POWER8 chips. The bus topology is a fully-connected topology to reduce latency, increase redundancy, and improve concurrent maintenance. Reliability is improved with ECC on the external I/Os.

Cache coherence is maintained by using a snooping protocol. Address broadcasts are sent to the snoopers, snoop responses are sent back in-order to the initiating chip, and a combined snoop response broadcast is sent back to all of the snoopers. Multiple levels of snoop filtering, Chip pump, and remote chip pump, are supported to take advantage of the locality of data and processing threads. This approach reduces the amount of interlink bandwidth required, reduces the bandwidth needed for system-wide command broadcasts, and maintains hardware enforced coherency using a single snooping protocol. Chip pump limits the command broadcast scope to the snoopers found on a physical POWER8 chip. When the transaction cannot be completed coherently using this limited scope, the coherence protocol forces the command to be re-issued to all chips in the system (system pump). Conversely, remote chip pump limits the broadcast scope to a remote POWER8 chip; if the operation cannot complete coherently, the command is re-issued using SystemPump to complete the operation.

6.1 POWER8 SMP Interconnect Features

6.1.1 General Features

- Master command/data request arbitration.
- Command requests are tagged and broadcast using a snooping protocol, enabling high-speed cache-to-cache transfers.
- Multiple command scopes are used to reduce the bus utilizations system wide. The SMP interconnect controller uses cache states indicating the last known location of a line (sent off chip), information maintained in the system memory (snoop filter bits), a coarse-grained MCD indicating when a line has gone off the chip, and combined response equations indicate if the scope of the command is sufficient to complete the command or if a larger scope is necessary.
- The command snoop responses are used to create a combined response, which is broadcast to maintain system cache state coherency. Responses are not tagged. Instead, the order of commands from a chip source, using a specific command broadcast scope, is the same order that combined responses are issued from that source.
- Data is tagged and routed along a dynamically selected path using staging/buffering along the way to overcome data routing collisions. Command throttling and retry command back-off mechanisms are used for livelock prevention.
- Multiple data links between chips are supported (link aggregation).

6.1.2 POWER8 Specific Features

- Chip pump, remote chip pump, and system pump command broadcast scopes with memory domain indicators in cache states, memory, and in the MCD to determine when an increase in the command broadcast scope is required.
- 1 - 4 socket configuration support (12-way - 48-way)
- 2x snoop bus support
- 64 local master (LM) system pump queue size (32 per snoop bus)
- 64 chip master (CM) chip pump queue size (32 per snoop bus)
- Service processor accessible SCOM registers for configuration setup

6.1.3 Off-Chip Features

- 2-byte DMI2 intergroup links (A0:A2)
- Aggregate data link support

6.1.4 Power Management Features

- Core chiplet frequency support
  - Doze mode within the floor/ceiling range
  - Doze mode outside the floor/ceiling requires change frequency command
- EX chiplet sleep mode support

6.1.5 RAS Features

- 100% ECC protection on external A-bus off-chip links
  - Single-bit error correction (incoming and outgoing data)
  - Double-bit error detection
- 100% ECC protection on internal data flow
- Livelock recovery mechanism
- Trace array
- Performance monitor
- FIR error reporting
  - Protocol errors
  - Underflow/overflow checkers
  - Asynchronous drop/repeat checkers
  - Parity checkers on coherency register files
- Error injection for system-code debug
  - Single-bit or double-bit errors on external SMP links

6.2 External POWER8 SMP Interconnect

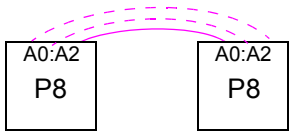
Within the SMP, the off-chip POWER8 SMP interconnect supports up to three coherent SMP links (A0:A2). The interchip A links connect up to four chips. The A links carry coherency traffic, as well as data, and are interchangeable with each other. The A links can also be configured as aggregate data-only links between groups.

The A links are configured as 2 bytes in width (A0:A2, 2-byte DMI2 intergroup links).

6.2.1 POWER8 SMP Interconnect Multichip Configurations

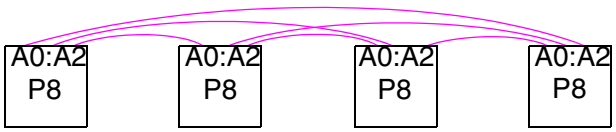
Figure 6-1 and *Figure 6-2* illustrate various POWER8 SMP interconnect multichip configurations.

Figure 6-1. Two Socket Configuration (24-way)



Note: The dotted line indicates an aggregate data-only link that only applies to the POWER8 processor in the SCM.

Figure 6-2. Four Socket Configuration (48-way)



6.2.2 Protocol and Data Routing in Multichip Configurations

The SMP ports configured for coherency are used for both data and control information transport. The use of the buses is as follows:

1. The chip containing the master that is the source of the command issues the reflected command and the combined response to all other chips in the group. The other chips direct the partial response that is a result of the reflected command back to the chip which provided the reflected command. Partial responses are collected at intermediate chips along the directed path from the partial response source to the chip containing the master. The number of partial responses received at the chip containing the master is equal to the number of SMP ports used to broadcast the reflected command.
2. Data is moved point-to-point. For read operations, the chip containing the source of the data directs the data to the chip containing the master. For write operations, the chip containing the master, directs the data to the slave that performs the write operation. Note that the routing tag contains chip and unit identifier information for this purpose.

6.3 Coherency Flow

6.3.1 Physical Broadcast Flow

There are three physical broadcast flows within the POWER8 SMP interconnect.

- Chip Pump (CP) - Command broadcast is limited to the local physical chip
- System Pump (SP) - Command broadcast is to all chips in the system.
- Remote Chip Pump (RCP) - Command broadcast is limited to the local physical chip and a remote physical chip.

Each physical broadcast flow has an independent set of coherency tracking structures (master tag FIFO, ticket counters, and so on).

6.3.2 Broadcast Scope Definition

The POWER8 broadcast terminology used in this documentation denotes the physical broadcast scope for commands. *Table 6-1* describes the physical broadcast scope and the equivalent coherency scope.

Table 6-1. POWER8 *Broadcast Scope Definition*

Coherency Command Scope	unit_pb_cmd_scope	Physical Broadcast
Chip scope	000	Chip
System scope	010	All SMP chips
Remote chip scope	011	Local/remote physical chips

6.4 Command Ordering Support

The processor bus supports the ability for a master to issue commands in a specific order per scope and for their completion to maintain the same order. The issue order is determined by the order the master issues the command request to the processor bus. The command completion is when the combined response is formed. The processor bus, however, does not guarantee the order in which commands are presented on the reflected command bus or the broadcast combined response bus.

6.5 Memory Coherence Directory

6.5.1 Directory Size

The Memory Coherency Directory (MCD) contains 2 bits per 16 MB page (1 bit for even 8 MB, 1 bit for odd 8 MB) for a total 2 TB of group LPC address space.

6.5.2 Operation

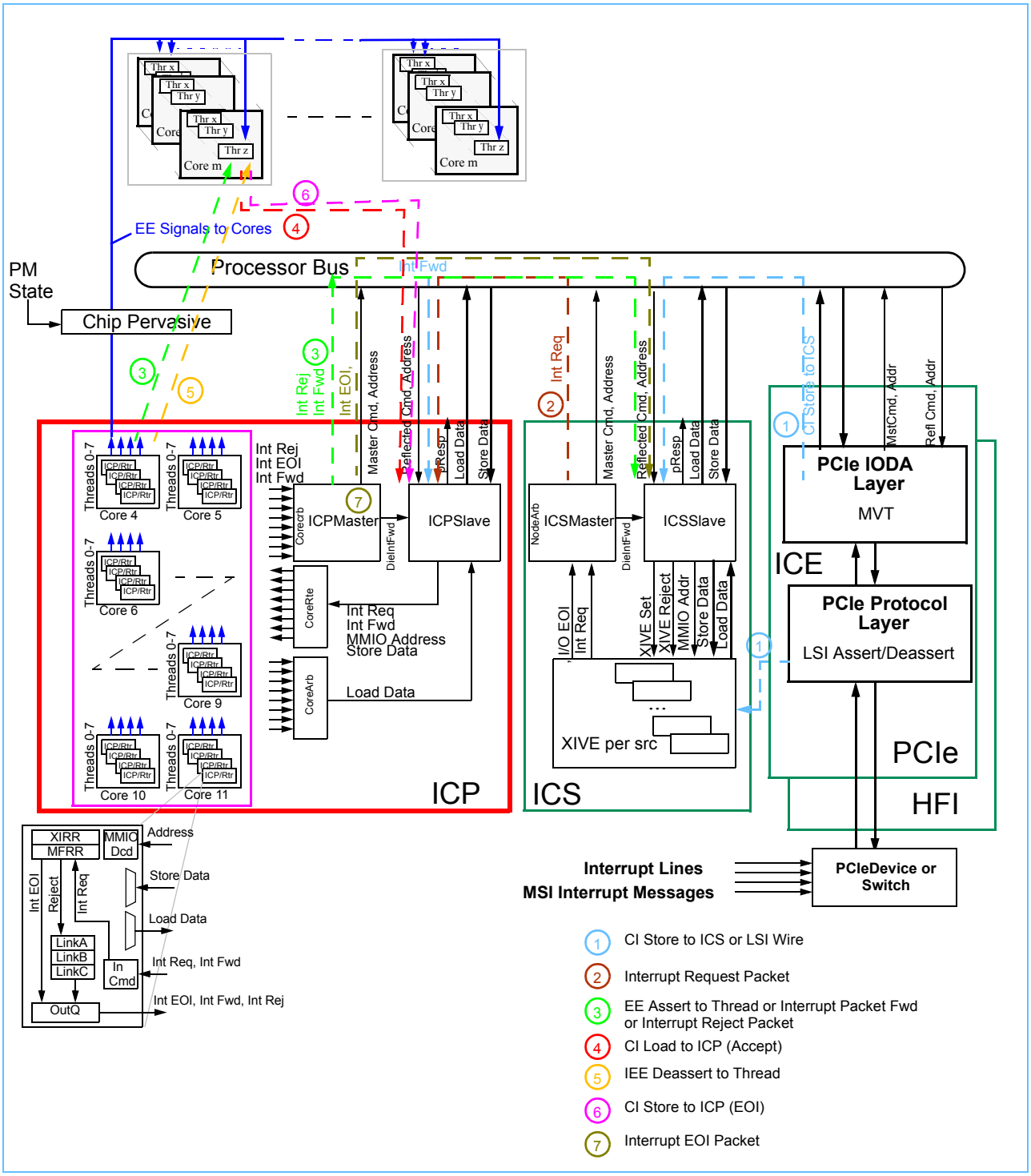
The MCD Lite is the MCD implementation for the POWER8 chip in the SCM. It is a coarse-grained directory of group scope memory domain status (MDS) bits designed to cover the full real address range of the memory controllers on the POWER8 chip. The size of memory address space covered by each bit (granule) of the MCD depends on the memory configuration. The MCD is functionally split into 2 units, called slices, each connected to one of the two processor bus ports. The two units operate independently and do not share granule state. Additionally, the MCD unit can be enabled by software to probe the memory domain status of cache lines through the processor bus command interface. The MCD can recover the on-group status of a granule if all cache lines are found on group. The MCD also has the ability for software to directly read and write any bit in the MCD arrays. For data reliability purposes, the MCD arrays are protected by ECC.

7. Interrupt Control Presenter

The interrupt control presenter (ICP) function combines the interrupt presentation and router functions. It decides which thread is to be notified of an available interrupt or returned to a source layer. The notification to a given thread is via a control wire to each PowerPC thread.

Interrupts reach the ICP via interrupt commands, issued through the processor bus, that either originate from a device attached to a PCIe bus or from an internal interrupt control source (ICS) function, depending on the system configuration. *Figure 7-1* on page 174 depicts the general interrupt structure.

Figure 7-1. POWER8 Logical Interrupt Controller Structure



7.1 Features

The interrupt presenter layer consists of the following facilities:

- Supports 96 threads (12 cores with 8 threads each) for the POWER8 processor in the SCM configuration.
- One External Interrupt Request Register (XIRR) per thread.
  - Current Processor Priority Register (CPPR).
  - External Interrupt Source Register (XISR) - interrupt source number (ISN) determines origin of interrupt.
- Most Favored Request Register (MFRR) per thread for interprocessor interrupts.
- PAPR Type II routing.
  - Three link registers for interrupt forwarding.
  - Link traversal detection creating an interrupt return.
- Continuous evaluation of priority of queued interrupt.
- Resides as a unit on the processor bus. Acknowledges (receives) interrupt request packets and produces interrupt return or end-of-interrupt (EOI) packets back to the requesting ICS or interrupt forward packets to pass the request to another ICP.
- Uses processor bus address-only transactions for interrupt requests, returns, EOIs, or forwarding.
- Interrupt source number of the processor is implemented as a 19-bit field.
- Supports internal ICS and PCIe.
- Has the following queue depths per chip.
  - 8-deep interrupt request queue.
  - 8-deep interrupt return queue.
  - 8-deep external forwarding queue.
  - 16-deep shared CI load/CI store queue.

7.1.1 Routing Layer

The IRL that handles the server number in the interrupt request acknowledges the command and passes it to the associated IPL. If the priority of the interrupt is more favored than the current operating priority, the IPL asserts the external interrupt signal to the thread and loads the interrupt into the XIRR. If the interrupt is less favored than the current priority, the interrupt is given back to the IRL. If the interrupt is a directed interrupt (also known as, specific to this thread), the IRL issues an interrupt return to the ICS for later representation.

If the interrupt is not directed, a field in the interrupt request (link pointer) identifies which of the link registers will be used for forwarding the interrupt to another thread's IRL using the interrupt forward command.

Because the link registers can be set up by software to form a ring, there is the danger that interrupts might be forwarded forever in such a loop. To avoid this, software must set the loop trip bit in one and only one of the link registers that form such a loop. When an interrupt-forward or an interrupt-request command reaches a link register, that has the loop trip bit set, the loop trip bit is carried in and passed on with all newly generated interrupt forward commands generated from that link register. An interrupt forward loop is detected when an interrupt-forward command reaches a link register that has the loop trip bit set and the command already carries the loop trip bit. If an interrupt forward loop is detected, the IRL issues an interrupt return to the ICS for later representation.

7.1.2 Presentation Layer

Each thread has an interrupt management area that contains register facilities that software uses to accept and initiate the end-of-interrupt operation. This interrupt management area is shown in *Table 7-1*. The base address (BA) is established by system firmware to locate the interrupt management area for each thread in the system address space.

Table 7-1. Interrupt Management Area: Interrupt Presentation Layer Ports

Address	Byte 0	Byte 1	Byte 2	Byte 3	Comments
BA+0	CPPR	XISR			XIRR without side effects
BA+4	CPPR	XISR			XIRR with load/store side effects
BA+8	Reserved				
BA+12	MFRR	Reserved or Unimplemented			
BA+16	LINK A				
BA+20	LINK B				
BA+24	LINK C				

The definition of these facilities are defined in *Table 7-2*:

Table 7-2. Facility Definitions

Facility Name	Description
CPPR	The Current Processor Priority Register is 8 bits in length allowing for 256 priorities values. The least favored level is x'0FF', and the most favored is x'000'. The interrupt presentation layer only signals the associated processor with interrupt conditions that are more favored than the current setting of the CPPR.
MFRR	The Most Favored interprocessor Request Register is 8 bits in length of identical format to the CPPR. Its value indicates the most favored interprocessor interrupt queued for the associated processor. If its value is more favored than the CPPR, an interprocessor interrupt is signaled to the processor.
LINK	One or more Link registers are found in only Type II interrupt presentation controllers. These registers are configured by platform code to form one or more circular linked lists of per processing unit thread interrupt presentation controllers that make up the group servicing a group server queue. The circular linked list replaces the functionality of the interrupt routing layer's Available Processor Mask used by type I interrupt presentation controllers. The Link register is composed of five fields, The length of four of these fields is implementation dependent. However, all Link registers of a given machine provide the same implementation-dependent values. All unimplemented bits read as zeros, and all unimplemented/read-only bits silently ignore writes. Below are defined the fields of the Link registers.
FLAG	The FLAG field of the Link register is a 1-bit long read-only field that defines the number of implemented Link Registers. The FLAG field of all Link registers except for the last implemented Link register FLAG is '1'. The FLAG field of the last implemented Link register FLAG is '0'.
LSPEC	The LSPEC field specifies the LINK register within the targeted interrupt presentation controller that contains the next link in the circular chain. The LSPEC field starts from the low-order bit of the LINK register (bit 31) consisting of M bits where M is the log base 2 of (the number of implemented link registers per interrupt presentation controller +1).
LOOP TRIP	The LOOP TRIP bit in the Link register tells the interrupt forward command to set this bit in the forward packet when an interrupt traverses this register. Once set, the next Loop Trip encounter causes the forwarding to cease and a return packet is generated. This is used by firmware to avoid creating open loops during partition updates.

Software accepts the interrupt condition and is responsible for resetting the interrupt condition through the operations defined in *Table 7-3*.

Table 7-3. Resetting the Interrupt Condition

Load/Store Operation	Interrupt Presentation Function
Load BA+0 (1 byte)	Poll of CPPR value.
Load BA+0 (4 bytse)	Poll of current interrupt status (source and priority).
Load BA+4 (1 byte)	Poll of CPPR value.
Load BA+4 (4 bytes)	Software accepts interrupt. This causes the deactivation of the interrupt signal to the processor thread.
Load BA+12 (1 byte)	Poll MFRR.
Store BA+4 (1 byte)	Set CPPR value.
Store BA+4 (4 bytes)	Software signals end-of-interrupt (EOI) processing. This cause the EOI to be sent to the source.
Store BA+12 (1 byte)	Software signals an interprocessor interrupt event of the priority stored.

When the interrupt presentation layer signals an interrupt to a processor-thread, it loads the XISR with the source of the interrupt. Any nonzero value in the XISR field causes an interrupt to be signalled to the processor. This signal is masked by the processor's MSR[EE] bit before the processor generates the interrupt sequence. If, at a later time, a more favored priority interrupt is made available to the interrupt routing layer, the interrupt routing layer can atomically change the value in the XISR to reflect the source of the more favored interrupt. If a more favored interrupt pre-empts a less favored interrupt in this way, the less favored interrupt is re-presented at a later time.

After the processor has read the XIRR at BA+4, the interrupt routing layer cannot change its mind and either pre-empt or cancel the request.

The XIRR facility appears twice in the external interrupt management area. Address BA+0 is designed to be used with interrupt polling. Address BA+4 has side effects when read or written, and is designed to allow efficient interrupt handler software by having the hardware assist the software in the interrupt queueing process.

The Most Favored Request Register (MFRR) holds the priority of the most-favored request queued on a software managed queue for this processor. When written to a value other than x'FF' the MFRR competes with other external interrupts for the right to interrupt the processor. When the MFRR priority is the most-favored of all interrupt requests directed to the processor, an appropriate value is loaded into the XISR and an interrupt is signaled to the processor. When the processor reads the XIRR at BA+4, the value in the MFRR is loaded by the hardware into the CPPR. The MFRR can be read back by the software to ensure that the MFRR write has been performed.

During the processing of an interprocessor interrupt, the highest priority request is de-queued by the software from the software queue associated with the MFRR and the priority of the next-favored request is loaded into the MFRR by the software.

7.2 Interrupt Control Presenter Registers

7.2.1 ICP Address Map

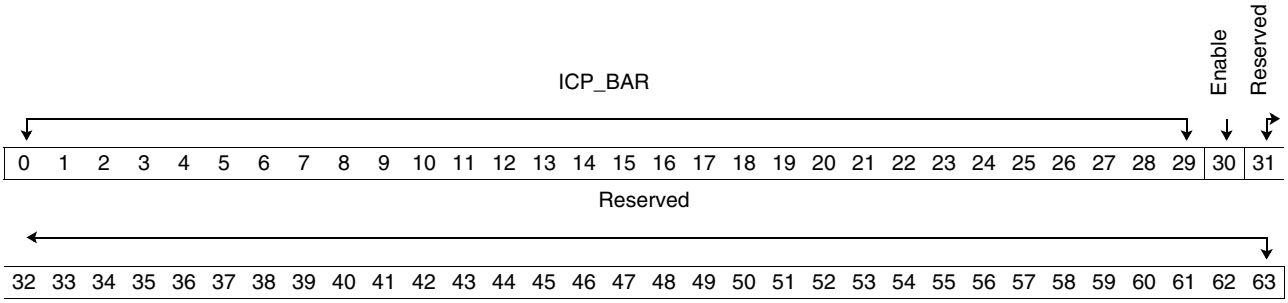
Table 7-4. Interrupt Presenter Register Address Map

Mnemonic	Register Name	Address (44:63)[1]	Length
PowerPC System Interrupt Registers: • Address (14:43) = 0xbbbb bbbb (where bbbb bbbb is from ICPBAR Register)			
XIRR	External Interrupt Request Register (CPPR and XISR) for processor thread nnnnnnn (without side effects)	0nnn nnnn 0000 0000 0000	1 byte / 4 bytes
XIRR	External Interrupt Request Register (CPPR and XISR) for processor thread nnnnnnn (with load/store side effects)	0nnn nnnn 0000 0000 0100	1 byte / 4 bytes
MFRR	Most Favored Request Register for processor thread nnnn	0nnn nnnn 0000 0000 1100	1 byte
LINKA	Link A Register	0nnn nnnn 0000 0001 0000	4 bytes
LINKB	Link B Register	0nnn nnnn 0000 0001 0100	4 bytes
LINKC	Link C Register	0nnn nnnn 0000 0001 1000	4 bytes

1. Where nnn_nnnn is defined as the following to identify the targeted thread: 45:48 - CoreID and 49:51 - ThreadID

7.2.2 Interrupt Base Address Register (ICPBAR)

Access**:** R/W/A/O SCOM only



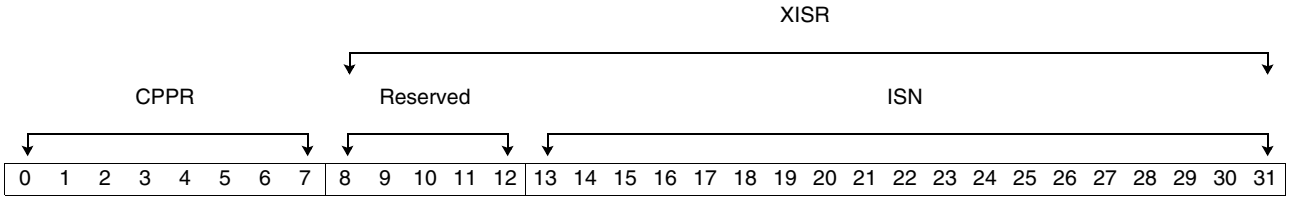
Bits	Field Name	Description
0:29	ICP_BAR	Interrupt Control Presenter Base Address Register. Contents of this register are compared to real address bits 14:43 for a match.
30	Enable	Indicates that ICP_BAR is valid.
31:63	Reserved	Reserved.

This register indicates the 1M region for the PowerPC architected presentation registers within the POWER8 processor. This register is sized to support a 50-bit physical address space.

The following registers are located at an address offset from base address (BA) established by the ICPBAR Register.

7.2.3 External Interrupt Request Register (XIRRt with t = 0 - 7)

Access: R/W (1-byte and 4-byte length to BA+4, 4-byte length only to BA+0)



Bits	Field Name	Description	Initial Value
0:7	CPPR	Current Processor Priority Register Gives the current operating priority of the processor (not the priority of the external interrupt being presented). x'FF' is lowest priority and x'00' is the highest.	x'FF'
8:12	Reserved	Reserved.	
13:31	ISN	Interrupt source number.	

This register is used to pass the ISN to the software on a read and to cause an end-of-interrupt (EOI) on a write. This register also contains the priority register CPPR. There is one register for each processor supported.

The XIRR is made up of two parts: the CPPR and the XISR. The XISR portion contains the ISN of an interrupt being presented. If the XIRR is read when there is no interrupt being presented, the XISR is all zeros.

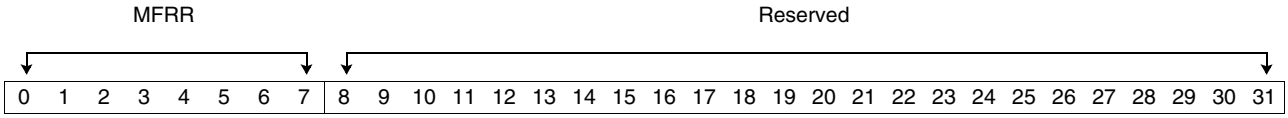
The XIRR can be read at two addresses, BA+0 and BA+4.

- A 4-byte read from BA+0 returns the current CPPR and XISR, but does not have any side effects.
- A 1-byte read from BA+0 returns the current CPPR and has no side effects.
- A 1- or 4-byte write to the BA+0 has no side effects and does not update the XIRR.
- A 1-byte write of the XIRR at BA+4 stores a new value to the CPPR. This results in an interrupt resend being broadcast on the SMP interconnect controller bus if the priority is lowered. This can also result in a pending interrupt being returned to its source with a interrupt return issued on the processor bus.
- A 4-byte write of the XIRR at BA+4 stores a new value to the CPPR. This results in an interrupt resend being broadcast on the processor bus if the priority is lowered. Software must ensure that the level being set in the CPPR is not higher than the current level. The XISR is not modified but the store data is sent as the ISN for the EOI command.

  Note: Essentially, software must store the XIRR exactly what it read from XIRR when it received the interrupt. This causes an EOI to the interrupt source and puts CPPR back to its original priority.

7.2.4 Most Favored Request Register (MFRRt with t = 0 - 7)

Access: R/W (1-byte length) BA+12



Bits	Field Name	Description	Initial Value
0:7	MFRR	Most Favored Request Register. Establishes the priority of an interprocessor interrupt. x'FF' is lowest priority, and x'00' is the highest priority.	x'FF'
8:31	Reserved	Reserved.	

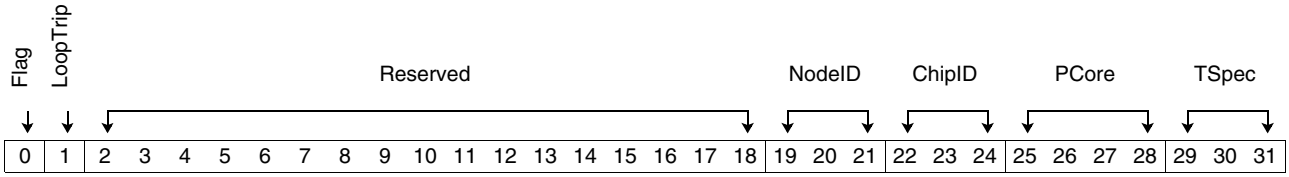
These registers are used by software to send interprocessor interrupts. There is one register for each processor/thread supported.

To send an interprocessor interrupt, software writes into the MFRR associated with the processor that it wants to interrupt. When software has written the MFRR to something other than x'FF', the POWER8 processor activates the interrupt to the associated processor if the MFRR priority is higher priority than the associated processor CPPR and higher than the priority of any pending interrupt to that processor.

When the software reads the XIRR at BA+4, the value in the MFRR and ISSR is loaded by the hardware into the XIRR. Further interrupts from this MFRR are blocked until software writes to this MFRR again. Reads of BA+4 for an interprocessor interrupt returns the ISN = 2. When no interprocessor interrupt is active, software should write a x'FF' to the first byte of this register.

7.2.5 Link Register A (LinkAt with t = 0 - 7)

Access: R/W (4 byte to BA+16)

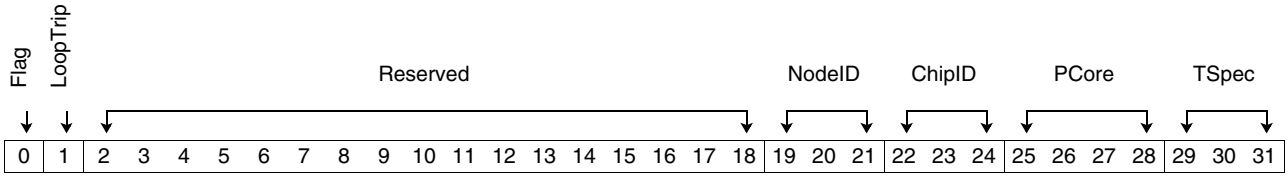


Bits	Field Name	Description
0	Flag	Indicates that the Link Register is the last one implemented of the three architected.
1	LoopTrip	Indicates that this presenter controls loop checking: 0 Do not set upon forward packet. 1 Set upon forwarded packet.
2:21	Reserved	Reserved.
22:24	ChipID	ChipID. See note 1.
25:28	PCore	Processor core that is the next link in the chain. See note 1.
29:31	TSpec	Thread Specification. Thread within the specific processor core that is the next link in the chain. See note 1.

1. If an interrupt request to the thread in question cannot be presented at this point in time, the interrupt is forwarded to the appropriate node, chip, core, or thread. Such a chain, as mentioned previously, can be created, where an interrupt is forwarded from a possible candidate (thread) to the next candidate. The current test relates to this chain, but is not very expressive.

7.2.6 Link Register B (LinkBt with t = 0 - 7)

Access: R/W (4 byte to BA+20)

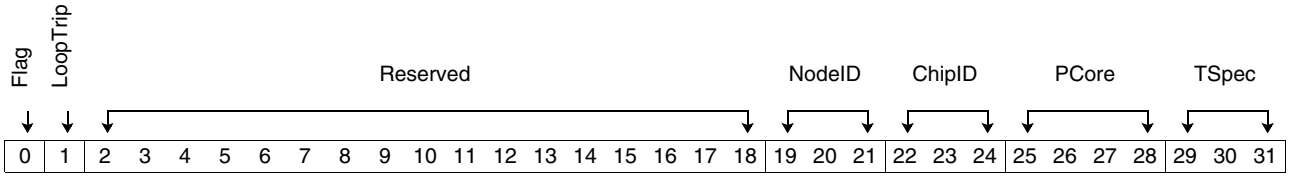


Bits	Field Name	Description
0	Flag	Indicates that the Link Register is the last one implemented of the three architected.
1	LoopTrip	Indicates that this presenter controls loop checking: 0 Do not set upon forward packet. 1 Set upon forwarded packet.
2:18	Reserved	Reserved.
19:21	NodeID	NodeID. See note 1.
22:24	ChipID	ChipID. See note 1.
25:28	PCore	Processor core that is the next link in the chain. See note 1.
29:31	TSpec	Thread specification. Thread within the specific processor core that is the next link in the chain. See note 1.

1. If an interrupt request to the thread in question cannot be presented at this point in time, the interrupt is forwarded to the appropriate node, chip, core, or thread. Such a chain, as mentioned previously, can be created, where an interrupt is forwarded from a possible candidate (thread) to the next candidate. The current test relates to this chain, but is not very expressive.

7.2.7 Link Register C (LinkCt with t = 0 - 7)

Access: R/W (4 byte to BA+24)



Bits	Field Name	Description
0	Flag	Indicates Link Register is the last one implemented of the three architected.
1	LoopTrip	Indicates that this presenter controls loop checking: 0 Do not set upon forward packet. 1 Set upon forwarded packet.
2:18	Reserved	Reserved.
19:21	NodeID	NodeID. See note 1.
22:24	ChipID	ChipID. See note 1.
25:28	PCore	Processor core that is the next link in the chain. See note 1.
29:31	TSpec	Thread specification. Thread within the specific processor core that is the next link in the chain. See note 1.

1. If an interrupt request to the thread in question cannot be presented at this point in time, the interrupt is forwarded to the appropriate node, chip, core, or thread. Such a chain, as mentioned previously, can be created, where an interrupt is forwarded from a possible candidate (thread) to the next candidate. The current test relates to this "chain," but is not very expressive.

8. PCI Express Controller

The PCI Express controller (PEC) bridges between the internal processor bus and the high-speed serial (HSS) links that drive the PCI Express (PCIe) I/O. The PEC acts as a processor bus master on behalf of the PCIe port, converting inbound memory read and write packets into processor bus DMA traffic. The PEC also acts as a processor bus slave, transferring processor load and store commands to the PCIe devices attached to the port.

8.1 Specification Compliance

The PEC is compliant with the following IBM and industry standards:

- POWER Architecture Platform Requirements (PAPR+) Specification, Version 2.1
- I/O Design Architecture v2
- PCI Express Base Specification, Revision 3.0

8.2 PEC Feature Summary

- PCIe Generation 3 Root Complex (RC)
  - Backwards compatible with Generation 1 and 2
  - 2.5, 5.0, and 8 GT/s signalling rate
- 32 PCIe I/O lanes configurable to three independent root complexes for the POWER8 SCM configuration
- Each root complex with 256 partitionable endpoints (PE) for LPAR support
- TCE based address translation for DMA requests.
  - 50-bit address support
  - Translation validation table based on PCI routing ID
- 2048 MSI interrupts per RC
- Eight LSI interrupts per RC
- IBM enhanced error handling (EEH) support
- Processor bus cache-inhibited space segmented by PE:
  - PCI 32-bit memory space segmented into 256 domains by the memory domain table
  - PCI 64-bit memory space segmented by 16 M64 BARs with 16 segments each
- Support for ECRC
- Support for lane swapping
- Support for TLP hints

8.2.1 Supported Configuration

The 32 lanes of HSS I/O can be configured to support three independent PCI buses. *Table 8-1* describes the maximum lane allocation. In addition to supporting PCI operations, the HSS I/O can be allocated for use by the processor bus SMP interface.

Table 8-1. Supported I/O Configurations

PEC0	PEC1	PEC2
16	16	Unused
16	8	8
8	16	Unused
8	8	8
Unused	16	Unused
Unused	8	8

9. Power Management

9.1 Overview

The POWER8 processor uses a number of the more traditional dynamic power-saving techniques, such as clock gating latches and arrays when they are not needed. These techniques make it possible to reduce peak power and therefore, thermal design point power (TDP). They also make it possible to dynamically power gate (turn the power off to) individual cores or full core chiplets when the core is not being used.

The POWER8 processor has an adaptive power management technique to reduce average power, collectively known as EnergyScale™, to proactively take advantage of variations in workload, environmental condition, and overall system utilization. This, coupled with a policy direction from both the customer and feedback from the Hypervisor/operating system that is running on the machine, is used to determine modes of operation and the best power and performance trade-off to implement during runtime to meet customer goals yet achieve best possible performance.

The POWER8 processor extends support to enable:

- Micropartition energy management by providing more synchronous input to the hardware platform upon LPAR switches through hypervisor control Pstate (performance states)
- Faster adaptive monitoring and actuation through the on-chip controller (OCC) (one per POWER8 die)
- Better instrumentation and control for memory power management (including partition-level memory monitoring and throttling)

Managing the power and performance trade-off is a complex problem. There are many ways to control the behavior of the hardware, but these also have a number of side effects, all of which vary based on the workload being processed. Because there is no "one-size fits all" policy that can be implemented, the POWER8 processor supports an adaptive approach to the problem in the form of a joint hardware, firmware, and software solution.

9.2 Power Management Infrastructure

The POWER8 processor supports a hierarchical solution to power management. An entity running on an attached service processor, with management software running on the cores, can establish power budgets. Then, the POWER8 processor can be required to stay within the budget. Hypervisor-based energy management algorithms for the partition and micropartition level can then influence the power/performance trade-off in conjunction with the on-chip controller (OCC), which deals with the processor and memory level. Features are handled at the lowest level possible to allow the greatest flexibility and to reduce overall complexity of the hardware design. In general, power management hooks exist inside the processor core itself, inside the processor core chiplet (the asynchronous entity which includes the core's L2 and L3 caches), in the chip-level nest unit level, and at the chip level. This affects how the features are implemented and therefore, laid out in the SCOM registers. For example, voltage is controlled at the chiplet level (as well as the chip), frequency and power gating in each core chiplet, and software-directed modes and instruction throttling controls inside the core itself.

9.3 Power Management Policies and Modes of Operation

The POWER8 interface must be simple but powerful to observe the operating power of the POWER8 system and its subcomponents and to direct overall policies and modes of operation for the system. The POWER8 power management subsystem (EnergyScale hardware and firmware) must provide clear policies that can be customized to achieve the desired level of power and performance efficiency, within bounds specified by the customer. The POWER8 processor supports multiple power management choices for the system operation. They can be selected by the customer, depending on the situation at any given time or for particular data center constraints.

9.3.1 Maximum Power Savings Based on Utilization and Idle

With this policy,[1] decisions are made by the EnergyScale and hypervisor firmware to take advantage of predictable idle periods of low-processor utilization. No performance (throughput) loss is allowed during long periods of invariant utilization percentage in the observable seconds or minutes timescale. This mode is tailored to satisfy the SPEC-Power benchmark, various upcoming government regulations, and some predictable data center usage models. In this policy, power is not reduced during periods of near 100% utilization, because full performance is maintained.

9.3.2 Adaptive Power Savings with Performance Loss Floor

This policy attempts to save as much power as possible with a maximum defined allowable performance loss. This policy is useful for system workloads or periods of time with high or unpredictable utilization. An attempt is made by the EnergyScale firmware and hardware to save as much power as possible while maintaining at least a minimal performance level. In more advanced forms, a power shifting technique can be used to maximize performance by giving more power to busier cores by taking power from less busy cores.

9.3.3 Power Cap

For this policy, the EnergyScale firmware allows the POWER8 processor to perform as well as possible underneath a hard power ceiling (to within milliwatts of accuracy). Frequency, voltage, and throttling are used to limit the throughput of the system to prevent the Power Cap from being exceeded. This policy allows data centers to limit the total power consumed by the server to budget their electric bill or work within physical constraints (power delivery, cooling capacity) of their data center

9.3.4 Turbo Performance Boost

This policy allows EnergyScale to boost frequency by managing voltage and internal modes assuming lower than peak workload. Chip health (temperature, circuit performance, and so on) is monitored by internal sensors to ensure safety, such that the chip never exceeds its power or thermal specification. By using activity event counters as an estimation for power being consumed at various periods during runtime, a given code sequence and mode selection is deterministic on every part in every environment. As a result, the execution time of a given piece of code is largely invariant between multiple runs, regardless of environmental

1. Any of these three policies can be run concurrently. Under these policies, a given code sequence can run at different frequencies (including overclocked "Turbo" when possible) and operating modes depending on the part, environmental conditions, workload, and so on. The system manages voltage, frequency, and internal modes as a function of customer policy and internal sensors (DTS and performance counters).

conditions, a part's ability to overclock, and so on. Under this policy, EnergyScale firmware can change frequency or modes dynamically during runtime, but this change occurs the same way every time code is run and on every part in that sort bin.

9.4 Feature Summary

The POWER8 processor has additional enhanced high-value power management features as follows:

- Enablement of micropartition energy management through SPR-based PStates controlled by the hypervisor.
- Autonomous, real-time management through firmware driven, on-chip controller (OCC).
- Per core chiplet voltage regulation to allow each core to be independently set based on the currently running workload. Bounds are managed by the OCC.
- Per-thread accounting (instruction completion, work rate, memory counting) to enable advanced utilization-based power management algorithms.
- Memory access threshold throttling to control both partition-level power management and power shifting between processors and memory, the two largest components of power in the CEC.

9.5 Overview of Chip Hardware Power-Management Features

9.5.1 Communication Paths for System Controllers

- Dedicated special wake-up bit per core chiplet owned by the OCC
- Hypervisor and OCC communication. Dedicated indirect OCB channel for hypervisor queue (up to 64 bytes) in OCC SRAM.

9.5.2 Sensors

- Digital thermal sensor (DTS)
  - Diode bandgap design.
  - Analog/digital converter built into hardware to provide a digital readout.
  - Available via SCOM during runtime; used by the OCC to safely implement turbo modes and to protect the chip from environmental changes that could lead to overheating.
  - Three implemented per core; one implemented per EX cache region.
  - Automated hardware thermal overtemperature protection is not supported. The real-time OCC firmware accomplishes this function.
- Dedicated performance, microarchitecture, and activity/event counters
  - Used for processor and memory utilization and weighted power activity proxy measurement to direct power and performance trade-off decisions and selection of appropriate power management techniques.
  - Not shared with the performance monitor unit (PMU).
  - Events and thresholds are also routed to the PMU and HTM for power and performance analysis using traditional performance techniques.

  - Per-thread run cycle, instruction dispatch, instruction completing, work rate, and programmable memory hierarchy counters for per-thread utilization accounting.
  - Chiplet memory access counter and throttling for per-core-chiplet memory power allocation and shifting.
  - Processor bus overcommit retry counter access.

9.5.3 Accelerators

- On-chip controller (OCC)
  - Embedded PowerPC 405 with 16 KB instruction cache and 16 KB data cache
  - On-chip SRAM tank (512 KB)
  - Access to system DRAM memory via the processor bus for instruction and data area overflow (firmware managed)
- SPIVID command interface to chip external VRMs controlling core voltage rails. SPIVID VRMs accept respective core V_{DD} and V_{CS} target values.

9.5.4 Actuators/Controls

- Architected idle modes (nap, sleep, winkle)
  - Hypervisor can execute idle instructions to quiesce the processor pipeline and allow varying levels of power savings, each with higher latency to enter or exit.
  - No support for doze.
  - Support for sleep instruction (fast sleep: core power on; deep sleep: core power off). The sleep instruction allows for the L3 cache to remain functional for use by other cores.
  - Support for winkle instruction (fast winkle: chiplet power to V_{RET}; deep winkle: chiplet power off).
- Per-chiplet core frequency control.
  - Digital PLL (DPLL) allows desired dynamic frequency range of -50% to +10%.
  - With winkle of other cores, can be up to +30% of nominal core frequency.
  - Automated frequency reduction nap, sleep, winkle and low activity detect – controlled by PMICR SPR settings.
- External (off-chip) VRM voltage control.
  - Three SPI VID interfaces to VRMs are associated with a given chip.
    - VRMs are passed respective core V_{DD} and V_{CS} targets with CRC to ensure transmission. VRMs, using load-line sensing, ramp the rails without offset overlap.
    - VRMs return a status frame for command confirmation of VID write validity.
  - Redundancy supported via a broadcast of VID commands on all configured interfaces.
    - If two interfaces, good if one responds positively (acknowledged and no errors).
    - If three interfaces, good if two respond positively (acknowledged and no errors).
  - If responses are negative (timeouts, acknowledged with errors), voltage changing is suspended. OCC notification for firmware handling.
- Memory controller (DIMM) throttling. See *POWER8 Memory Buffer User's Manual* for full support details.
- Firmware runtime power-management controls.

  - Power Management Control Register (PMCR) SPR for run-time Pstate control. (See *Section 9.8 Architected Control Facilities* on page 202 for details.)
    - Pstates: 8-bit signed values that are an offset from the part's nominal frequency.
    - Global: requests the global voltage rail to be changed.
    - Local: requests the local frequency and voltage to be changed.
  - Power Management Idle Control Register (PMICR) SPR for idle-time Pstate control.
    - Nap Pstate
    - Sleep Pstate
    - Winkle Pstate
  - Power Management Memory Access Register (PMMAR) SPR for chiplet memory traffic control.
    - Chiplet memory traffic accounting.
    - Chiplet memory traffic throttling (if in single-core chiplet partition mode).

9.5.4.1 Configurations with Unused Components

- Dynamically or statically disable active power in the I/O devices for unused buses.
  - Static disablement of unused buses available in each EI4 bus using SCOM.
  - EI4 buses support hardware-driven dynamic spare lane power down.
- Dynamically clock-gating unused units.
  - The following units can be clock gated (via thold) when the associated chip is not attached to the interface.
    - PCI-E Host Bridges (PHB)
    - SMP interconnect A0:A2 Elastic Differential I/O (EDI) buses individually
- Statically clock-off unused units/buses
  - Each MCS0 or MCS1 unit and associated differential memory interface (DMI) buses.
  - Memory control buffer (MCB) for direct drive memory
  - Elastic I/O - four buses individually.
- Partial good requirements
  - Both partial good "bad" scenarios and gard/runtime deallocation
  - Bad core chiplets (with a bad L3 cache region) can be completely power gated at IPL time (chiplet power same as winkle).

9.6 Chip Hardware Power-Management Features

9.6.1 Chiplet Voltage Control

The POWER8 processor supports several VRM control mechanisms to support multiple different system configurations. The core chiplets are on a separate voltage plane than the other Nest components of the chip. The chip-level power management control (PMC) macro and the OCC are, in combination, programmable to support these configurations.

Core chiplets all share the same voltage plane and have to run with "highest common denominator" (that is, the core demanding the highest voltage sets the value of the voltage rail). The OCC is responsible for establishing the best frequency and therefore voltage bounds based on the workload running, the power/performance efficiency policy selected by the customer, and the system budgets established by the thermal management component.

9.6.2 Chip-Level Voltage Control Sequencing

The external VRMs (eVRMs) sourcing the logic (V_{DD}) and array (V_{CS}) rails are controlled by voltage control interfaces from the POWER8 chip. These interfaces use serial peripheral interconnect (SPI) signaling to a VRM chipset that converts the addressed VID command to industry standard Intel VRM-11 interface components or others as implemented by the VRMs.

9.6.2.1 SPIVID VRM Control Sequencing

The V_{DD} and V_{CS} target voltages are sent in one command to the VRM set. That target can be the full voltage swing request (V_{MAX} to V_{MIN} or vice versa) or any subset. With the V_{DD} and V_{CS} targets, the VRMs, through sampling of load lines, manage the offset of the two rails during the slew.

9.7 Functional Description of Processor Core Chiplet

9.7.1 Power Gating

The POWER8 processor implements per-core and per-core-chiplet power gating with two separately controlled power grid regions inside the EX core chiplet. These power regions also correspond to clocking domains. Thus all circuits on the full-frequency core clock grid (that is, processor core and L2 cache) are controlled by the core power gate. The EX cache power gate controls the voltage to the circuits on the half-speed cache_nclk region of the chiplet (for example, L3 cache, NCU, and processor bus interface units). This allows the entire core chiplet to be powered off to save maximum power when not in use (via the winkle state or static EX cache mode). It also allows only the core region to be powered off via the sleep state, such that the L3 cache remains available for other cores on the chip to use as an L3.1 extended cache for lateral castouts to maximize their performance.

The benefits of POWER8 power gating include:

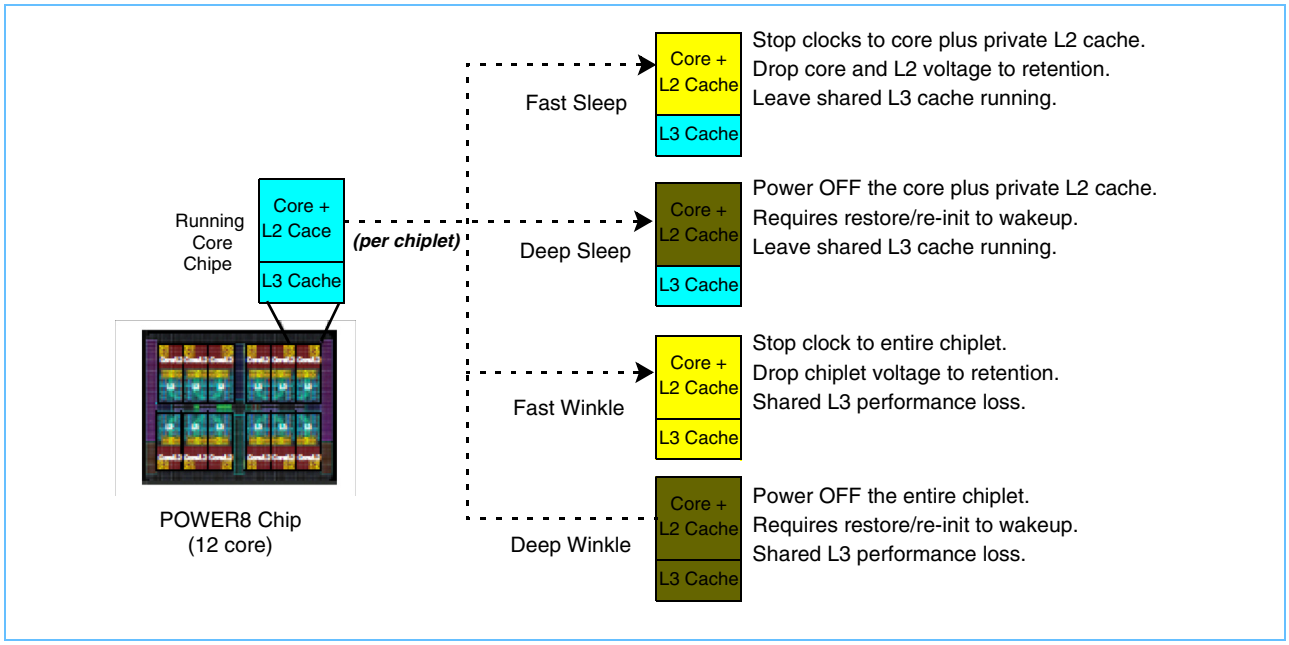
- Dynamic EX cache mode can be a runtime option via sleep mode.
- The adjacent core can enter turbo mode to take advantage of overall lower chip power and temperature.
- Idle power is significantly reduced.

- Per-chiplet control enables better partition-level power management capability (due to per-chiplet control).
- Better power efficiency modes are enabled.
- Better energy managment capability and benchmark scores.

9.7.2 Idle States

When a section of a unit or macro detects that it is idle, it gates clocks to the unneeded latches to save active power. This is done extensively in both the core and the nest which has the effect of reducing peak power because all circuits cannot be kept busy 100% of the time. However, when the hypervisor decides that the entire processor core can be idle, it can execute one of the architected idle state instructions. These instructions stop the fetch and dispatch of instructions, quiescing the core. Depending on the idle instruction executed, hardware can further reduce voltage and clock grid frequency or even power the core completely off to zero volts to save leakage power. *Figure 9-1* shows the effect of each of these idle modes on the processor core chiplet power.

Figure 9-1. Idle Mode Summary



A single thread entering any architected idle state causes that thread to stop dispatching and fetching instructions.

When a thread wakes up from an idle state, it takes an SRESET interrupt to restart program execution. The SRR0 contains the address of the idle instruction that caused the thread to go into idle state. (Note: The architecture says that SRR0 content is undefined.) SRR1 contains the reason for the wake-up and amount of state that was lost due to being in that idle mode.

All threads on a core must be in that architected idle state or deeper[1] for that core to enter an architected idle mode. The core enters the least aggressive idle mode of the eight threads. For example, if a core has one thread in nap, two in sleep, and one in winkle, the Core enters nap mode.

Note: Modes exist to disable chiplet clock/voltage off for lab modes such as hardware trace (performance and debug analysis).

The POWER8 processor supports three of the four architected power-save or idle states defined in the Power ISA (nap, sleep, and winkle). There is an important distinction between thread idle state and core idle state. When all threads on a core are in an architected idle state, more aggressive power-management techniques can be engaged at the core, core chiplet, or even chip level.

Table 9-1. Supported Chiplet Power Management Modes

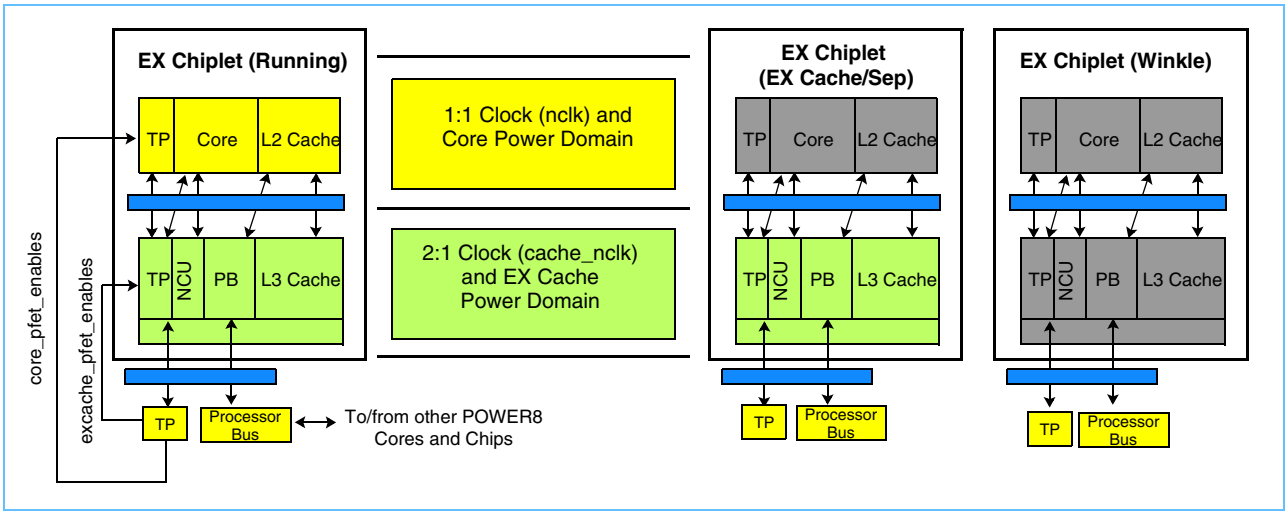
Mode/State	Core Domain Voltage	EX Cache Domain Voltage	Core Domain Mesh	EX Cache Domain Mesh	Core Domain THolds	EX Cache Domain THolds[1]	Estimated Relative Power [2]	Estimated Entry and Exit Latency
Running	V_{FUNC}[3]	V_{FUNC}[3]	Running	Running	Running	Running	N/A	N/A
Nap	V_{FUNC}[3]		Running[4]		Partially stopped	Running	90%	Typical: ~ 2 µs Worst case: < 50 µs
Fast Sleep	V_{ON}	V_{FUNC}[3]	RefClk	Running[4]	Stopped	Running	60%	Expected typical < 1 ms
Deep Sleep	V_{OFF}[5]	V_{FUNC}[3]	Off	Running[4]	Stopped	Running	25%	Expected typical < 5 ms
Fast Winkle	V_{ON}		RefClk		Stopped	Stopped	45%	N/A
Deep Winkle	V_{OFF}[5]	V_{OFF}[5]	Off	Off	Stopped	Stopped	Effectively zero (~0.1 W)	Expected typical < 20 ms

1. Full L3.1 performance.
2. Compared to operating system idle loop.
3. If enabled, these states allow for internal voltage reulation.
4. These states have an optional frequency reduction available.
5. This state requires POR assistance to restore lost state.
6. Power save mode transition rules: There are no transitions between power management idle states. Entry and exit from all architected idle states happens to and from a running state.

1. Deeper is defined as an idle state that is architecturally defined to save more power; that is, winkle is deeper than sleep which is deeper than nap.

With the power-gating capabilities implemented, the sleep and winkle idle states provide a progression as shown in *Figure 9-2*.

Figure 9-2. Sleep and Winkle Power Gating Progression



9.7.2.1 Core and Thread Doze

Core and thread doze are not supported. The hypervisor does not use doze mode.

9.7.2.2 Single Thread Nap, Sleep, and Winkle

Thread state can be lost when a single thread enters nap, sleep, or winkle mode, because it can cause the core to perform an SMT thread switch for performance gain on the remaining non-idle threads.

The only state lost in thread nap mode is the thread-specific state in the case that SMT mode changed. All non-thread-specific states including timebase registers and hypervisor states are preserved. Some thread-specific hypervisor states are preserved in nap.

When an SMT thread switch is enabled, the napping thread's resources can be given to other threads to improve core performance. Software must restore the architected state of the dormant thread upon exiting nap mode except for the following architected facilities, which are preserved by the hardware:

- SLB State
- All Hypervisor Special Registers (includes PURR, SPURR, AMOR, UAMOR, AMR, ACOP)
- DEC
- SPRG0-3
- DAR
- DSISR
- DABR, DABRX
- DSCR
- All Performance Monitoring Special Registers (PMCs, SIAR, SDAR)

9.7.2.3 Sleep

Generally, the sleep instruction removes the core from operation while leaving the associated L3 cache for use by other core chiplets. The amount of power savings and the inverse amount of exit latency is controlled by whether the core power gating is used or not.

Upon entering sleep, hardware invalidates the coherency caches (L1, ERAT, TLB, and so on) in the core and purges cache/state in the L2 units. The L3 cache is fenced from the core/L2 and retained as operational in lateral castout (LCO) mode, so that it can be used by other operational core chiplets.

Upon fencing the L3 portion (referred to as the extended cache option), the core and L2 cache are either:

- Powered off using power gating core FETs within the chiplet (defined as deep sleep).
- Have all clocks in the core stopped and the core clock grid dropped to the refclk frequency. The voltage is lowered to a $V_{RETENTION}$ level for low-latency exit (defined as fast sleep).

Logically, all state of the processor core domain is lost, including hypervisor state.

Before entering sleep, the hypervisor must update the power-on memory image with the state of any registers that must be set after wake-up. This list includes: HRMOR, HSPRG0, LPCR, HID0, HID1, HID5, along with meaningful Timer Facility PURR/SPURR/TB values. In addition, the hypervisor can set values into the PMICR for the following fields:

sleep_pstate_req	Indicates the Pstate that the EX cache portion of the chiplet is requested to run.
sleep_pstate_en	Enables as valid the sleep_pstate_req field.
sleep_global_en	Indicates if the sleep_pstate_req is to be treated as global.

Because the threads are waking up in real mode, the hypervisor guarantees that the first thread to wake-up restores SDR1 and LPIDR before the others leave real mode.

Any runtime changes to the state of the processor core and L2 cache by the off-chip firmware components (OCC and FSP) must be done with special wake-up asserted and a simultaneous update to the power-on memory image containing the necessary SCOM operations to restore it.

SCOM (and PCB) accesses to a core in sleep are no longer possible as the power to the core may been shut off (deep sleep case); for simplicity, fast sleep is treated the same. To access the core for these operations, a special wake-up action must take place. See *Section 9.7.3 Special Wake-up* on page 198.

Note: The hypervisor is required to set LPCR[PECE] = '101' on nonguarded cores (otherwise the core hangs when an interrupt occurs).

Exiting sleep only occurs because either a malfunction alert (caused by some other core checkstop) or external interrupt targeting a thread in the core. In the case of a deep sleep exit, a mini-power-on reset is performed on the core by a chip level power-on reset (POR) to bring the core back online (scan, ABIST, SCOM, and so on.). In the case of a fast sleep exit, voltage and frequency are restored and fencing is dropped to allow processing of the waking condition.

In either case, after wake-up, the hypervisor must resynchronize the Timebase facility.

9.7.2.4 Nap

In this state, clocks to a majority of the processor core is turned off while preserving coherency (L1 and L2 caches, ERATs, TLB, and SLB). The clock rate to the chiplet is dropped to lowest frequency that the current processor bus setting allows. This allows for low-latency entry and exit (micro-seconds). Generally, the entire IFU and LSU remain running while allowing for gating of non-snoop portions. A portion of pervasive remains running (for example, sensors, Timebase, and interrupt processing).

Nap Frequency Change

Based on the setting of the PMICR, the execution of the nap instruction will drop the chiplet to reduced frequency (nap Pstate). Upon wake-up. latency of a few microseconds will be incurred for the DPLL to ramp back to full frequency.

9.7.2.5 Winkle

Generally, the winkle instruction removes the core and associated L3 EX cache from operation.The amount of power savings and the inverse amount of exit latency is controlled by whether the core and EX cache power gating is used or not.

The winkle idle instruction causes power to be gated off to the entire chiplet (core/L2/L3) and cause the chiplet to be disconnected from the processor bus. Like Sleep, all state in the processor core domain is considered lost, including Hypervisor state. However, unlike Sleep, any Hypervisor and Firmware state in the remainder of the chiplet is also lost.

Upon disconnection from the processor bus, the core and chiplet are either:

- Powered off using power gating FETs within the core and chiplet (defined as deep winkle)
- Have all clocks in core and EX cache stopped and the core and chiplet clock grid dropped to the refclk frequency (defined as fast winkle)

Before entering winkle, the hypervisor must update the power-on memory image with the state of any registers that must be set after wake-up. This list includes: HRMOR, HSPRG0, LPCR, HID0, HID1, HID5, along with "meaningful" Timer Facility PURR/SPURR/TB values. In addition, the hypervisor can set values into the PMICR for the following fields:

winkle_pstate_req	Indicates the Pstate that might be requested globally (to lower voltage); locally, this has no effect.
winkle_pstate_en	Enables as valid the winkle_pstate_req field.
winkle_global_en	Indicates if the winkle_pstate_req is to treated as global.

Because the threads are waking up in real mode, the hypervisor guarantees that the first thread to wake up restores the SDR1 and LPIDR Registers before the others leave real mode.

Any runtime changes to the state of the processor core, L2 cache, and L3 cache by the off-chip firmware components (FSP) must be done with a special wake-up asserted and a simultaneous update to the power-on memory image containing the necessary SCOM operations to restore it.

SCOM (and PCB) accesses to a core in winkle are no longer possible, because the power to the core chiplet has been shut off. To access the core for these operations, a special wake-up action must take place. See *Section 9.7.3 Special Wake-up* on page 198.

Exiting winkle only occurs due to either a malfunction alert (caused by some other core checkstop) or an external interrupt targeting a thread in the core. In the case of a deep winkle exit, a mini-power-on reset is performed on the core by a chip-level power-on reset to bring the core and EX cache back online (scan, ABIST, SCOM, and so on). In the case of a fast winkle exit, voltage and frequency are restored and fencing is dropped to allow processing of the waking condition.

9.7.3 Special Wake-up

Normal wake-up from architected idle states resumes instruction execution in the processor core, beginning at the SRESET vector, due to an interrupt condition being present. There are times when various components of firmware need to access facilities inside the chiplet without resuming instructions. A special wake-up mode is provided to each firmware component (FSP, OCC, and hypervisor). It suppresses new architected idle states from being entered by the chiplet. If already in an idle state, it causes, if necessary, restoration of power and state first. This allows the clocks to resume to the chiplet for the access to be completed.

When special wake-up is cleared, one of two things happen:

- If a wake-up condition exists or if the processor core is no longer in an architected idle state because a wake-up interrupt already occurred, nothing happens.
- If the core is still in an idle state and no wake-up condition exists, the chiplet re-enters the appropriate idle state that was previously requested by the core.

While special wake-up is asserted, the core chiplet runs at the frequency represented by the idle Pstate (sleep Pstate or winkle Pstate) as defined in the PMICR unless bounded by the settings of PMax and PMin as established by the OCC. Low-activity detect is disabled from changing state when architected idle state and special wake-up are both active.

9.7.4 Pstates

The POWER8 processor implements the Pstate architecture with a local Pstate table of 128 logical entries.

9.7.4.1 Architectural Overview

Hypervisor firmware and operating system software control the scheduling of work on the processor core threads. It is necessary to abstract out the low-level details of frequency and voltage settings into a facility called a Pstate. This abstraction provides an isolation layer from the hardware, because the combination of voltage and frequency can vary per chip based on manufacturing variation and technology while allowing control of performance and power efficiency, Performance and power efficiency have a linear relationship to frequency and a quadratic relationship to voltage. As long as this linear and quadratic relationship to Pstates is understood, the energy management components in the hypervisor can appropriately manage both performance and power efficiency.

The Pstate itself does not logically alter the behavior of the core in terms of fetch, dispatch, and execution rates of instructions. Rather, it is an indication of the performance level represented in terms of a frequency target that the platform (processor chip and underlying control firmware) attempts to fulfill within the constraints of available power and thermal capacities while also taking the opportunity to minimize the same.

The Pstate architecture is based upon a hierarchical view. Some elements of a request can be handled in a manner local to the core chiplet making the request. A primary example of a local operation is frequency change. Other elements must be sent out of the core chiplet to the central, or global, entity which might have

to arbitrate between multiple core chiplet requests to take action in response. A primary example of a global operation is where voltage rails are shared among core chiplets and coordination is required to slew such a rail.

Underlying the Pstate request notion, as made by energy management components with the hypervisor, is a platform set of elements that manage the allowable power (and, hence, performance) that is expended by elements in the machine. In the case of Pstates, the element is the core chiplet. Thus, limits are placed on the maximum allowable frequency and voltage based on budgetary thresholds as well as, on minimum allowable frequencies that are necessary to maintain machine coherency. This is why the Pstates are considered requests in that the platform hardware and firmware can limit or bound the value requested. Therefore, mechanisms are provided to allow the hypervisor components to read the actual operational values so as to react, if necessary, to any such clipping actions.

9.7.4.2 Definitions

Fnom

Fnom is the nominal frequency established after part characterization that allows the part to fully run all supported code streams within the power and thermal envelope provide by the system. The value of Fnom is, thus, both part and system dependent.

The Fnom value is established by platform-specific firmware that has access to both information about the part instance (typically held in a vital product data (VPD) record) as well as the present system characteristics. This value is written into core chiplet and global chip hardware facilities as a representation of a chip-specific frequency value. It is then used as the basis from which all Pstate calculations (global and local) are made.

Pstate Number (PSN)

A Pstate Number is a signed value that is an offset from the chip Fnom frequency value. PSNs range from +127 to -127.

Pmax

Pmax is the maximum PSN allowed for the core chiplet and is established by platform-specific firmware. It takes into account a number of factors including, but not limited to:

- The power delivery into the socket containing this core chiplet
- The number of core chiplets implemented in this socket
- The thermal environment of the socket
- The policy established by the customer (maximum performance, maximum power savings, deterministic performance, and so on)

Hardware ensures that no mechanism, software in the form of a Pstate, or hardware in the form of power shifting, or guardband management mechanisms exceeds this value.

Pmin

Pmin is the minimum PSN allowed for the core chiplet and is established by platform-specific firmware. It takes into account a number of factors including, but not limited to:

- The cache coherency of the SMP machine, because timely response to cache snooping actions is typically required
- Any applicable performance floors that customer-driven policy can establish

Global Pstate (GPS)

The Global Pstate represents a signed value that is an offset from the chip Fnom value. It indicates that a frequency change is being requested that requires coordination by a chip-level entity (for example, outside the core chiplet) to manage any elements across other dependent core chiplets. Typically, the element that needs to be managed is the voltage setting. Some examples of a voltage change action are as follows:

- The voltage rail supplying this core chiplet is shared by other core chiplets. In this case, a "vote" about what the voltage should be must take place. If warranted, a voltage change action is completed.
- The voltage rail supplying this core chiplet is segregated from other core chiplets but the control means for changing the voltage setting is external to the core chiplet itself. In this case, the central entity completes a voltage change action on the core chiplet's behalf.

In the case of Global Pstate increase, any voltage change that needs to occur must be completed and the core chiplet notified of this fact before the frequency change can take place.

In the case of Global Pstate decrease, the core chiplet requests the decrease to the central entity, but might or might not lower its present local operating Pstate. This choice on the local Pstate is a matter of policy. If the request was honored (that is, it passed the rail volting), the central entity notifies all chiplets of the reduction as a new Global Actual Pstate. It waits for acknowledgements that the new voltage is honored and then reduces the voltage.

Local Pstate (LPS)

The Local Pstate represents a signed value that is an offset from the chip Fnom value. It indicates that a frequency change is being requested that does not require coordination by a chip-level entity; it is handled locally within the core chiplet. For implementations that do not have any means of varying voltage locally, this becomes a request to immediately move frequency within the bounds established by the current Global Pstate or FMax (whichever is lower) on the upper side and FMin on the lower side. Upon change of the frequency, the new current value is reflected in the Actual Local Pstate field.

Because any operational changes are local, no communication to the central element is performed. Implementations can (and probably will) make the Actual Local Pstate accessible via the service network so that the values can be read as necessary.

Nap Pstate (NPS)

The Nap Pstate is the PSN that indicates the frequency (and, optionally, voltage) change to request upon the execution of the Nap instruction.

Sleep Pstate (SPS)

The Sleep Pstate is the PSN that indicates the frequency (and, optionally, voltage) change to request upon the execution of the Sleep instruction.

Winkle Pstate (WPS)

The Winkle Pstate is the PSN that indicates the frequency (and, optionally, voltage) change to request upon the execution of the Winkle instruction.

9.7.4.3 Permissible Behavior

- Global Pstate requests are made to the central area upon any PMCR store. There is no interlock defined that ensures that a previous request was, in fact, honored.
- The core chiplet can run at a higher local Pstate than presented in a Global Pstate Request because the global actual Pstate is, at all times, honored.

While it is generally viewed that Energy Management algorithms (in the hypervisor) make the local Pstate less than or equal to the global Pstate request (to save energy), the hardware does not enforce this limitation.

9.7.4.4 Interaction with Idle Modes

The sleep and winkle high, medium, and low latency controls are intended to allow platforms the option of leveraging power gating (or similar technologies), because these have a major influence on restoration latency versus power expended.

The winkle Pstate can be used in conjunction with the latency controls where power gating cannot be invoked; therefore, a Pstate is used to communicate to the platform the intended operating point.

Architecture Note

An intended usage model is:

- Hypervisor energy management algorithms establish the nap Pstate (NPS), sleep Pstate (SPS), and winkle Pstate (WPS) fields and their respective global and local settings.
- Eventually, all threads will achieve nap (inclusive of some in sleep and some in winkle), sleep (inclusive of some in winkle) or winkle state
  - For the case of nap, the NPS is used to affect frequency and voltage per the flow respective of the global/local bit.
  - For the case of sleep, the SPS is used to affect frequency and voltage per the flow respective of the Sleep Global Enable bit.
  - For the case of winkle, the WPS is used to affect frequency and voltage per the flow respective of the Winkle Global Enable bit.

9.7.5 Resonant Clocking

Resonant Clocking is a mode of the clock distribution whereby on-chip inductors are used in conjunction with the natural clock mesh capacitance to form a resonant structure to save power drawn by the clock mesh. This mode can only be enabled when the frequency (Pstate) is within one of two specific bands. These bands are defined by SCOM registers associated within each chiplet and are setup at POR. After the transition facility is enabled, any frequency changes that exit these defined bands cause the resonant clocking mode to be disabled before the transition. Then, after the new frequency is achieved, if that new frequency is within a defined resonant band, the resonant clocking circuits are re-enabled.

Resonant clocking is modally supported.

9.8 Architected Control Facilities

9.8.1 Power Management Control Register (PMCR)

The PMCR is the mechanism used by the hypervisor firmware to request Pstate changes. This register has only one instance for the core.

Table 9-2. Power Management Control Register (PMCR) - SPR 884 (Sheet 1 of 2)

Field	Field Name	Access	Function
0:7	global_pstate_req	R/W	Global Pstate request Writes to this field initiate a coordination action with any available central element that arbitrates between other cores that might share a power rail with this core. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent[1].
8:15	local_pstate_req	R/W	Local Pstate request Writes to this field initiate a local performance change request without explicitly orchestrating with any available central element. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent[1].
16	Reserved		Reserved

1. For the POWER8 processor, the increment is the reference clock frequency divided by the DPPL divider.

Table 9-2. Power Management Control Register (PMCR) - SPR 884 (Sheet 2 of 2)

Field	Field Name	Access	Function
17:23	spurr_fraction	R/W	SPURR fraction Modifies utilization/accounting of the SPURR for the requested Local Pstate. This accounting only occurs if the requested Pstate is honored. The default value is '1000000'. It is multiplied by other SPURR factors to create a composite discount value. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent[1].
24:51	Reserved		Reserved
52:63	lpar_id	R/W	LPAR ID The value representing the logical partition ID that will be executing with the settings defined by the other fields. Writes to this field provide the LPAR ID for the platform to associate the power for the subsequent execution window (for example, until the next PMCR store). The actual accounting action is controlled by the power_acctng_change field. Reads from the field return the value last written.

1. For the POWER8 processor, the increment is the reference clock frequency divided by the DPPL divider.

9.8.2 Power Management Idle Control Register (PMICR)

The PMICR controls the Pstates used under idle modes.

This register has only one instance for the core.

Table 9-3. Power Management Idle Control Register (PMICR)- SPR 852 (Sheet 1 of 2)

Field	Field Name	Access	Function
0:7	nap_pstate_req	R/W	Nap Pstate (NPS) request Writes to this field initiate a coordination action with any available central element that will arbitrate between other cores that can share a power rail with this core. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent[1]
8	nap_pstate_en	R/W	Nap Pstate enable 1 Enable the Nap Pstate (NPS) request, Nap Global Enable, and Nap Latency functions. 0 Disables the Nap Pstate (NPS) Request, Nap Global Enable, and Nap Latency functions. Reads from the field return the value last written.
9	nap_global_en	R/W	Nap global enable 1 Upon the execution of a Nap instruction, the Nap Pstate (NPS) request is be sent to the central element as a Global Pstate Request. 0 Upon the execution of a Nap instruction, the Nap Pstate (NPS) request is be sent to the local element as a Local Pstate Request. Reads from the field return the value last written.
10:15	Reserved		Reserved
16:23	sleep_pstate_req	R/W	Sleep Pstate (NPS) request Writes to this field initiate a coordination action with any available central element that arbitrates between other cores that can share a power rail with this core. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent.[1]
24	sleep_pstate_en	R/W	Sleep Pstate enable 1 Enables the sleep Pstate (NPS) request, sleep global enable, and sleep latency functions. 0 Disables the sleep Pstate (NPS) request, sleep global enable, and sleep latency functions. Reads from the field return the value last written.
25	sleep_global_en	R/W	Sleep global enable 1 Upon the execution of a sleep instruction , the sleep Pstate (NPS) request is sent to the central element as a global Pstate request. 0 Upon the execution of a sleep instruction , the sleep Pstate (NPS) request is sent to the local element as a Local Pstate request. Reads from the field return the value last written.
26:31	Reserved		Reserved
32:39	winkle_pstate_req	R/W	Winkle Pstate (NPS) request Writes to this field initiate a coordination action with any available central element that arbitrates between other cores that can share a power rail with this core. Reads from the field return the value last written. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent.[1]

1. For POWER8, the increment is the reference clock frequency divided by the DPPL divider.

Table 9-3. Power Management Idle Control Register (PMICR)- SPR 852 (Sheet 2 of 2)

Field	Field Name	Access	Function
40	winkle_pstate_en	R/W	Winkle Pstate enable 1 Enable the winkle Pstate (NPS) request, winkle global enable, and winkle latency functions. 0 Disables the winkle Pstate (NPS) request, winkle global enable, and winkle latency functions. Reads from the field return the value last written.
41	winkle_global_en	R/W	Winkle global enable 1 Upon the execution of a winkle instruction, the winkle Pstate (NPS) request is sent to the central element as a global Pstate request. 0 Upon the execution of a winkle instruction, the winkle Pstate (NPS) request is sent to the local element as a local Pstate request. Reads from the field return the value last written.
42:43	winkle_latency	R/W	Winkle latency 00 Ceases instructions and honors Pstate change but does not perform additional state changing actions. 01 Indicates to the platform that a sub-state that has the lowest latency is enabled. 10 Indicates to the platform that a sub-state that might have a medium exit latency is enabled. 11 Indicates to the platform that a sub-state that might have higher exit latency is enabled. It is platform-specific as to the differentiation between low, medium, and high.
42:63	Reserved		Reserved

1. For POWER8, the increment is the reference clock frequency divided by the DPPL divider.

9.8.3 Power Management Status Register (PMSR)

The PMSR provides access to the platform values in place. This register has only one instance for the core.

Table 9-4. Power Management Status Register (PMSR) - SPR 853

Field	Field Name	Access	Function
0:7	global_pstate_actual	RO	Actual Global Pstate Reads from this field return the presently established global Pstate value. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent.
8:15	local_pstate_actual	RO	Actual Local Pstate Reads from this field return the presently established local Pstate value. This value is always less than or equal to global_pstate_actual. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent.
16:23	pmin	RO	Pstate Minimum Reads from this field return the presently established minimum Pstate set by the platform. This value can change autonomously based on the current policy in place and the physical constraints of the platform. The value is an 8-bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent.
24:31	pmax	RO	Pstate Maximum Reads from this field return the presently established maximum Pstate set by the platform. This value can change autonomously based on the current policy in place and the physical constraints of the platform. Value is an 8 bit signed integer representing an offset from $F_{nominal.}$ Legal values are +127 to -127 with the value increment being platform dependent
32:63	Reserved		Reserved.

9.8.4 Power Management Memory Activity Register (PMMAR)

The PMMAR controls the memory activity that is allowed to be produced by the partition. This register is replicated by partition if in multiple-partition mode.

Table 9-5. Power Management Memory Activity Register (PMMAR) - SPR 854

Field	Field Name	Access	Function
0:31	mem_op_limit	R/W	Memory operation limit Defines the number of memory operations operations allowed within the rolling credit window defined by mem_crdt_window**.** Once this limit is reached, the memory operations are stalled until the mem_crdt_window expires**.**
32:45	mem_crdt_window	R/W	Memory credit limit Defines the rolling credit window time for which memory operations are managed. x'0000' Disabled (no stalling) x'0001' 64 µs ... x'3FFF' 1.048 s
47:63	Reserved		Reserved

10. Performance Profile

This chapter provides information about the performance profile of the POWER8 processor. Due to the complex nature of the speculative, out-of-order execution core coupled with the multilevel storage hierarchy, it is important that people concerned with performance and performance optimization develop an insightful understanding about the operational nature of the machine.

10.1 Core

10.1.1 Level-1 Instruction Cache

The L1 I-cache is a 32 KB, 8-way set-associative cache of instructions allocated in 128-byte lines with 32-byte sector valids. The replacement algorithm is pseudo LRU. The I-cache is also kept coherent with the L2 cache (an **icbi** instruction can be treated as a NOP). The I-cache is used to feed instructions to the rest of the core at up to eight instructions per cycle as long as the addresses are presented each cycle and each address results in a cache hit.

The I-cache consists of three parts:

- Instruction cache
- I-cache directory (IDir)
- Effective address directory (EADIR)

The EADIR is used to predict the way select for a given I-cache access keeps the I-cache access time to one cycle. The EADIR is accessed using EA bits 52:56 and tagged using EA bits 41:51. All entries are initially assumed to be shared between threads, but certain combinations of reduced sharing are also possible (sharing is limited to thread pairs in Cloud mode). The basic I-cache miss latency assumes an EADIR miss. An EADIR hit that is later determined to be an I-cache miss adds approximately eight cycles to the base miss latency. The I-cache directory contains the real address (RA) of the lines in the I-cache, line and sector valids, and certain MSR state bits (PR, LE, and HV) from when the line was fetched. Both the EADIR and the IDir are addressed using EA bits 52:56.

Single-thread aliasing can occur when a given EA[41:56] cache line is required for more than one real address, or combination of MSR bits. Only one of the two combinations can be valid in the I-cache for a given thread at any one time, and an EADIR invalidate is required before fetching the other alias. Multi-thread EADIR aliasing results when two threads map the same EA(41:56) to two different real addresses or MSR combinations. The second address is brought in as private, after an EADIR invalidate. In 4 LPAR mode, only the two threads in the same LPAR can share I-cache lines.

The I-cache is banked and allows a concurrent read and write, if they go to different banks. On an I-cache miss and L2 hit, instructions are typically returned on the L2 cache interface in two consecutive 64-byte beats. If the instructions are coming from the L3 cache, four 32-byte sectors are typically expected to arrive in every other cycle. If the instructions are coming from the processor bus, the critical sector comes first and it is typical to expect the successive sectors to come every fifth processor cycle. Note that the I-cache miss latency is typically four cycles longer than the D-cache latency.

The I-cache is arranged in 128-byte lines of four 32-byte sectors. Each sector has its own valid. Within a line, the I-cache can be accessed on any 16-byte boundary to return 32 consecutive bytes. Addressing the last 16-byte boundary of a line returns only 16 bytes.

Data in the I-cache must also be in the L2 cache (inclusivity). This means that if the L2 cache removes a line that is thought to be in the I-cache, it must send an invalidate for that line to the I-cache. Invalidates are handled in parallel with read accesses so that normally they are invisible to performance. The LRU algorithm in the L2 cache requires data to be accessed to keep it from being replaced. This means that a line in the I-cache that is being heavily used, can be forced out of the L2 cache due to an L1 D-cache request. To avoid this situation, the processor periodically sends a dummy request to the L2 cache on an I-cache hit address. When bandwidth permits, these requests serve only to update the L2 LRU for the I-cache lines in hopes of preventing them from being aged out.

10.1.2 Level-1 Instruction ERAT

On instruction fetches, effective address bits are used to index into the I-cache, the directory and the effective-to-real-address-translation (ERAT) table. The ERAT is a fully-associative 64-entry table and contains both the effective addresses and the real addresses. For an ERAT hit, the effective address of the instruction must match the effective address contained in the ERAT entry being indexed, and the ERAT entry must be valid. In addition, the IR, US, HV, and PR bits from the MSR at the time of ERAT miss are stored in the ERAT when the ERAT is loaded on a ERAT miss. These bits must match the corresponding bits in the MSR at the time of instruction fetch for an ERAT hit. I-ERAT minimum miss penalty (assuming a TLB hit) is 18 cycles.

The I-ERAT directly supports 4 KB, 64 KB, and 16 MB page sizes. Other page sizes are stored in the next smaller supported page-size granule.

10.1.3 Instruction Prefetch

If a particular instruction fetch misses in the I-cache, a demand fetch reload request (possibly speculative) is sent to the L2 cache subsystem. The L2 cache processes this reload request with high priority and forwards it onto subsequent levels of cache or memory in the event that it misses in these caches.

In addition to these demand-oriented instruction fetching mechanisms, the POWER8 processor also works to automatically prefetch instruction cache lines that might be referenced soon into its instruction cache. If there is an I-cache miss, it generates prefetch requests for the next one in both SMT2 and SMT4 mode or three sequential cache lines (in ST mode). In SMT8 mode, no instruction prefetching requests are generated. As these requests return cache lines, they are stored directly in the instruction cache. The banked cache design of the POWER8 instruction cache allows a concurrent read and write (as long as they go to different banks), so that writing prefetch lines into the instruction cache does not steal cycles from fetching the instructions from the cache.

The POWER8 processor has an additional mode for systems where memory bandwidth is a concern. This mode is called reduced-speculation mode. In this mode, demand requests are marked as speculative if, at the time of the I-cache miss, the fetch is not for the next-to-complete instruction. If the request is marked speculative, the memory subsystem can return an indication of "no data" if the requested data is not found in the L3 cache and local memory has indicated that recent bandwidth demands were greater than a programmable threshold. In this case, the demand request must be retried when the request becomes next-to-complete. Also, in this mode, prefetches are always marked speculative and are not retried regardless of the returned state.

10.1.4 Branch Prediction

In each cycle, up to eight instructions are fetched from the instruction cache. From there, these instructions are sent to the branch prediction logic. The branch prediction logic scans all the fetched instructions looking for branches; this information is used by the instruction decode logic to look for up to two branches for group formation. Depending upon the branch type found, various branch prediction mechanisms engage to help predict the target address of the branch or the branch direction or both. More specifically, branch target addresses for **bclr** and **bcctr** instructions can be predicted using the link stack or the count cache mechanism (target addresses for absolute and relative branches are computed directly as part of the branch scan function). Dynamic branch direction prediction (taken or not taken) is done through the use of three branch history tables. Static branch direction prediction is done using hints as defined by *Power ISA User Instruction Set Architecture (Book I)*.

It is important to note that all conditional branches are predicted in the POWER8 processor (even if the condition is resolved well ahead of time or the value of the link or count register is known when the branch to link or count instruction is fetched) and no unconditional branches are "predicted". As branch instructions flow through the rest of the pipeline and ultimately execute in the branch execution unit, the actual outcome of the branches is determined. At that point, if the predictions were correct, the branch instructions are simply completed like all other instructions. If the prediction is incorrect, the instruction fetch logic issues a flush and redirects the pipeline down the corrected path.

10.1.4.1 Branch Direction Prediction using the Branch History Tables

The POWER8 processor uses a set of three branch history tables to predict the direction of branch instructions. The first table, called the local predictor, is similar to the traditional branch history table. It is a 16K entry array that is indexed by the address of the branch instruction to produce a 2-bit predictor. The MSb of the 2-bit predictor indicates whether the branch direction should be "taken" or "not-taken".

The second table, called the global predictor, works to predict a branch based on the actual path of execution to reach the branch. The path of execution is identified by a 20-bit vector, one bit per fetch group (that is, the group of instructions fetched in a cycle), for each of the previous 20 fetch groups. This vector is referred to as the global history vector. Each bit in the global history vector indicates whether the next group of instructions fetched are from a sequential cache sector (0) or not (1). The global history vector captures this information for the actual path of execution through these sectors. That is, if there is a redirection of instruction fetching, some of the fetched group of instructions are discarded and the global history vector is corrected immediately. The 20-bit global history vector is first folded (by a bitwise XOR of bits 0:10 with bits 9:19 to generate an 11-bit path vector, which is then hashed by a bitwise XOR with the address of the branch instruction to index into the 16K entry global history table to produce another 2-bit branch direction predictor. Similar to the local predictor, the MSb of this 2-bit global predictor is an alternate indicator of whether the branch should be predicted to be "taken" or "not-taken".

Finally, the third table, called the selector table, keeps track of which of the two prediction schemes works better for a given branch. It is used to select between the local and the global predictions. The 16K entry selector table is indexed exactly the same way as the global history table is indexed and the MSB of the selected entry is used as the 1-bit selector. This combination of branch prediction tables has been shown to produce very accurate predictions on a wide range of workload types.

If the first branch encountered in a particular cycle is predicted as not taken, the POWER8 processor can predict and act on a second branch in the same cycle. In this case, the machine registers both branches as predicted (for subsequent resolution at branch execution), and it redirects the instruction fetching based on the second branch.

As branch instructions are executed and resolved, the branch history tables (as well as, the other predictors described in the following sections) are updated to reflect the latest and most accurate information. Unconditional branches (including branches with the BO field set to '1z1zz') and statically predicted conditional branches (such as, branches with the "a" bit set to '1') do not have an entry in the BHTs. Therefore, they do not cause any BHT update.

All three BHTs are implemented as banked arrays and allow concurrent read and write operations. If the concurrent accesses are to different banks, both are honored. However, if there is a bank conflict, the read is given higher priority. The BHT update logic will perform multiple write updates speculatively up to 5 attempts before forcing a hole in the fetch logic to allow the write to be done.

10.1.4.2 Branch Prediction using Static Prediction and "a", "t" Bits

For some conditional branches, the software knows what the branch direction prediction ought to be. The POWER8 processor allows the software to override the dynamic branch prediction in such cases. Software communicates its wish to override dynamic branch prediction by setting the "a" bit in the BO field. If the "a" bit is '0', then the "t" bit is ignored and the dynamic branch prediction described above is used. If the "a" bit is '1' for a branch, dynamic branch prediction is not used and no entry is updated in the branch history table when such a branch is executed. The static branch direction prediction itself is communicated by properly setting the "t" bit in the BO-field ('1' for taken and '0' for not-taken). Software is expected to use this feature only for those conditional branches for which it believes that software branch prediction will result in at least as good a performance as the hardware branch prediction.

Three separate cases have been identified where the software should override hardware branch prediction for improved performance:

- *When the conditional branch is known by the software to be almost always uni-directional.* For example, branches that guard segments of code that are only executed when a rare event occurs.
- *For the branches that close out a lock acquisition sequence*. It is desirable to force the branch prediction to be *not taken*. This provides the best performance for the most common case where the lock is successfully acquired. Even if the lock is not successfully acquired on this iteration, it is still best to assume (from a branch prediction standpoint) that it will be acquired in the next iteration. Note that, left alone, if the lock is not acquired in the first iteration, the branch history mechanism would work to update the prediction to predict *taken* (that is, predict lock acquisition failure and cause more "lwarx" traffic) for the next iteration.

```
top: lwarx
      add
      stwcx
      bc top     <-- Power8 will predict this branch to be not taken, through
                     software directives that properly set the "a" and "t" bits.
```

- *To force a conditional branch to be always mispredicted to initiate instruction prefetching.* This allows some instructions to be speculatively executed or processed to some extent by the instruction fetch logic before they are discarded. The instruction in the (wrongly) predicted path can be used as hint instruction to the memory subsystem. For example, *software prefetching of instructions* from location "Line_to_touch" can be initiated by forcing a branch misprediction as follows ("a"-bit in the **bc** instruction indicates "must agree with static prediction").

```
Short distance touches:
             bc Line_to_touch       // Static prediction taken, but CR bit is set "not-taken"
Long distance touches:
             bc Next                // Static prediction not-taken, but CR bit is set "taken"
```

```
        b Line_to_touch        // Initiate prefetch for cache line "Line_to_touch"
Next:...                       // Instructions in the actual instruction stream
```

This type of software prefetching is useful if the line to prefetch is in the L3 cache or beyond. Due to high penalty for branch misprediction, it might not be beneficial if the referenced line is already in the L2 cache and even harmful if it is already in the I-cache. It is beneficial if the compiler makes special attempts to schedule code around such a branch that reduces the misprediction penalty. Attempts to reduce the forced branch misprediction penalty can be made by:

- Setting the CR bit used by the "bc" as early as possible
- Scheduling such a branch in a code segment where there are relatively few branches so that the branch does not wait too long in the branch issue queue behind other branches
- Trying to overlap a likely D-cache miss with the forced branch misprediction
- Scheduling such a branch after an existing long chain of flow dependency

10.1.4.3 Address Prediction Using the Link Stack

The POWER8 processor uses a link stack to predict the target address for a branch-to-link instruction that it believes corresponds to a subroutine return. By setting the hint bits in a branch-to-link instruction, software communicates to the processor whether the instruction represents a subroutine return or a target address that is likely to repeat or neither (see *Table 10-1*).

When instruction fetch logic fetches a branch and link instruction either unconditional or conditional but predicted taken, it pushes the address of the next instruction into the stack. When it fetches a branch-to-link instruction with "taken" prediction and with hint bits indicating a subroutine return, the stack is popped and instruction fetching starts from the popped address.

In the POWER8 processor, the link stack is 32-entries deep per thread in single-thread and SMT2 mode. In SMT4 mode, it is 16-entries deep. In SMT8 mode, it is 8-entries deep. In all modes, entries are preserved to keep speculative pushes, which can be used for branch misprediction recovery.

Speculative execution can corrupt the link stack, both its pointer and its contents. The exact nature of the corruption depends on the sequence of the stack-modifying branch instructions that get purged from the system on a misspeculation.

Because branch-to-link instructions are fairly common and the branch misprediction penalty is high, the POWER8 processor uses an extensive speculation tolerance mechanism in its link-stack implementation that allows it to recover the link stack under most circumstances. To recover the stack pointer at misspeculation, the value of the stack pointer at the time a branch is scanned by the instruction fetch logic is stored in a table and restored from on branch misprediction.

Table 10-1. Handling of bclr and bclrl Instructions

Instruction	BH Field	POWER8 Design	Power ISA
bclrl	xx	If the branch is predicted taken, the link stack address is used as the predicted target address; however, the link stack is not popped.	Reserved.
bclr	00	If the branch is predicted taken, the link stack is popped and the popped address is used as the predicted target address.	The branch is a subroutine return.
bclr	01	If the branch is predicted taken, the target is predicted using the count cache. The count cache data and confidence fields might be updated when the branch is executed and resolved. No action is taken by the link stack.	Target address is likely to repeat.
bclr	10	Same as BH = '00'	Reserved.
bclr	11	If the branch is predicted taken, the link stack address is used as the predicted target address; however, the link stack is not popped.	Target is not predictable.

10.1.4.4 Address Prediction using the Count Cache

The target address of a branch to count (**bcctr**[**l**]) instruction is often repetitive and can be predicted if the address is saved in a cache from an earlier execution of the same instruction. This is also true for some of the branch-to-link (**bclr**[**l**]) instructions, which are not predictable through the use of the link stack because they do not correspond to a subroutine return. By setting the hint bits appropriately, software communicates to the hardware whether the target address for such branches are predictable using a cache. See *Table 10-2*.

The POWER8 processor uses two 256-entry count caches (one with hashed addressing similar to the global branch history table and one using the unhashed instruction address) for predicting the target of a **bclr**[**l**] or **bcctr**[**l**] instruction whose target address is likely to repeat. Each entry in the count cache can hold a 62-bit address. When a **bclr**[**l**] or **bcctr**[**l**] instruction is executed, for which the software indicates that the target is predictable using such a cache, the target address is written in the count cache. When such an instruction is fetched, the target address is predicted using the count cache. The count cache does not have a separate array to select between the two count caches. Instead, the 2-bit selector value is stored in the array using the unhashed instruction address.

Sometimes a given branch has a small set of targets, but predominantly favors a particular target. To help predict such branches, the count cache also uses a 2-bit confirmation counter (for each entry) for replacement. Every time an entry is used for successful prediction, the counter is incremented (saturating counter), and it is decremented on a misprediction. If the counter is zero, the entry can be replaced. When an entry is first allocated or installed, the counter value is set to 1.

Note: **bcctr** instructions that are found in the local count cache must be located in octword blocks for the POWER8 processor. The local count cache is primarily used for **bctr**(**l**) instructions that have only one target. The POWER8 processor can only make one prediction per fetch group and bases the prediction off the fetch group address (octword) and not the branch that is eventually identified as the first taken branch in the group. If there are other non-bcctr branches in the octword, there is no problem. But if there are two or more bcctr branches in the fetch group, only one can be in the count cache at a time so prediction can thrash.

Table 10-2. Handling of bcctr and bcctrl Instructions

Instruction	BH field	POWER8 Design	Power ISA
bcctr, bcctrl	00	If the branch is predicted taken, the target address is predicted using the count cache. Update the count cache when the branch is executed, if the branch is resolved as taken. For the **bcctrl** instruction, if the branch is predicted taken, push in the link stack the address of the next sequential instruction when the **bcctrl** instruction is fetched.	Target address is likely to repeat.
bcctr, bcctrl	01	Same as BH = '00'.	Reserved
bcctr, bcctrl	10	Same as BH = '00'.	Reserved
bcctr, bcctrl	11	Same as BH = '00'.	Target is not predictable

10.1.4.5 Round-Trip Branch Processing

Instruction fetch logic can fetch up to eight instructions in a given cycle and forward them down the pipeline to the instruction decode logic. It scans through this group of instructions in the next cycle to determine the position of all the branches in the group and the predictions for them. At the same time, it fetches the next sequential cache sector of eight instructions, if there is no I-cache miss, I-ERAT miss, fetch redirection, pipeline hold (for example, due to successive stages not having enough resources to process the instructions already in progress). The next sequential I-cache sector is fetched with the assumption that there is no unconditional branch or a conditional branch that is predicted taken in the previous group of instructions that it fetched. If the assumption is correct, instruction fetch proceeds unhindered. If incorrect, a *bht redirection* event occurs, when instruction fetch logic instructs the instruction decode logic to discard the instructions it sent after that branch and starts fetching from the target of that branch. If it is a branch-to-link/count instruction, the target of the branch is predicted based on the prediction mechanism described earlier. Otherwise, the target of the branch is calculated from the instruction and its address. On a BHT redirection, two cycles worth of instruction fetching might be lost.

If the branch is mispredicted, all the instructions in the pipeline that are fetched after the branch instruction are purged from the system and new instructions from the new address, as determined by the executed branch instruction, are fetched. The purging involves restoring all the queues and the mapper register files to the state that is consistent with not having fetched or executed any instruction after the branch.

Because of its deep pipeline, the branch misprediction penalty is high in the POWER8 processor. If a mispredicted branch is executed before some of its earlier instructions (in program order), the pipeline will not be completely dry after the misprediction. These earlier instructions can keep the execution units busy while the instruction fetch logic brings new instructions, thus reducing the penalty to some extent. The average branch misprediction penalty depends on many factors including when the associated CR bit becomes available, the number of instructions that survive a misprediction, and associated cache misses. However, after a branch is executed and found to be mispredicted, it takes a minimum of 16 cycles for a new instruction from the redirected path to come down the pipeline.

10.1.4.6 BC+4 Handling

Unconditional branches with the link bit set and a displacement of four can be used to set the address of the next instruction into the link register. Architecturally these branches are taken and go to the next instruction. The hardware handles BCL+4, with a BO = 20 (unconditional taken) as a special case. For this special case, the branch is always treated as *not taken* for fetch, resulting in a no-fetch penalty. When the special branch executes, it updates the link register, and does not cause a flush (even though fetch processed it as *not taken* and architecturally it was *taken*). The PMU sees these special branches as requiring direction prediction, direction predicted correctly, and branch not taken counts.

10.1.4.7 BC+8 Handling

In addition to the normal branch handling, bc instructions whose BO field indicates only a CR dependency with a displacement of +8 are handled as a special case. Such bc+8 instructions are marked as potential candidates for conversion to predication. When the next instruction after a bc+8 is in the list of allowed to be predicated (see *Table 10-3*), the bc+8 is marked as first for group formation processing. Finally, if the branch prediction for the bc+8 instruction is predicted not taken, the instruction pair is treated similar to an **isel**, in that the branch cannot be mispredicted and the next instruction is conditionally executed based on the bc outcome. Any time a bc+8 instruction is executed, the branch prediction algorithm writes a value assuming the branch was not taken. This encourages all bc+8 instructions to be converted to predication. Use of a NOP can be used to prevent bc+8 conversions in specific cases (making the bc+8 a bc+12).

Table 10-3. bc+8 Pairable Instructions

Assembler Mnemonic
addi
addis
add
and
or
xor
ori
stb
sth
stw
std

When a bc+8 instruction pair is converted to predication, the destination register now is pending until the predication is resolved. This can mean that subsequent dependent instructions are prevented from issuing. The bc+8 conversions must be avoided for the following circumstances:

1. If the branch is expected to be very predictable, the branch prediction hardware outperforms predication.
2. If the pairable operation is the start of a dependency chain involving loads, sometimes it is more beneficial to have the loads start execution even if the branch eventually flushes. Predication delays dependent operations from executing. Therefore, if a long latency load is delayed, the performance is better served by allowing the branch to be predicted. This is especially true when the resultant memory access can be the same cache line in either outcome. Branch misprediction can be hidden by the long latency memory

reference while the dependency delay due to predication cannot be hidden. Heap sort algorithms show this effect.

10.1.5 Store-Hit-Load Avoidance Table

In an out-of-order execution machine, a younger load can be executed ahead of an older store. When the addresses are independent, this is not a problem. But if the addresses overlap, the load has the wrong data. The base design detects this condition and flushes the instruction stream after the store causing all subsequent instructions to be refetched. This ensures that the load follows the store and gets the newer data.

There are cases where the store hit load is repeated many times (possibly in a loop) where performance is degraded due to the flushes. Therefore, the POWER8 processor adds an 8-entry table that records the store address when an SHL flush occurs. On the second occurrence, the decode logic is instructed to mark the RA field of the store as written (store with dependency). Assuming the out-of-order load uses the same RA register (RA != 0), the load is now delayed due to this artificial dependency. This performs significantly better than flushing. The table also monitors the store address for subsequent SHL flushes and disables the function when it does not appear to help.

The store-hit-load table is managed with an LRU algorithm. Entries are also invalidated when the corresponding I-cache congruence class is written. Only certain stores can be converted to store with dependency. *Table 10-4* lists ineligible stores.

Table 10-4. Stores Ineligible for SHL Avoidance

Ineligible Stores for SHL Table
All stores when RA = 0
All stores with update
All store conditionals

10.1.6 Instruction Buffer

Instructions read out of the I-cache are forwarded to the IBuffer as a staging area for group formation. The IBuffer is arranged as a register file where each row can hold up to four instructions (16-byte aligned from the I-cache). There are a total of 32 rows, but the number of rows per thread varies by SMT mode as shown in *Table 10-5*. Branches not from the last instruction of an aligned quadword and not to the first instruction of an aligned quadword cause inefficiencies in the IBuffer.

Table 10-5. IBuffer Rows per Thread

SMT Mode	Rows
1	16
2	16
4	8
8	4

IFetching for a thread is prevented if there is no room in the IBuffer and is restarted as the IBuffer is drained. There must be room for eight instructions before an IFetch is initiated. There are non-flush circumstances where a full IBuffer could go empty before newly fetched instructions make it back to the IBuffer if groups are large and dispatch is unencumbered. But this is expected to be rare.

10.1.7 Group Formation

Instructions are pulled from the IBuffer and formed into dispatch groups. Instructions in a dispatch group enter the out-of-order part of the processor in parallel and must complete together so that group formation can limit the overall performance of a processor.

In single thread mode, the group size is limited to six nonbranch and two branch instructions. All instructions must come from the oldest two quadwords in the IBuffer, and two quadwords can be emptied per cycle (with some exceptions). Slots are numbered 0 - 5 for the non-branch instructions and 6 - 7 for the branch instructions.

In multi-thread mode, group size is limited to three nonbranch instructions and one branch instruction for each of two threads (groups 0 and 1). All instructions must come from the oldest two quadwords per thread, but only one quadword per thread can be emptied per cycle. Nonbranch slots are numbered 0 - 2 for the group 0 thread and 3 - 5 for the group 1 thread. The branch slot is 6 for the group 0 thread and 7 for the group 1 thread.

Group formation rules for the POWER8 processor are in the following sub-sections.

10.1.7.1 General Rules

General group formation rules (regardless of SMT mode) are:

- Can go past the first branch (predicted T or NT) to pick up more instructions (branch anywhere). A special case ends a group when a backward branch to within 1024 bytes is predicted taken. This allows a compiler to assume group formation starts at a loop-entry point after the first iteration.
- Instructions marked First must start a group.
- Instructions marked Last must end a group.
- No FPU operations are allowed in a group after a branch (T or NT).
- A bc+8/12 operation is marked First if the branch confidence table indicates to predicate.
- A 2-way cracked operation occupies two nonbranch slots.
- A 3-way cracked operation occupies three nonbranch slots (modified from 4-way cracked). This ends the group.
- Branch and link instructions that update the Link Register are marked Last.
- A 3-source operation in slots 0 or 1 reserves slot 2 for transmitting the third operand. A 3-source operations in slots 3 or 4 reserves slot 5 for transmitting the third operand. No 3-source operations are allowed in slots 2 or 5.

10.1.7.2 Rules Specific to ST Mode

Rules specific to ST mode only (6 + 2 dispatch) are:

- A group is sourced from the bottom two rows of the IBuffer in each cycle.
- Maximum of six nonbranches per group.
- Maximum of two branches per group.
- A group ends immediately after the second branch (T or NT).
- Eight instructions total per group.
- Slot 2 is reserved when a bc+8/12 starts a group. This slot is used for extra source operands in the predicated group.
- Two-way cracked operations go to slots 0 and 1. A second 2-way cracked operation can be in the same group, if they are next to each other in the code stream (goes to slots 3 and 4).
- Three-way cracked operations go to slots 0, 1, and 2, and end the group.

10.1.7.3 Rules Specific to SMT Modes

Rules specific to SMT modes (dual 3 + 1 dispatch) are:

- The two 3+1 dispatch halves operate independently, reading two threads out of the IBuffer in each cycle.
- A group is sourced from the bottom two rows of the IBuffer (two rows per thread) in each cycle.
- Maximum of three nonbranches per group.
- Maximum of one branch per group.
- Four instructions total per group.
- Two-way cracked operations go to slots 0 and 1 for group 0, or slots 3 and 4 for group 1.
- Three-way cracked operations go to slots 0, 1, and 2 for group 0, or slots 3, 4, and 5 for group 1.

Both dispatch halves share a single microcode engine. This can cause one half to stall the other if both halves need microcode simultaneously.

10.1.8 First and Last Instructions

Instructions that require specific execution units are marked as *First*. See *Table 10-6* on page 220 for a list of instructions marked as First.

Instructions that require that no other instructions follow in the same dispatch group are marked as *Last* and end the present dispatch group prematurely. See *Table 10-7* on page 221 for a list of instructions marked as Last.

Table 10-6. List of Instructions Marked as First

Mnemonic	Mnemonic	Mnemonic	Mnemonic
addc	**mcrf**	**selii.04.**	**slbie**
addc.	**mcrxrt**	**selii.05.**	**slbmfee**
addco	**mfbhrbe**	**selii.06.**	**slbmfev**
addco.	**mfcr**	**selii.07.**	**slbmte**
addeo.	**mfmsr**	**selii.08.**	**sleep**
addg6s	**mfocrf**	**selii.09.**	**sp_attn**
addmeo.	**mfspr** [non-renamed]	**selii.10.**	**stbcix**
addzeo.	**mfspr_CTR**	**selii.11.**	**stbcx.**
and	**mfspr_default**	**selii.12.**	**stdcix**
clrbhrb	**mfspr_LR**	**selii.13.**	**stdcx.**
crand	**mfspr_xer**	**selii.14.**	**sthcix**
crandc	**mfsr**	**selii.15.**	**sthcx.**
creqv	**mfsrin**	**selir.00.**	**stmd**
crnand	**msgclr**	**selir.01.**	**stmw**
crnor	**msgclrp**	**selir.02.**	**stsdi**
cror	**msgsnd**	**selir.03.**	**stsdx**
crorc	**msgsndp**	**selir.04.**	**stswi**
crxor	**msle**	**selir.05.**	**stswx**
dcbf	**mtcrf**	**selir.06.**	**stwcix**
dcbfl.l=1	**mtiamr**	**selir.07.**	**stwcx.**
dcbfl.l=3	**mtmsr**	**selir.08.**	**subfc**
dcbst	**mtmsr_ee**	**selir.09.**	**subfc.**
dcbz.0	**mtmsrd**	**selir.10.**	**subfco**
doze	**mtmsrd_ee**	**selir.11.**	**subfco.**
dsixes	**mtspr** [non-renamed]	**selir.12.**	**subfeo.**
eieio	**mtspr_CTR**	**selir.13.**	**subfmeo.**
hrfid	**mtspr_default**	**selir.14.**	**subfzeo.**
icswx	**mtspr_LR**	**selir.15.**	**sync**
icswx.	**mtspr_xer**	**selri.08**	**syncpte**
isel	**mtsr**	**selri.12**	**tabort.**
isync	**mtsrin**	**selri.13**	**tbegin.**
lmd	**nap**	**selri.14**	**tcheck.**
lmw	**or**	**selri.15**	**tend.**
logmpp	**pbt.**	**selrr.08**	**tlbie**
lq	**rfebb**	**selrr.12**	**tlbiel**
lqarx0_1	**rfid**	**selrr.13**	**tlbiellpg**
lqarx1_1	**rfscv**	**selrr.14**	**tlbsync**
lsdi	**rvwinkle**	**selrr.15**	**trechkpt**
lsdx	**sc**	**slbfee.**	**treclaim.**
lswi	**scv**	**slbia**	**tsr.**
lswx	**selii.00.**	**slbia.001**	**txer***
ltptr	**selii.01.**	**slbia.010**	**waitasec**
ltptr_op	**selii.02.**	**slbia.011**	
lwsync	**selii.03.**	**slbia.1--**	

Table 10-7. List of Instructions Marked as Last

Mnemonic	Mnemonic	Mnemonic	Mnemonic
addc	**mfspr_MMCR2a**	**mfspr_TB40**	**mtspr_HID0**
addc.	**mfspr_MMCR2b**	**mfspr_TBL**	**mtspr_HID1**
addco	**mfspr_MMCRA0**	**mfspr_TBU.269**	**mtspr_HMER**
addco.	**mfspr_MMCRA1**	**mfspr_TBU.285**	**mtspr_HSPRG0**
addeo.	**mfspr_MMCRC**	**mfspr_TEXASR**	**mtspr_HSPRG1**
addg6s	**mfspr_MMCRH**	**mfspr_TEXASRU**	**mtspr_HSRR0**
addmeo.	**mfspr_MMCRS**	**mfspr_TFHAR**	**mtspr_HSRR1**
addzeo.	**mfspr_PCR**	**mfspr_TFIAR**	**mtspr_IC**
and	**mfspr_PIR**	**mfspr_TFMR**	**mtspr_LDBAR**
clrbhrb	**mfspr_PMC1.0**	**mfspr_TMFR**	**mtspr_MMCR2a**
doze	**mfspr_PMC1.1**	**mfspr_TRACE**	**mtspr_MMCR2b**
hrfid	**mfspr_PMC2.0**	**mfspr_trig0**	**mtspr_MMCRC**
mfcr	**mfspr_PMC2.1**	**mfspr_trig1**	**mtspr_MMCRS**
mfspr_BESCR	**mfspr_PMC3.0**	**mfspr_trig2**	**mtspr_PIR**
mfspr_BESCRR	**mfspr_PMC3.1**	**mfspr_VTB**	**mtspr_PMCR**
mfspr_BESCRRU	**mfspr_PMC4.0**	**mfspr_xer**	**mtspr_PMICR**
mfspr_BESCRS	**mfspr_PMC4.1**	**msle**	**mtspr_PMMAR**
mfspr_BESCRSU	**mfspr_PMC5.0**	**mtmsr**	**mtspr_PMSR**
mfspr_CIR	**mfspr_PMC5.1**	**mtmsr_ee**	**mtspr_PPR**
mfspr_CTRL	**mfspr_PMC6.0**	**mtmsrd**	**mtspr_PPR32**
mfspr_DEC	**mfspr_PMC6.1**	**mtmsrd_ee**	**mtspr_PSPB**
mfspr_EBBHR	**mfspr_PMCR**	**mtspr_BESCR**	**mtspr_PURR**
mfspr_EBBRR	**mfspr_PMICR**	**mtspr_BESCRR**	**mtspr_RPR**
mfspr_HDEC	**mfspr_PMMAR**	**mtspr_BESCRRU**	**mtspr_RWMR**
mfspr_HID0	**mfspr_PMSR**	**mtspr_BESCRS**	**mtspr_SIER0**
mfspr_HMEER	**mfspr_PPR32**	**mtspr_BESCRSU**	**mtspr_SIER1**
mfspr_HMER	**mfspr_PSPB**	**mtspr_CFAR**	**mtspr_SPMC1**
mfspr_HPMC1	**mfspr_PURR**	**mtspr_CIABR**	**mtspr_SPMC2**
mfspr_HPMC2	**mfspr_RWMR**	**mtspr_CIR**	**mtspr_SPRC**
mfspr_HPMC3	**mfspr_SIER0**	**mtspr_CSIGR**	**mtspr_SPRD**
mfspr_HPMC4	**mfspr_SIER1**	**mtspr_CTRL**	**mtspr_SPRG0**
mfspr_IC	**mfspr_SPMC1**	**mtspr_DEC**	**mtspr_SPRG1**
mfspr_IMC	**mfspr_SPMC2**	**mtspr_EBBHR**	**mtspr_SPRG2**
mfspr_LDBAR	**mfspr_SPRC**	**mtspr_EBBRR**	**mtspr_SPRG3**
mfspr_MMCR0.0	**mfspr_SPRD**	**mtspr_FSCR**	**mtspr_SPURR**
mfspr_MMCR0.1	**mfspr_SPURR**	**mtspr_HDEC**	**mtspr_SRR0**
mfspr_MMCR1.0	**mfspr_TAR**	**mtspr_HEIR**	**mtspr_SRR1**
mfspr_MMCR1.1	**mfspr_TB.268**	**mtspr_HFSCR**	**mtspr_TACR**

Mnemonic
mtspr_TAR
mtspr_TB40
mtspr_TBL
mtspr_TBU
mtspr_TCSCR
mtspr_TEXASR
mtspr_TEXASRU
mtspr_TFHAR
mtspr_TFIAR
mtspr_TFMR
mtspr_TRACE
mtspr_trig0
mtspr_trig1
mtspr_TSCR
mtspr_TSRR
mtspr_TTR
mtspr_VTB
mtspr_xer
nap
or
rfebb
rfid
rfscv
rvwinkle
sc
scv
selii.00.
selii.01.
selii.02.
selii.03.
selii.04.
selii.05.
selii.06.
selii.07.
selii.08.
selii.09.
selii.10.
selii.11.
selii.12.
selii.13.

Mnemonic
selii.14.
selii.15.
selir.00.
selir.01.
selir.02.
selir.03.
selir.04.
selir.05.
selir.06.
selir.07.
selir.08.
selir.09.
selir.10.
selir.11.
selir.12.
selir.13.
selir.14.
selir.15.
sleep
sp_attn
subfc
subfc.
subfco
subfco.
subfeo.
subfmeo.
subfzeo.
tabort.
tcheck.
tend.
trechkpt
treclaim.
tsr.
waitasec
tbegin.
tend.
tcheck.
msgclr
msgclrp
msgsnd

Mnemonic
msgsndp
lqarx0_1
lqarx1_1
mtiamr
mfspr_DSCR3
mtspr_DSCR3
stbcix
sthcix
stwcix
stdcix
lq
stbcx.
sthcx.
stwcx.
stdcx.
isync
slbie
slbia
slbia.001
slbia.010
slbia.011
slbia.1--
slbmfev
slbmfee
tlbie
tlbiel
tlbiellpg
mfsr
mfsrin
slbfee.
slbmte

10.1.9 2-Way and 3-Way Cracked Instructions

For some instructions that are important for performance and that cannot be done in a single IOP but do fit in two or three IOPs, instruction cracking is used rather than microcode. Instruction cracking involves expanding a single architected instruction into two or three IOPs in the same instruction dispatch group. The IOPs are uninterruptable in the middle of the sequence. *Table 10-8* lists the 2-way cracked instructions and *Table 10-9* on page 225 lists the 3-way cracked instructions.

Table 10-8. 2-Way Cracked Instructions

Mnemonic	Mnemonic	Mnemonic	Mnemonic
addic.	**divdo.**	**selri.07**	**selrr.08**
addo.	**divdeo.**	**selri.08**	**selrr.09**
subfo.	**divwu.**	**selri.09**	**selrr.10**
addc.	**divweu.**	**selri.10**	**selrr.11**
addco	**divwuo.**	**selri.11**	**selrr.12**
subfc.	**divweuo.**	**selri.12**	**selrr.13**
adde.	**divdu.**	**selri.13**	**selrr.14**
addeo	**divdeu.**	**selri.14**	**selrr.15**
subfe.	**divduo.**	**selri.15**	**extsb.**
subfeo	**divdeuo.**	**selir.00**	**extsh.**
addze.	**selii.00**	**selir.01**	**extsw.**
addzeo	**selii.01**	**selir.02**	**cntlzd.**
subfze.	**selii.02**	**selir.03**	**cntlzw.**
subfzeo	**selii.03**	**selir.04**	**rldicl.**
addme.	**selii.04**	**selir.05**	**rldicr.**
addmeo	**selii.05**	**selir.06**	**rldic.**
subfme.	**selii.06**	**selir.07**	**rlwinm.**
subfmeo	**selii.07**	**selir.08**	**rldcl.**
nego.	**selii.08**	**selir.09**	**rldcr.**
mullw.	**selii.09**	**selir.10**	**rlwnm.**
mullwo.	**selii.10**	**selir.11**	**rldimi.**
mulld.	**selii.11**	**selir.12**	**rlwimi.**
mulldo.	**selii.12**	**selir.13**	**sld.**
mulhw.	**selii.13**	**selir.14**	**slw.**
mulhd.	**selii.14**	**selir.15**	**srd.**
mulhwu.	**selri.15**	**selrr.00**	**srw.**
mulhdu.	**selri.00**	**selrr.01**	**xsradi.**
divw.	**selri.01**	**selrr.02**	**srawi.**
divwe.	**selri.02**	**selrr.03**	**srad.**
divwo.	**selri.03**	**selrr.04**	**sraw.**
divweo.	**selri.04**	**selrr.05**	**dtcs**
divd.	**selri.05**	**selrr.06**	**tbegin.**
divde.	**selri.06**	**selrr.07**	**tend.**

Mnemonic
tcheck.
logmpp
pbt
pbt.
msgclr
msgclrp
msgsnd
msgsndp
lqarx0_1
lqarx1_1
mtiamr
mfspr_DSCR3
mtspr_DSCR3
stbcix
sthcix
stwcix
stdcix
lq
stbx
sthx
stwx
stdx
stbux
sthux
stwux
stdux
sthbrx
stwbrx
stdbrx
stbcx.
sthcx.
stwcx.
stdcx.
isync
slbie
slbia
slbia.001
slbia.010
slbia.011
slbia.1--

Mnemonic
slbmfev
slbmfee
tlbie
tlbiel
tlbiellpg
mfsr
mfsrin
slbfee.
slbmte
stvebx
stvehx
stvewx
stvx(l)
stxvd2x
stxvw4x

Table 10-9. 3-Way Cracked Instructions

Mnemonic
addg6s
addc
addc.
addco
addco.
subfc
subfc.
subfco
subfco.
addeo.
subfeo.
addzeo.
subzeo.
addmeo.
submeo.
selii.00.
selii.01.
selii.02.
selii.03.
selii.04.
selii.05.
selii.06.
selii.07.
selii.08.
selii.09.
selii.10.
selii.11.
selii.12.
selii.13.
selii.14.
selii.15.
selir.00.
selir.01.
selir.02.
selir.03.

Mnemonic
selir.04.
selir.05.
selir.06.
selir.07.
selir.08.
selir.09.
selir.10.
selir.11.
selir.12.
selir.13.
selir.14.
selir.15.

10.1.10 Microcode

Instructions requiring microcode to execute have additional performance penalties to access the microcode. A microcoded instruction after a non-microcoded instructions incurs a 2-cycle penalty before the first group of the microcode routine is available. These cycles are only visible if the pipeline is not stalled. Consecutive microcoded instructions do not require the two additional cycles. There is no overhead at the end of a micro-code sequence. *Table 10-10* lists those instructions that access the microcode.

Table 10-10. Instructions that Access Microcode

Instruction	Notes
lmw, **lmd**	
stmw, **stmd**	
lswi, **lsdi**	
lswx, ls**dx**	
stswi, **stsdi**	
stswx, **stsdx**	
ld, **ldu**, **ldx**, **ldux**	Little-endian mode only
lfdu, **lfdux**, **lfdx**, **lfd**	Little-endian mode only
lha, **lhax**, **lhau**, **lhaux**, **lhz**, **lhzx**, **lhzu**, **lhzux**	Little-endian mode only
lwa, **lwax**, **lwau**, **lwaux**, **lwz**, **lwzx**, **lwzu**, **lwzux**	Little-endian mode only
mtsr, mtsrin	

10.1.11 Instruction Fusion

POWER8 instruction fusion involves combining information from two adjacent instructions into one instruction so that it executes faster than the non-fused case. "Adjacent instructions" refers to the instruction locations after group formation. In single-threaded mode, 6/2 groups are formed where there are up to six non-branch instructions and up to two branch instructions. In multi-threaded mode, two 3/1 groups are formed where there can be up to three non-branch instructions and one branch instruction. To be adjacent, the instructions to be fused must be in the same group of non-branches, without a branch in between.

There are two fuse types:

{addi} followed by one of these {lxvd2x, lxvw4x, lxvdsx, lvebx, lvehx, lvewx, lvx, lxsdx}
- Requirements:
  - addi(rt) = lxvd2x(rb)
  - lxvd2x(ra) cannot be 0
- Result:
  - addi - no change
  - lxvd2x gets the immed field from addi., rb is not used.
  - This effectively provides a d-form version for the vector loads.
  - The dependency between the two operations is removed.

{addis) followed by one of these {LD, LBZ, LHZ, LWZ}
- Requirements:
  - addis(rt) = ld(ra) = ld(rt) - (cannot be 0)
  - addis(SI) first 11 bits must be all 0's or all 1's

Result:

addis gets changed to a NOP. (It still takes up a dispatch slot, but is sent directly to completion.) The last 5 bits of addis(SI) are sent with the ld (information from the addis is passed to the ld). The addis is removed from execution.

10.1.12 Instruction Dispatch

Dispatch represents the last in-order stage of the pipeline until completion. At this point, resources are checked and the pipeline stalled if not available. *Table 10-11* lists the dispatch stall conditions due to resource limitations.

Table 10-11. Resource Requirements for Dispatch

Dispatch Hold Condition	Minimum Cycles Required	Release (Completion Based)	Inefficiencies
Branch Issue Queue full			
CR Issue Queue full			
Unified Issue Queue full	6	+6 cycles	Three cycles reserved per dispatch lane
Insufficient CR mapper entries	6	+0 cycles	Three cycles reserved per dispatch lane
Insufficient XER mapper entries	6	+6 cycles	
Insufficient CTRLR mapper entries	4		
Insufficient GPR mapper entries		+9 cycles	Three cycles reserved per dispatch lane
Insufficient FPSCR mapper entries	2		
Insufficient VRF mapper entries		+9 cycles	Three cycles reserved per dispatch lane
LRQ full			
SRQ full			
GCT full (setup stage)			

10.1.13 Instruction Issue

Up to eight internal operations (IOPs) can be dispatched and renamed per cycle. After renaming, these IOPs are placed into a set of issue queues that are distributed by instruction type.

There are three different issue queues:

- Unified issue queue (UniQ): This is a 64-entry queue that is partitioned into two halves. Queue half 0 issues instructions to FX0, VS0, LS0, and LU0. Queue half 1 issues instructions to FX1, VS1, LS1, and LU1. Each queue half can issue instructions to one FXU, one VSU, one LU, and one LSU. Instructions from either queue half can issue to the DFU and Crypto units.
- BRQ: 15-entry issue queue for branch operations
- CRQ: 8-entry issue queue for condition register logical operations and mfSPRs, for SPRs that are owned by the IFU, ISU, and PC units. mfSPR IOPs occupy one entry in both the UniQ and CRQ.

10.1.13.1 Steering Policy

In ST mode, instructions are assigned to queue halves based on dispatch slots.

The instruction in slots 0, 1, and 2 are steered to alternating queue halves. For example, if IOP 0 is steered to queue half 1, IOP 1 would be steered to queue half 0 and IOP 2 would be steered to queue half 1. The IOP in slot 3 must be steered to the opposite queue half as the IOP in slot 0. Likewise IOP 4 is steered opposite IOP 1, and IOP 5 is steered opposite IOP 2.

The first instruction in a group is assigned to the queue half that had the fewer number of instructions assigned to it in the previous dispatch cycle.

In SMT2, SMT4, and SMT8 modes, the two queue halves are partitioned by thread set, so instructions are steered by thread set. Instructions in slots 0, 1, and 2 (thread set 0) are assigned to queue half 0. Instructions in slots 3, 4, and 5 (thread set 1) are assigned to queue half 1.

10.1.13.2 BRQ and CRQ Operation

Instructions are allocated into the BRQ and CRQ at dispatch time. Each cycle, the oldest ready instruction out of each queue is selected for execution. The age between threads is determined by dispatch order. An instruction is deallocated from its issue queue two cycles after it issues.

10.1.13.3 UniQ Issue Policies

Each cycle, each queue half selects four instructions to be issued, one each to the FXU, VSU, LU, and LSU for that queue half. In general, selection is biased toward the oldest ready instruction. Instructions can be speculatively issued before their source operands are really ready. They can also be speculatively issued before knowing if there is going to be contention for a resource such as a GPR write port. Instructions are rejected if they cannot be successfully issued, and are re-issued again later.

10.1.13.4 FXU and VSU Selection

In general, the oldest ready instruction is selected for issue on a given unit. However, due to cycle time restrictions, it is impossible to determine the oldest among 32 instructions and select one in a cycle. Instead, the issue queue half is divided into two quadrants of 16 entries each. The oldest ready instruction in each quadrant is selected. Each cycle, one quadrant is determined to be high priority and the other is low priority. An LFSR is used to select which quadrant is high priority. If the high-priority quadrant has any ready instructions, the oldest one from that quadrant is selected. Otherwise, the oldest from the low-priority quadrant is selected.

10.1.13.5 LU Selection

In addition to loads, simple fixed-point instructions are also eligible to be issued to the LU. The selection logic picks the first instruction found in this list:

- Oldest LU-eligible load or store_data in the high-priority quadrant
- Oldest LU-eligible load or store_data in the low-priority quadrant
- Youngest simple fixed-point in the high-priority quadrant
- Youngest simple fixed-point in the low-priority quadrant

10.1.13.6 LSU Selection

Stores, simple fixed-point, and some loads are eligible to be issued to the LSU. The selection logic picks the first instruction found in this list:

- Oldest store (or LSU-only load) in the high-priority quadrant
- Oldest store (or LSU-only load) in the low-priority quadrant
- Oldest LSU-eligible load in the low-priority quadrant
- Youngest simple fixed-point in the low-priority quadrant
- Youngest simple fixed-point in the high-priority quadrant

10.1.13.7 Dispatch Bypass Instruction Selection

Instructions are selected for issue using the normal selection mechanism if they were dispatched at least four cycles ago. Instructions in the dispatch+3 stage are selected for issue using a bypass (such as, dispatch+3 bypass), which uses a different selection policy.

The following instructions are not eligible to use the dispatch+3 bypass:

- Instructions that are dependent upon older instructions in the same dispatch group or the group dispatched in the previous cycle
- Conditional IOPs (used by bc+8)
- Loads and stores waiting for real LRQ or SRQ entries
- Completion-serializing instructions
- VS-routed operations
- Instructions with XER sources

Instructions are speculatively scheduled (oldest first) for dispatch bypass as if there are no older ready instructions in the issue queue in stages disp+4 or later.

10.1.13.8 Back-to-Back Issue Policy

In general, single-cycle fixed-point instructions are issued in consecutive cycles if they are in the same half of the queue. An instruction is considered to be back-to-back ready (B2B_RDY) if, for all of its source operands, either (1) the available lines are already set, or (2) the source operands in the same half of the queue and are currently marked as FX0_RDY and are single-cycle instructions. The oldest instruction that is either B2B_RDY or FX0_RDY is selected for execution on FX0.

10.1.13.9 Limitations of Back-to-Back

There are three sources of inefficiency in the back-to-back mechanism:

1. If an instruction has two source operands in the same half of the queue, neither of which has been deallocated, the dependent cannot wake up back-to-back unless both of the producer instructions are marked as Ready (such as, they are candidates for selection).

In the example shown in *Table 10-10*, instruction C has two source operands, A and B. Instructions A and B are independent. All three instructions are in the same half of the queue. Instruction A is selected in cycle 0. In cycle 1, it issues and its available line is set and dependent instructions (might) wake up. In cycle 1, B

wakes up, and it is in the FX0_RDY latch in cycle 2. However, by this point in time, A is no longer marked in the FX0_RDY latch. Instruction A has not been deallocated from the queue yet; therefore, C is still tracking two source operands - A and B. Because A is not marked as FX0_RDY, instruction C cannot wake up via the back-to-back mechanism in cycle 2.

Table 10-12. Example where Back-to-Back is not Possible A->C. B->C (All in the same half of queue)

	0	1	2	3			
A	sel	iss	rf				
B		wak	sel	iss	rf		
C				wak	sel	iss	rf
fx0_rdy	A		B		C		
b2b_rdy							

2. Only the first level in a chain of dependent instructions can execute via the back-to-back mechanism. For example, suppose instruction C is dependent on B, which is dependent on A. B can be selected back-to-back after A, but then C cannot be selected back-to-back after B. This is shown in *Table 10-13*. Note that D can then execute back-to-back after instruction C.

   If B is not selected back-to-back after A, C can be selected back-to-back after B.

Table 10-13. A -> B -> C (All in the same half of queue)

	0	1	2	3	4		
A	sel	iss	rf				
B	b2b wake	sel	iss				
C			wake	sel	iss		
D				b2b wake	sel	iss	
fx0_rdy	A			C			
b2b_rdy		B			D		

3. If there are two or more FX instructions marked as ready in the ready latches (that is, candidates for selection) and zero load/stores that are candidates for selection, then in the following cycle there will be no instructions woken up back-to-back that are candidates for selection. This is to handle the problem of FX instructions that are issued to the LSU attempting to wake up their dependents back-to-back.

10.1.13.10 Dual-Issued Stores

The store_agen and store_data portions of D-form store instructions share the same entry in the issue queue. For fixed-point stores, the store_agen is issued to the LSU and the store_data is issued to the LU. For FP/VMX/VSX stores, the store_agen is issued to the LSU and the store_data is issued to the VSU. The dependence tracking logic only tracks the source operand of the store_agen. Both the store_data and store_agen will "wake up" and attempt to issue as soon as the store_agen's source operand is ready. The store_data will be delayed until that point even if it was already ready to issue. If the store_data's source operand was not yet available and it tries to issue anyway, it will get rejected.

10.1.13.11 Wake-up Misspeculations

There are several cases where instructions issued can be woken up too early. Misspeculations are detected when the dependent instruction is in the register file access stage. In the following cases, instructions get a source-unavailable reject:

- Dispatch bypass: instructions can be issued off of a dispatch bypass before their source operands are ready.
- Store_data IOPs for dual-issued stores wake up when the store_agen wakes up. They can try to issue at any time after that point, except one or two cycles after the store_agen is issued. They can continue to reject over and over if their source operands are not ready.
- SAR bypass: If instructions are issued with 'needs_rtag' bits on, or if there was a recent castout, they are rejected. See *SAR Bypass - Related Rejects* on page 235.
- Dependents of load misses or LSU-rejected loads wake up their dependents too early.
- The instruction's producers were rejected for any reason (non-ready sources or any unavailable resource) after they woke up their dependents.

10.1.13.12 Chains of Misspeculations

It should be pointed out that 1-, 2-, 3-, and 4-cycle instructions can wake up their dependents before it is known that they must be rejected. This is because their available lines get set before or while the misspeculation is detected. Hence, it is possible for an infinite chain of dependent instructions to wake up and be scheduled speculatively. Any of the resource conflicts or wake-up misspeculations discussed previously can cause these scenarios.

Chains of dependents of load instructions continue to wake-up and are issued even after the load is rejected or misses in the L1 cache and DVAL is known to be zero.

10.1.13.13 Other Issue Inefficiencies

Normally, a 2-cycle FXU operation has an issue-to-issue latency of two cycles within the same FXU, and three cycles to the other FXU. However, if a 2-cycle FXU operation is issued and then rejected for any reason, the subsequent times it is issued, the UniQ will attempt to wake up instructions in the other half of the queue assuming a 2-cycle issue-to-issue latency. If a dependent instruction in the other half of the queue is issued too early, it will be rejected in the EXE stage because the result of the 2-cycle instruction is not available.

10.1.13.14 Issue-to-Issue Latencies

Table 10-14 on page 232 shows the minimum issue-to-issue latencies for different classes of instructions. There are some exceptions, as noted in the comments at the end *Table 10-14*. A brief explanation of the acronyms follows:

PM	VMX/VSX permute
XS	VMX/VSX simple operations
FX	All instructions reading from or writing to the FPSCR and VSCR
VX	VMX complex operations

FD	VSX scalar; BFU floating-point
VD	Vector double-precision floating-point
VS	VMX/VSX 4-way vector single-precision floating-point
DFU	Decimal floating-point
CY	Cypher / cryptographic instructions
VSU-FX	Move to/from FPSCR
FXU	Any operation issued to the fixed-point unit
LU	Any operation issued to the load unit, including simple fixed-point and store-data operations
LSU	Any operation issued to the load-store unit, including simple fixed-point operations

Table 10-14. Issue-to-Issue Latencies (Sheet 1 of 2)

From	To	Latency	Comments
PM/XS/FX	any	2 or 7	[1]
FD/VD	any besides FD/VD	7	
FD/VD	FD/VD	6	
CY	FD/VD/VX/VS	7	
CY	any besides FD/VD/VS/VX	6	
VS	any besides VS	7	
VS	VS	6	
DFU	any	13	[2]
VX	any	7	
VSX-to-GPR move	any fixed-point operation	4 or 5	[3]
GPR-to-VSX move	any VSX operation	5	
FP/VS/VMX load	any	5	
fixed-point load (no XER dest)	any	3	
fixed-point load (with XER dest)	any	4	
store w/ update	any	3	
FXU 1-cycle (no XER dest)	same FXU	1	[4]
FXU 1-cycle (no XER dest)	other FXU, either LU / LSU	2	
FXU 1-cycle (with XER)	same FXU	2	
FXU 1-cycle (with XER)	other FXU, either LU / LSU	3	
FXU 2-cycle (no XER)	same FXU	2	
FXU 2-cycle (no XER)	other FXU, either LU / LSU	2	

1. Normally permutes and VMX-simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted to a 7-cycle operation to prevent a resulting bus collision.
2. This applies to pipelined DFP operations only. It applies to only the DFP IOP of the cracked DFP instruction.
3. Latency of four cycles if VSU pipe 0(1) to FXU pipe 0(1). Otherwise five cycles.
4. See rules about back-to-back-to-back (*Section 10.1.13.9 Limitations of Back-to-Back* on page 229).
5. Minimum two cycles issue-to-issue. Subject to special rules: **cmp** and **bc** must be in the same dispatch group, or else the **cmp** must be the most recent CR writer.

Table 10-14. Issue-to-Issue Latencies (Sheet 2 of 2)

From	To	Latency	Comments
FXU 2-cycle (with XER)	same FXU	3	
FXU 2-cycle (with XER)	other FXU, either LU / LSU	3	
FX multiply	same FXU	4	
FX multiply	other FXU, either LU / LSU	5	
divw	same FXU	variable	
divw	other FXU, either LU / LSU	above +1	
mf CTR/LR/TAR/CR	FXU / LU / LSU	5	
cmp	bc	2	[5]
mtCTR/LR/TAR/CR	dependent CRQ operation	6	
mtCTR/LR/TAR	dependent branch	5	
branch updating CTR/LR/TAR	branch reading CTR/LR/TAR	3	
branch updating CTR/LR/TAR	mfCTR/LR/TAR	4	
CR-logical	CR-logical or mfCR	3	
CR-logical	branch	3	

1. Normally permutes and VMX-simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted to a 7-cycle operation to prevent a resulting bus collision.
2. This applies to pipelined DFP operations only. It applies to only the DFP IOP of the cracked DFP instruction.
3. Latency of four cycles if VSU pipe 0(1) to FXU pipe 0(1). Otherwise five cycles.
4. See rules about back-to-back-to-back (*Section 10.1.13.9 Limitations of Back-to-Back* on page 229).
5. Minimum two cycles issue-to-issue. Subject to special rules: **cmp** and **bc** must be in the same dispatch group, or else the **cmp** must be the most recent CR writer.

10.1.14 Pipeline Hazards

10.1.14.1 ISU Rejects

The issue queue can reject instructions for the following types of conflicts and speculation. Reject penalties are shown in parenthesis.

- Result bus arbitration. (5)
- Finish port arbitration. (5)
- Other resources internal to the execution units. (5)
  - FX and VS dividers (5)
  - DFP, crypto units (5)
  - SPR bus (5)
  - LSU prefetcher utilizes Agen logic (5)
- Instructions need to request the SAR bypass. (7 - 10)
- Instructions need to wait for a swap to be processed before they can issue. (7)
- Instructions were speculatively issued before their source operands were available. (6)

Conflicts Due to Write-Back Collisions and VSU/SPR Resources

The following resource conflicts result in a 5-cycle rejection loop:

FXU pipe:

- FX 2-cycle instruction followed by a 1-cycle FX instruction. The 1-cycle instruction gets rejected.
- FX 4-cycle instruction followed by one bubble followed by an FX 2-cycle instruction. The 2-cycle instruction gets rejected.
- FX 4-cycle instruction followed by two bubbles followed by an FX 1-cycle instruction. The 1-cycle instruction gets rejected. (This case is frequently, but not always, prevented by blocking 1-cycle FX instructions within UniQ from issuing three cycles after a multiply issues. They can still be issued off the dispatch bypass.)
- FX divide followed by another FX divide. The divide continues to issue and reject. (The next divide can issue three cycles before WB of the first).
- The CRQ issues an **mfCTR**/**LR**/**CR** that has a write-back collision with an add issued in the previous cycle on the same thread set or trying to use the same GPR write port. The **add** gets rejected.
- A multiply or VSU-extract collides with **mfCTR**/**LR**/**CR** on the same thread set or pipe: the multiply/extract wins and mfspr is not issued.
- An **mfCTR**/**LR**/**CR** collides with a 2-cycle operation; **mfspr** has priority.
- Any 4, 6, and 10-cycle **mfspr** colliding with 1, 2, 4, variable-length FX operations and VSU extracts. Depends on latency and which can be cancelled.
- A VSU extract is issued at the same time as a multiply on the same queue half. Only one can be issued. Toggle LFSR to determine priority.
- A VSU extract is issued at the same time or up to four cycles before a FX multicycle operation. The multicycle operation is rejected.
- A VSU extract is issued causing WB collisions with FX 1- or 2-cycle operations. The FX operation is rejected.
- In SMT-shared mode, the GPR SAR bypass must arbitrate with FX instructions.
  - For 1- and 2-cycle FX instructions, priority between the issuing instruction and SAR bypass is determined by an LFSR that can be controlled to give priority to the SAR bypass 25, 50, 75, or 87% of the time. The default recommendation is to leave this at 75%.
  - For multiplies, the SAR bypass has priority if both of the following are true:

    (1) The LFSR above indicates that the SAR bypass should win over the FX operation.

    (2) Previous SAR bypass attempts have been killed.

    Note: This means that if there are many multiplies attempting to do a SAR bypass, it takes a long time to make forward progress because the SAR bypass must have failed once before.
  - Divides always have priority over SAR bypass. The SAR bypass is killed if it collides with a divide WB.
- If two **mfspr**s or **mtspr**s are issued at the same time on any FXU or CRU issue port, one must be cancelled. A priority LFSR determines which is cancelled. (This is an SPR bus, not a GPR write port limitation.)

VSU pipe:

- Write-back contention between the following classes of pipelined instructions: 13-cycle (DFP), 7-cycle (FD/FX/VD/VS/VX/CY), 2-cycle (XS/PM/SD/SQ).

  Note: If a 2-cycle operation is issued five cycles after a 7-cycle operation, the 2-cycle operation is converted to a 7-cycle operation. This means it has an issue-to-issue latency of seven cycles, and finishes at the same time as a 7-cycle operation.

- CY issued to both pipes; one must be rejected (LFSR determines priority).
- DFP issued to both pipes; one must be rejected (LFSR determines priority).
- Swap colliding with a VSX load in SMT-shared mode. The load wins and the swap is killed. The instruction dependent on the swap will issue and request to use SAR bypass.
- For up to three cycles after issuing a multi-cycle instruction, an attempt is made to issue any ready VS instructions and then reject them.

LSU/LU pipe:

- LU/LSU uses a pipe for the D-ERAT reload; an operation issued at the same time is rejected.
- LU/LSU pipe is used by data prefetch. (LSU sends back a "steal agen FX only" signal.) A simple-FX operation issued at the same time is canceled or rejected
- LU only: A resulting bus conflict due to L2 data coming back for a GPR or VSX load; load issued to the LU at the same time is rejected.
- LSU sends back XER results for **stcx** or **icswx**. Conflicting instruction on the LSU pipe gets a 6-cycle reject.
- Store forwarding: (LU/LSU sends a "delay5" indication) conflicting instruction (issued five cycles after the load/**mfspr** getting store forwarding) gets a reject.
- Same operation issued to both LSU and FXU (for FX->LS); no rejecting is required in this case; the LSU issue is cancelled and the operation is issued to the FXU.
- An **mtspr** and **mfspr** issued to both LSU pipes at the same time; one must be rejected.
- In SMT-shared mode, loads issued to the LSU will block swaps, causing them to be cancelled.

SAR Bypass - Related Rejects

Instructions can be issued even if they do not have valid source rtags (that is, it has a "needs rtag" indicator), but their source operand data is in the SAR. When this happens, the instructions request a SAR bypass. Due to the time it takes to read data out of the SAR, the instructions requesting a SAR bypass are rejected by the issue queue and re-issued after the data as been placed in bypass latches.

- If an instruction is issued with a "needs rtag" bit for a source operand, and the GPR data mux was not configured to use the SAR bypass, then the instruction requests a SAR bypass and gets a reject.
  - If it wins arbitration of the SAR bypass, it gets an 8, 9, or 10-cycle reject, depending on if the instruction needs 1, 2, or 3 SAR bypasses.
  - If the instruction does not win SAR bypass arbitration, it gets a 7-cycle reject.
- If the front end of the issue queue thinks the instruction must get data out of the SAR (that is, the instruction is issued with a "needs rtag" bit on) and the GPR SAR bypass controls was set up correctly but the data was not read out of the SAR due to a SAR bank collision, or if the wrong data was read out of the SAR, the instruction must get a 6-cycle reject. On the FX/LS/LU pipes, this can happen if an instruction

was flushed, and another one with the same queue position (qpos) was dispatched in its place, and a bypass latch was reserved for that qpos. This case does not apply to the VS pipe.

- If an instruction is *not* issued with a "needs rtag", but the GPR source mux had been configured to get data from the SAR, the instruction must get a 5/6-cycle reject. Like the previous case, this one only happens if there is a false qpos match due to quick qpos reuse, and only applies to the FX, LS, and LU pipes.
- If an instruction requests SAR bypass and then successfully uses it or gets flushed, and a second instruction is dispatched into the same qpos, a false SAR bypass can be driven if the SAR bypass latch contents are no longer valid.
- If an FX or VS instruction gets both a resource conflict and a SAR reject, it has a reject latency according to the SAR reject (7 - 10 cycles).
- If an LS or LU instruction gets both a resource conflict and a SAR reject, it has a reject latency according to the resource conflict (5 cycles).

AMC Castout-Related Rejects

If an instruction's source was recently cast out of the AMC, because either a completion castout or a swap victim castout, there is a window of time where the instruction will get rejected. This mechanism is in place because, if a given rtag X is used in a swap, it is not possible to determine if instructions with source rtag X are dependent on the swap victim or the swap grantee. For VRF castouts, there is a 4-cycle window where dependents of the rtag being cast out get rejected. For GPR castouts, the window is 6 cycles.

AMC/SAR Swap-Related Rejects

The front end of the UniQ can attempt to issue instructions that are waiting for swaps if they are issued in the disp+3, disp+4, or disp+5 cycles. These instructions must be rejected, because if they were not, they can complete before the swap was processed. This can potentially cause errors in the AMC. Therefore, at issue time, check to see if the issued instructions should be waiting for swaps. If so, they get a 5-cycle reject.

XER ARF Related Rejects

If an instruction with an XER source has its data in the XER ARF, but the XER source rtag of that instruction matched one of the XER destination rtags of recently issued instructions, the instruction with the XER source will be rejected. The reason for this is that it is not known until late in the ISS+1 cycle if the XER data resides in the ROB or the ARF. The DL bits and XER bypass controls are dependent on this information. They are computed as if the false rtag matches are true dependences.

FPSCR Related Rejects

VS instructions reading a renamed FPSCR can get a source-not-ready reject if the FPSCR producer has not successfully issued.

An FPSCR consumer also gets a reject if the producer completed in the window of time between when the consumer was dispatched and issued. This is because the location of the FPSCR data (ARF or ROB) is not known until ISS+2.

10.1.14.2 LSU Rejects

The LSU detects several internal conditions that cause it to reject an instruction on a given port. When the LSU rejects an instruction, it causes a 10-cycle turn-around for the instruction to be re-issued. A list of conditions that cause an instruction to be rejected follows:

- SRQ - load-hit-store same cycle (store is older): The younger load is rejected.
- SRQ - load-hit-store (different group, store forwarding not possible): Load is rejected if the data needed by the load is not fully contained in the store data queue (SDQ) for the store instruction.
- SRQ - load-hit-store (data not available): Load is rejected because the data to be stored by the store instruction is not yet available in the store data queue.
- SRQ - load-hit-sync: Load is younger and therefore rejected because **sync** has to complete first.
- SRQ - load-hit-DCB (**dcbz**, **cidcbf**): Load is younger and hits the cache line targeted by the DCB instruction but is rejected because the DCB instruction must complete (**dcbz** could have forwarded '0', but the POWER8 processor does not do that).
- SRQ - **lwarx**/**ldarx** not next-to-complete: If a previous **lwarx**/**ldarx** instruction sets the reservation, a successive **lwarx** is rejected unless it is next-to-complete. This prevents excessive loss of reservation and still guarantees forward progress.
- Global - force reject for hang detect: When the pervasive unit sends the hang detect signal, all load/store instructions are rejected by the LSU.

10.1.14.3 Flush Conditions

Pipeline flushes are the last and most expensive means of reordering instructions in the out-of-order section of the processor. Instructions that have not completed are eligible to be flushed. Most flushes are a result of late detection of an architecturally unsafe ordering of instructions. Others are rare conditions that are most easily solved by flushing when they occur yet allow the processor to achieve higher performance under the more frequent circumstances. *Table 10-15* lists all of the flush conditions.

Table 10-15. Flush Conditions (Sheet 1 of 2)

Flush Condition	Special Considerations	Programming Ramifications
Cache inhibited and not single instruction group or cache inhibited per page table I bit.	Force to single instruction group and wait until next-to-complete to make nonspeculative per architecture.	Avoid in performance critical code.
Guarded storage, L1 cache miss and not single instruction group.	Force to single instruction group and wait until next-to-complete to make nonspeculative per architecture.	Avoid in performance critical code.
Unaligned	Unaligned references not handled directly in hardware need to reference microcode. Instruction first flushed to single instruction group and then flushed to microcode routine entry point.	Avoid unaligned data in performance critical regions. See *Section 10.1.15.1 Storage Alignment* on page 238 for unaligned cases not handled directly by hardware.
ECC error on reload data	Data forwarded assuming no error. Load must be restarted.	None
Invalidate-hit-reload	Store or invalidate while load miss outstanding to the same address. Load must be restarted.	Use LARX/STCX for shared data in performance critical regions.
Load-hit-store EA alias	Because EA(44:51) is different, hardare does detect. Flush until store drains.	Avoid EA aliasing in shared memory.

Table 10-15. Flush Conditions (Sheet 2 of 2)

Flush Condition	Special Considerations	Programming Ramifications
Load-hit-store, same group, cannot forward	Load data not contained in single-store data-queue entry.	Code-specific optimizations. Can detect and replace by register move if performance limiting.
Store-hit-load	Issued out-of-order, older store to same address as younger load. Flush after store to restart load.	SHL avoidance in hardware when RA fields match. See *Section 10.1.5 Store-Hit-Load Avoidance Table* on page 217.
Load-hit-load with snoop	Issued out-of-order, older load to same address after intermediate snoop after younger load.	Use LARX/STCX for shared data in performance critical regions.
LARX-hit-LARX	Issued out-of-order, older LARX detects younger LARX. Only one LARX per thread can be outstanding at a time. Need to flush younger LARX.	On a fail, software must spin on a load until the value changes.
LRQ atomic	First or second part of an LQ or LQARX instruction has been invalidated. Architecturally required to be atomic.	Use LARX/STCX for shared data in performance critical regions.
RMSC flush	HV = '1', DR = '0', not single instruction and ERAT miss. Force to single instruction group and wait until next-to-complete to make nonspeculative per architecture.	Avoid in performance critical code.
sync flush for speculative load that has a matching snoop	Performance optimization that allows speculation beyond a **sync**. Flush is used to recover when speculation must be backed out.	None.
TM load with cache line crossing where the second address cannot be sent to the L2.	Performance optimization allowing cache line crossings to be faster. Because both addresses must be seen by the L2 if both are L1 cache hits, anything blocking the second address from making it to the L2 cache must cause a flush.	Avoid unaligned cache line crossings in performance sensitive code.

10.1.15 Level-1 Data Cache

10.1.15.1 Storage Alignment

The LSU performs most loads and stores that are unaligned with the same timing as naturally aligned loads and stores with some exceptions (see *Section 3.6.6.3 Storage Access Ordering* on page 85). When these cases occur, a misaligned flush is generated, which causes a refetch of this instruction, and the microcode unit generates multiple instructions that will now cross the boundary without flush.

For simple loads (not string instructions), the group with the unaligned load or store is flushed and re-executed with microcode that avoids the boundary crossing. If the load or store was the first instruction in the group, only one flush is needed. If not the first instruction in a group, the first flush causes the instruction to come back in a single instruction group, and a second flush is needed for the misaligned flush.

String instructions are already microcoded as single instruction groups; thus only one flush is needed.

Misaligned flushes tend to be around 30 cycles each for the best case.

Special Load Cache Line Crossing Cases

If a load instruction crosses a cache line with a data access and both cache lines are in the L1 data cache, it is highly likely that the access will have only a 5-cycle penalty over a normal L1 cache access. In the POWER8 processor, all loads that are allowed to cross a cache line can get this treatment with any byte alignment. The implementation saves the bytes from the first cache line access. Then five cycles later, it accesses the second cache line and merges data from the first cache line with the data from the second cache line. When this occurs, an issues slot is taken for the second access.

There is one hazard that exists between 16-byte VSX loads on LUx pipes and a load on an LSx pipe, where the x's are the same (meaning the same pipe half) when these two instructions occur in the same address generation cycle. If the sum of the bytes from the first cache-line access of the VSX instruction and the bytes from the first cache-line access of the load[1] are greater than 15, the load in LSx is rejected. This is because there are not enough resources to save more than 15 bytes for each pipe half.

10.1.15.2 Special Case of Store Crossing a 64-Byte Boundary

If a store crosses a 64-byte boundary, when the store drains from the store queue, a 1-cycle penalty occurs compared to normal store drains. This is because it takes two write cycles to write across a 64-byte boundary as compared to one write cycle for all other stores.

10.1.16 Level-1 Data ERAT

In the POWER8 processor, there are two logical primary D-ERATs implemented as four physical D-ERATs. There is one physical ERAT for each of the two load-store units and one physical ERAT for each of the two load units. Logically, the two primary D-ERATs for LSU0 and LU0 always contain the same data and the D-ERATs for LSU1 and LU1 always contain the same data. Each primary D-ERAT is implemented as a fully associative 48-entry array, with modified binary LRU replacement algorithm. D-ERAT entries are created for 4 KB, 64 KB, or 16 MB pages only. 16 GB pages are broken into 16 MB pages in the D-ERAT, where the installed page contains the referenced address. Similarly, 1 MB pages are broken into 64 KB pages where the installed page contains the referenced address. The four individual ERATs operate in one of two modes: shared or split. In single-thread mode and SMT2, the contents of all four D-ERATs are identical; this is referred to as shared mode. But in SMT4 or SMT8 mode, two ERATs (one LSU and one LU) contain translation for half of the active thread (two threads for SMT4 mode and four threads for SMT8 mode). In split mode, the two paired ERATs both contain addresses that can be different from the other paired ERATs. In split mode for the POWER8 processor, the two ERAT pairs can be loaded at the same time (with different data) for the following cases:

1. Both ERAT pairs can be loaded from the secondary ERAT.
2. One ERAT pair can be loaded from the secondary ERAT, and one ERAT pair can be loaded from the TLB.
3. One ERAT pair can be loaded from the secondary ERAT, and one ERAT pair can be loaded from memory (tablewalk data).

The POWER8 D-ERAT supports hit-under-miss. Up to four D-ERAT misses are entered in the ERAT miss queue (EMQ) per cycle, one per LSU pipe. There are eight total EMQ entries. These eight entries arbitrate for access to the secondary ERAT. The secondary ERAT contains 256 entries managed in two halves, with a round-robin LRU algorithm. An instruction that misses the primary ERAT but hits the secondary ERAT receives a reject on its initial ERAT lookup and a hit on its second lookup. Similar to the primary ERAT, the secondary ERATs operates in either split or shared mode. In split mode, each of the halves of the secondary

1. This must be an integer load and thus cannot be bigger than 7 bytes in this case.

ERAT is dedicated to half of the threads. The two halves for the secondary ERATs correspond to the ERAT pairs in the primary ERAT. The secondary ERAT can process two misses concurrently in split mode but only one miss for shared mode. If an address misses the secondary ERAT, it is sent to the SLB and TLB. The EMQ entries are the only eight D-ERAT misses that are sent to the TLB, others result in a reject. These eight entries and two I-side misses arbitrate for the SLB and TLB sequentially. If there is a hit in both, the information for the D-ERAT is returned to both the primary and secondary ERATs. TLB hits are handled speculatively, and the I-ERAT and D-ERAT are loaded speculatively in this case.

10.1.17 Level-2 Data ERAT

The secondary D-ERAT is the second level of non-instruction translation caching in the core. When a data reference misses the primary ERAT, it looks up the address translation in the secondary D-ERAT. If the translation is found in the secondary D-ERAT, it is then loaded into the primary D-ERAT. If the translation is not found in either the primary or the secondary D-ERAT, the request then has to check the SLB and TLB.

The best-case reload time for a hit in the secondary D-ERAT is to have the translation present in the primary D-ERAT nine cycles after the initial look-up. This best-case timing is fast enough to ensure that the translation is installed in the primary D-ERAT before the instruction is re-issued by the issue queue, because the fastest time that the issue queue can re-issue an instruction is greater than nine cycles. When an instruction misses both the primary and secondary D-ERAT, the translation is installed in both. However, because the replacement policies are managed independently, it is possible for a translation to be aged out of the secondary D-ERAT while still remaining in the primary D-ERAT.

The secondary D-ERAT consists of four 64-entry fully associative CAM arrays, for a total of 256 entries. In single thread mode, all 256 entries are available for that thread. In split SMT mode, the secondary D-ERAT is treated as two 128-entry arrays. The secondary D-ERAT replacement policy is a simple FIFO scheme.

10.1.18 Translation Look-Aside Buffer

The TLB has 2048 entries and is four-way associative. It uses a true LRU replacement algorithm. The TLBs are indexed with a hashed address calculated from portions of the virtual address and the page size. Each entry in the TLB represents a particular page size: 4 KB, 64 KB, 1 MB, 16 MB, and 16 GB (that is, all page sizes are natively supported in the TLB). Server software does not have any plan to exploit the 1 MB page sizes; however, to the processor, 1 MB pages are always enabled.

The real address is returned to both the primary and secondary D-ERAT on a TLB hit. The primary and secondary ERATs have different LRU algorithms; therefore, the secondary ERAT is not guaranteed to contain the same address translations contained in the primary ERATs. The TLB contains entries for both the I-cache and D-cache, while the secondary ERAT only contains entries for the D-cache. The I-cache and D-cache maintain separate primary ERATs. Therefore, the D-side ERAT contains entries only for the D-cache, and similarly the I-side ERAT contains entries only for the I-cache. A TLB hit reloads the primary (and secondary) ERAT after two rejects. A following ERAT lookup then hits in the primary ERAT.

If the TLB request does not hit in the TLB, a tablewalk is initiated that loads the TLB, secondary ERAT, and primary ERAT with the translation for the instruction that generated the TLB miss. Up to four outstanding tablewalks can be active at one time. The implementation allows tablewalks for speculative instructions but does not update the Ts or C bit in a PTE entry unless the instruction is NTC when the PTE entry is found. The TLB is reloaded with the corresponding PTE entry even if the instruction is speculative.

10.1.19 Load Miss Queue

The Load Miss Queue (LMQ) is a 16-deep queue that handles all data cache misses for the LSU, including data prefetches to the L1 cache. These 16 entries are shared by all eight threads on this core. The queue is 4-ported; thus, it can take a miss from each one of the four load/store pipes, and can pass those four requests to the L2 cache in one cycle as well. The two LU ports (load only ports) have a fast path to the L2 cache. Even then, if two demand loads occur in the same cycle on the LUs, only one can win, and the other is placed in the L2 queue, causing a minimum of a 3-cycle penalty. The LS ports, take a minimum 3-cycle penalty to L2 latency, because they must be queued before they can enter the L2 pipe.

Data is returned from the nest in two 64-byte beats or four 32-byte beats. The critical beat (the beat with the requested data) is favored to return first. Normally, two 64-byte beats are returned back-to-back if the line was in the L2 cache and two 64-byte beats from a line that was in the L3 cache, return with a bubble (cycle) in between. If the data came from the power bus, the data returns in four 32-byte beats, with at least one bubble in between each beat.

Merging into the LMQ is not allowed for loads. In other words, if two loads for the same line are executed, the first gets a miss queue entry, and the second one is rejected until the second load hits the cache after the data has been refilled from the nest. One exception is that a load can always merge with an L1 prefetch if they are for the same line.

If a flush occurs after a load has made an L2 request, the load does not forward data, but the miss queue remains busy until all data is returned and placed into the cache.

Cache-inhibited loads are handled by the LMQ as well. Although they do not write to the cache, they still occupy an entry until data is returned.

If a store with the same real address as a load in the LMQ occurs after the load request but before the data is back, the LMQ does not allow that data to be written into the cache.

If a data cache miss request finds the data in memory, a special tag is returned indicating that the ECC check has not been done on this beat of data. The memory controller reserved the right to cancel this line on the last beat of data if the ECC checker finds an error. This allows the memory controller to source data much earlier because it does not have to go through the ECC checker first. The LMQ must mark any load that uses this line as speculative when that load finishes. The global completion table logic must not complete these loads until the LMQ indicates that the data is good.

10.1.20 Transactional Memory

The L2 cache is largely responsible for tracking the transactional memory (TM) footprint. The TM footprint consists of 128-byte cache lines that have been accessed by either load or store instructions while the thread is executing in the transactional state. Generally in rollback-only-transactions (ROTs), only stores are tracked. However, due to implementation-specific reasons, a small fraction of loads in an ROT can also be included in the tracking footprint. The footprint is tracked by four banks of 16-entry CAMs, which are shared by all threads on a given chiplet. A footprint overflow type of transaction failure results if either the CAM or the L2 congruence class capacity is exceeded. If either a transactional or non-transactional access initiated by another thread conflicts with a given thread's footprint, the transaction also fails and the appropriate type of conflict is reported in the TEXASR per the Power ISA.

If a tlbie request from any thread in the same LPAR hits a page used in a transaction, the transaction is required to fail. Rather than exactly comparing all TM pages to a **tlbie**, a bloom filter[1] approach is used.

10.1.21 Store Queue and Store Forwarding

The LSU contains a 40-entry store reorder queue (SRQ) that holds real addresses and a 40-entry store data queue (SDQ) that holds a quadword of data. These store queues are dynamically shared among the available threads (in both SMT2, SMT4, and SMT8 modes). In the POWER8 processor, each SDQ entry can hold one store operation, which can be up to 16 bytes wide (for VSX and VMX stores, or a **stq**, **stqcx.**).

Store addresses are loaded into the SRQ when the instruction is issued. These can be issued in any order. Fixed-point data is loaded into the SDQ from the FXU after the data is accessed from the GPR (st_data is transferred via the LU0 or LU1 RB operand bus). Floating-point data is loaded into the SDQ from a shared 16-byte data bus from the FPU after the data is accessed from the FPR. This shared 16-byte data bus also transfers VSX/VMX store data to the SDQ from a dedicated bus from the FPU after the data is accessed from the FPR. In the POWER8 processor, there are two dedicated 16-byte data buses to transfer FPU/VSX/VMX data to the SDQ. Stores are removed from the SRQ and SDQ and written to the cache in program order after all the previous instructions are committed.

Loads that are issued, which hit (address match) stores in the SRQ that have not been written to the cache, are candidates for store forwarding. A store forward can occur under the following conditions:

- The data is available in the SDQ.
- The load is completely contained within the store (load byte count is less than or equal to the store byte count).
- The store is older than the load.
- The page-table I-bit is not set.
- No collision with an L1 reload operation (AGEN steal).

There are some exceptions to these rules; for example, see *Section 10.5.1.1 Store Quadword* on page 264 for more details. If the above conditions are not met, the load instruction can be either rejected or flushed. A load that becomes a store forward operation is treated as a miss from the standpoint of the issue logic. The latency of a store forward operation is five cycles.

In addition to stores, **sync**, **lwsync**, **ptesync**, **eieio**, **dcbz**, **dcbf**, **icbi**, **tlbsync**, and **tlbie** (non-local) are installed in the SRQ. Additionally, on the POWER8 processor, these instructions will install in the SRQ: **isync**, **tm_begin**, **tm_check**, **tm_end**, **pbt.**, **pbt**, and **logmpp**.

10.1.21.1 Stores in Real Mode (MSR[DR] = 0)

For store instructions, the store re-order queue above the caches is used as a temporary holding spot for both the address and the data. When the store passes the completion point, the store queue is marked to allow the store data to be written into any of the appropriate caches. If the real-mode I-bit (located in the HID4 Register) is set, cache misses are expected, so that the store simply works its way towards memory or memory-mapped I/O. Note that, as a result, the store action is only observed once per store instruction.

1. This can cause a transaction to fail when it did not need to, but greatly reduces the overhead of detection for what is a rare event.

10.1.22 Data Prefetch

The purpose of the data prefetch mechanism is to reduce the negative performance impact of increasing memory latencies, particularly on technical workloads. These programs often access memory in regular, sequential patterns. Their working sets are also so large that they often do not fit into the limited size L1, L2, and L3 caches used in the POWER8 processor. For additional information, see *Table 10-17 Instruction Latencies and Throughputs* on page 246.

Designed into the load-store unit, the prefetch engine can recognize sequentially increasing or decreasing accesses to adjacent cache lines and then request anticipated lines from more distant levels of the cache/memory hierarchy. The usefulness of these prefetches is reinforced as repeated demand references are made along such a path or stream. The depth of prefetch is then increased until enough lines are being brought into L1, L2, and L3 cache that much or all of the load latency can be hidden. The most urgently needed lines are prefetched into the nearest cache levels. During stream start up, several lines ahead of the current demand reference can be requested from the memory subsystem. After steady state is achieved, each stream confirmation causes the engine to bring one additional line into the L1 cache, one additional line into the L2 cache, and one additional line into the L3 cache. To effectively hide the latency of the memory access while minimizing the potentially detrimental effects of prefetching such as cache pollution, the requests are staged such that the line that is being brought into the L3 cache is typically several lines ahead of the one being brought into the L1 cache. Because the L3 cache is much larger than the L1 cache, it can tolerate the most speculative requests more easily than the L1 cache can.

A basic prefetch start-up sequence follows.

Prefetch begins by saving the effective address of the L1 D-cache misses in a 16-entry queue, offset up or down by one line address (the initial miss to line *n* triggers the allocation of an entry for line *n+1*). This initial allocation is managed on a pseudo-LRU basis, and causes a request for line *n+1* to be brought into the L3 cache. A subsequent demand L1 lookup for line *n+1* matches the newly allocated entry (referred to here as a confirmation). This confirmation triggers an update to the existing queue entry, which now points to line *n+2*. In addition, the confirmation triggers a request for lines *n+2,3,4,5* to be brought into the L3 cache. A request is also made for line *n+2* to be brought into the L1 and L2 cache. Upon the next confirmation (demand L1 lookup for line *n+2*), the queue entry is again updated, this time to point to line *n+3*. This confirmation triggers requests for lines *n+6,7,8,9,10* to be brought into the L3 cache. Line *n+3* is also brought into the L1 and L2 cache. A third confirmation (demand L1 lookup for line *n+3*) triggers requests for lines *n+11,12,13,14,15* to be brought into the L3 cache, as well as requests for lines *n+4,5* to be brought into the L1 and L2 cache.

At this point, the L3 prefetch engine is prefetching 12 lines ahead of the current demand load, The L1 prefetch engine is prefetching two lines ahead of the demand load, and the stream has reached steady state. Subsequent confirmations do not cause the L3 engine to go any further ahead than 12 lines or the L1 engine to go any further ahead than two lines. A fourth confirmation (demand L1 lookup for line *n+4*) triggers a request for line *n+16* to be brought into the L3 cache, as well as a request for line *n+6* to be brought into the L1 and L2 cache. Because the L3 prefetch engine is now in steady state, this request for line *n+16* can happen immediately or it can be queued up to take advantage of efficiencies in the memory subsystem. Either way, each subsequent confirmation eventually triggers a request for one additional line to be brought into the caches.

Although this description represents a typical stream ramp profile, the maximum depth of the stream and the urgency with which the prefetch engine attains that depth are fully programmable via the Data Stream Control Register (DSCR). The DSCR is described in Book II of the Power ISA.

The direction of the stream is always assumed to be increasing in magnitude. However, during stream start-up, the prefetch engine simultaneously examines streams for both increasing and decreasing patterns using a "shadow" queue. Subsequent references can either confirm the increasing direction or the decreasing direction. If the stream is confirmed by a decreasing pattern, the direction of the stream is marked as such from that point forward.

In addition to the stream detection described previously, the prefetch engine is augmented by a stride-N detection engine. The purpose of the stride-N engine is to detect streams that have regular access patterns, but which do not fetch from consecutive cache lines in memory. The hardware can detect strides up to 8 KB in length, with a 32-byte granularity. The detection is handled in a 4-entry buffer that examines the stride between the current cache miss and the previous four cache misses. When a pattern is detected, a stream is created in the regular prefetch queue. From this point forward, the stream is treated just like any other stream, with the difference being subsequent prefetch requests can fetch from nonconsecutive cache lines.

Prefetch streams are tracked by the effective address and are allowed to cross small and medium memory page boundaries, but will be invalidated when crossing a large-page boundary. All prefetch requests must therefore go through address translation before being sent to the memory subsystem. When address translation is not found in the ERAT for a prefetch request, the prefetch initiates an ERAT miss request. This has the effect of prefetching not only the data that is needed by the program, but also the address translation. If any type of exception is encountered while performing the translation lookup for the prefetch, the request is dropped and the requesting stream is invalidated. Streams otherwise remain active until replaced with a new address by the allocation mechanism, or until the thread that allocated the stream is no longer running on the processor.

In all cases, a prefetch request is completed when the lines are found already resident in or have been returned to the target cache level. If a demand miss "catches up with" an outstanding prefetch request, it is either merged or rejected depending on the level in the cache hierarchy. If it is rejected, it is retried until it hits in the cache.

10.2 Chiplet

10.2.1 Level-2 Cache

Each L2 cache on the POWER8 processor core is a unified cache for its respective core. The L2 cache is an 8-way associative 512 KB cache with fast access to its own private 8 MB L3 cache region through a private low latency bus. The L2 cache maintains full hardware coherence within the system and can supply intervention data to the other cores on this POWER8 chip or to other cores on other POWER8 chips. Logically, the L2 cache is an in-line cache. Unlike the L1 caches, which are store-through, it is a store-in cache. It is fully inclusive of the L1 D-caches and the L1 I-caches.

Each L2 cache is comprised of 512 associative sets (called congruence classes); each congruence class contains eight 128-byte cache lines. Real address bits 48:56 are used in the 9-bit congruence class address.

The L1 D-cache is a store-through design. As such, when a cacheable store is removed from the SRQ, if it hits in the L1 D-cache, the line is updated. In addition, all stores (hit or miss) are forwarded on to the L2 cache. If the store is a miss in the L1 D-cache, the data is not written to the D-cache and the line is not reloaded from the L2 cache (no L1 D-cache allocate on store miss).

10.2.2 Level-3 Cache

Each L3 cache region on the POWER8 chip is a unified victim cache for its respective core/L2 cache, as well as for other L3 caches on chip. The resident cache lines installed from the attached L2 cache are referred to as L3.0 lines, and the resident cache lines installed from other L3s are referred to as L3.1 lines. When castout from this L3 cache, L3.1 lines go to memory and L3.0 lines have the option of being castout to other L3 caches on chip. The L3 cache is an 8-way associative 8 MB cache. The L3 cache maintains full hardware coherence within the system and can supply intervention data to the other cores on this POWER8 chip or to other cores on other POWER8 chips. Logically, the L3 cache is an in-line cache. The L3 cache is a victim cache of the L2 cache (that is, all valid lines that are victimized in the L2 cache are castout to the L3 cache). The L3 cache is not inclusive of the L2 cache. Each L3 cache is comprised of 8192 associative sets (called congruence classes), each congruence class contains eight 128-byte cache lines. Real address bits 44:56 are used in the 13-bit congruence class address.

10.3 Latencies

10.3.1 Cache Latencies and Bandwidth

Table 10-16 lists several key bandwidth and latency values for the chip. These represent best case values under ideal conditions. Actual values can be somewhat higher due to resource limitations or queueing effects. Still, these value are useful in understanding system performance.

Table 10-16. Cache Latencies and Bandwidth

Description	Latency	Bandwidth
L2 D-cache load hit (bypass)	8.5 pclks	64 bytes/pclk
L2 I-cache load hit (bypass)	9.5 pclks	64 bytes/pclk
L3 load hit	26.5 pclks	32 bytes/pclk
L2.1 load hit	94 pclks	16 bytes/pclk
L3.1 load hit	112 pclks	16 bytes/pclk
L2.5 load hit	325 pclks	
L3.5 load hit	343 pclks	
Memory load chip pump	80 ns	230 GB/sec (2:1 read:write)
Memory load system pump	140 ns	
A link		25.6 GB/s (peak)

Note: Pclks represent one processor clock.

10.3.2 Instructional Latencies and Throughputs

The fixed-point latency, when specified as x-y, is the latency where x is for two instructions on the same thread-set side and y is for two instructions on opposite thread-set sides.

Table 10-17. Instruction Latencies and Throughputs (Sheet 1 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
ld ldx ldu ldux lwa lwax lwaux lwz lwzx lwsu lwzux lha lhax lhau lhaux lhz lhzx lhzu lhzux lbz lbzx lbzu lbzux	LSU or LU, FXU	3 cycles (RT) 2 - 4 cycles (RA)	2/cycle	1	N/A	Broken into a basic load, an **exts**, and an **add**.
lq	LSU, FXU	3 (RT) 3 (RT+1) 6 (XER)	1/cycle	1	N/A	
stb sth stw std	LSU, LU	N/A	2/cycle	1	no	Stores are internally issued twice (once as an ea_gen operation, and once as a data operation). Stores put their data into the STQ, which later writes to the cache/memory hierarchy (the STQ supports forwarding for many cases).
stbx sthx stwx stdx	LSU, LU	N/A	2/cycle	1	no	Three source operands are cracked to minimize breadth or renaming facility.
stbu sthu stwu stdu	LSU, LU	3 cycles for updated register	2/cycle	1	no	
sthbrx stwbrx	LSU, LU	N/A	2/cycle	1	no	Cracked into a **store**_agen and **store_data** IOP.
stbux sthux stwux stdux	LSU, LU	3 cycles for updated register	2/cycle	1	N/A	Cracked into two IOPs.
stq	LSU, LU	N/A	1/cycle	1	N/A	Cracked into three IOPs.
lmw lmd	LSU or LU	N/A	2 register loads per cycle after start-up	1	N/A	Number of dispatch groups is equal to the number of registers modulo three. The microcode generates inline sequence of basic loads.
stmw stmd	LSU, LU	N/A	2 register stores per cycle after star-tup	1	N/A	Number of dispatch groups is equal to number of registers modulo three. The microcode generates inline sequence of basic stores.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 2 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
lswi lsdi (naturally aligned)	LSU or LU	N/A	2 register loads per cycle after start-up	1	N/A	Number of dispatch groups is approximately equal to number of registers modulo three. The microcode assumes natural alignment and generates inline sequence of basic loads.
lswx lsdx (naturally aligned)	LSU or LU	N/A	2 register loads per cycle after start-up	1	N/A	Number of dispatch groups is approximately equal to number of registers modulo three plus four more for startup. The microcode assumes natural alignment and generates inline sequence of basic loads.
stswi stsdi (naturally aligned)	LSU, LU	N/A	2 register stores per cycle after start-up	1	N/A	Number of dispatch groups is approximately equal to number of registers modulo three, Microcode assumes natural alignment and generates inline sequence of basic stores.
stswx stsdx (naturally aligned)	LSU, LU	N/A	2 register stores per cycle after start-up	1	N/A	Approximately equal to (number of registers modulo three plus four more for startup. The microcode assumes natural alignment and generates inline sequence of basic stores.
lswi lsdi lswx lsdx stswi stsdi stswx stsdx (unaligned)	LSU,LU	N/A	N/A	1	N/A	String instruction is first decoded and dispatched as described previously. At execute, the LSU notes that it is unaligned and causes a *machine flush*. As the string instruction goes through decode the second time, it is broken up in a way that takes the misalignment into account.
lwarx ldarx	LSU or LU	N/A	N/A	1	no	Forced to miss data L1 cache.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 3 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
stwcx. stdcx.	LSU, LU	N/A	N/A	1	no	Must establish coherency block ownership before completing the instruction (other stores do not have to do this). Can take anywhere from 10 cycles to 100's depending on the state of the coherency block in the memory hierarchy.
addi addis add add. subf subf. addic subfic adde addme subfme addze. subfze neg neg. nego	FXU (or LU or LSU for non-dot forms)	1 - 2 cycles (GPR) 2 cycles (XER), 5 cycles (CR)	6/cycle, 2/cycle (with XER or CR updates)		no	
addic. adde. subfe. addme. subfme. addze. subfze.	FXU	1 - 2 cycles (GPR, 5 - 6 cycles (CR)	1/cycle		no	Cracked into a basic **add (sub)** and a **cmp**.
addo.subfo.addeo subfeo addmeo subfmeo addzeo subfzeo nego.	FXU	1 - 2 cycles (GPR), 2 cycles (XER), 5 cycles (CR	1/cycle		no	These operations might architecturally change the summary overflow bit. Summary overflow is written at completion time. If change is detected, everything younger is flushed.
addeo. addmeo. subfmeo. addzeo. subfzeo.	FXU	1 - 2 cycles (GPR), 2 - 3 cycles (XER), 6 - 7 cycles (CR)	2/3 cycles		no	These operations might architecturally change the summary overflow bit. Summary overflow is written at completion time. If change is detected, everything younger is flushed.
addo subfo	FXU	1 - 2 cycles (GPR) 2 cycles (XER)	2/cycle		no	These operations might architecturally change the summary overflow bit. Summary overflow is written at completion time. If change is detected, everything younger is flushed.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 4 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
addc subfc	FXU	1 - 2 cycles (GPR)	2/cycle		wait for non-ren ame scoreboard bit to clear	
addc. addco addco. subfc. subfco subfco.	FXU	1 - 2 cycles (GPR) 5 - 6 cycles (CR) 2 cycles (XER)	1/cycle		wait for non-ren ame scorebo ard bit to clear	
isel	FXU	5 cycles (GPR)				
mulli	FXU	4-5 cycles	2/cycle		no	Pipelined.
mullw mulhw mulhwu	FXU	4-5 cycles	2/cycle		no	Pipelined.
mulld mulhd mulhdu	FXU	4-5 cycles	2/cycle		no	Pipelined.
mullwo mulldo	FXU	4-5 cycles (GPR) 5 cycles (XER)	2/cycle		no	Pipelined in FXU. These operations might architecturally change the summary overflow bit. The first attempted execution of these instructions assume that the SO-bit does not change. If it does, the instruction causes a flush and then is re-executed.
mullw. mulld. mulhd. mulhw. mulhdu. mulhwu.	FXU	4 - 5 cycles (GPR) 8 cycles (CR)	2/2cycles		no	Pipelined in FXU. Cracked into baseline operation and a **cmp**.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 5 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
mullwo. mulldo.	FXU	4 - 5 cycles (GPR) 8 cycles (CR)	2/2cycles		no	Pipelined in FXU. Cracked into baseline operation and a **cmp**. These operations might architecturally change the summary overflow bit. The first attempted execution of these instructions assumes that the SO-bit does not change. If it does, the instruction causes a flush and then is re-executed.
divd divdu divwe divweu	FXU	12 - 23 cycles (GPR)	2/12 cycles to 2/23 cycles		no	Actual latency depends on data. Non-divides can be issued underneath.
divw divwu	FXU	12 - 15 cycles (GPR)	2/12 cycles to 2/15 cycles		no	Actual latency depends on data. Non-divides can be issued underneath.
divde divdeu	FXU	14-41 cycles	2/14 cycles to 2/40 cycles		no	Actual latency depends on data. Non-divides can be issued underneath.
divd. divw. divdu. divwu. divde. divwe. divdeu. divweu.	FXU	same as above for GPR + 6 cycles for CR	same as above		no	Non-divides can be issued underneath.
divdo divwo divduo divwuo divdeo divweo divdeuo divweuo	FXU	same as above for GPR, +1 for XER	same as above		no	These operations might architecturally change the summary overflow bit. The first attempted execution of these instructions assumes that the SO-bit does not change. If it does, the instruction causes a flush and then is re-executed.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 6 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
divdo. divwo. divduo. divwuo. divdeo. divweo. divdeuo. divweuo.	FXU	same as above for GPR, +6 cycles for CR	same as above		no	These operations might architecturally change the summary overflow bit. The first attempted execution of these instructions assumes that the SO-bit does not change. If it does, the instruction causes a flush and then is re-executed.
cmpi cmp cmpii cmpl cmpla	FXU	5 cycles (CR)	2/cycle		no	
tdi twi td tw txer (XBI field refers to renamed bits in Xer)	FXU	N/A	2/cycle		no	
selii (XBI field refers to renamed bits in Xer)	FXU	3 - 4 cycles	1/cycle		no	
andi. andis. ori oris xori xoris and and. or or. xor xor. nand nand. nor nor. eqv eqv. andc andc. orc orc.	FXU/LSU	1 - 2 cycles (GPR) 6 cycles (CR)	6/cycle 2/cycle (with XER or CR destination)		no	
ori 0,0,0 (preferred NOP)	none	0 cycles	6/cycle		no	A special form of this instruction is completed at the same time that it is dispatched.
extsb extsb. extsh extsh. extsw, extsw.	FXU	1 - 2 cycles (GPR) 4 - 5 cycles (CR)	2/cycle	1 when rc = 1	no	
cntlzd cntlzd. cntlzw cntlzw.	FXU	3 cycles (GPR) 6 - 7 cycles (CR)	2/cycle	1 when rc = 1	no	
rldicl rldicl. rldicr rldirc. rldic rldic. rlwinm rlwinm. rldcl rldcl. rldcr rldcr. rlwnm rlwnm. rldimi rldimi.	FXU	1 - 2 cycles (GPR) 5 - 6 cycles (CR)	2/cycle 1/cycle when rc = 1	1 when rc = 1	no	

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 7 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
rlwimi	FXU	1 cycle (GPR)	2/cycle		no	
rlwimi.	FXU	1 - 2 cycles (GPR) 5 - 6 cycles (CR)	1/cycle		no	
sld sld. slw slw. srd srd. srw srw. sradi srawi srad sraw	FXU	1 - 2 cycles (GPR) 5 - 6 cycles (CR)	1/cycle	1 when rc = 1	no	
sradi. srawi. srad. sraw.	FXU	1 - 2 cycles (GPR) 5 - 6 cycles (CR)	1/cycle		no	
dtcs.	FXU	2 cycles	1/cycle		no	
mtspr(xer)	FXU (one)	N/A	N/A		wait for non-ren ame scoreboard bit to clear	Can only be done by one of the FXU pipelines.
mtspr(**lr**) **mtspr**(**ctr**)	FXU (one)	5 cycles	1/cycle		no	Can only be done by one of the FXU pipelines.
mtspr(**others**) **mtmsr mtmsrd**	Either FXU or LSU (depends on SPR)	varies based on SPR	varies based on SPR		wait for non-rename scoreboard bit to clear	Can only be done by one of the FXU pipelines.
mfspr(**xer**)	FXU (one)	8 cycles	N/A		wait for non-rename scoreboard and bit to clear	The FXU pipeline is blocked while waiting for SPR value.
mfspr(**lr**) **mfspr**(**ctr**)	CRLogic	5 cycles	1/cycle		no	The FXU pipeline is blocked for one cycle.
mfspr(others) **mfmsr**	FXU (one) LSU (one), or CR logic	varies based on SPR	varies based on SPR		wait for non-ren ame scoreboard bit to clear	For FXU, IFU, and perva-sive-owned SPRs, the FXU pipeline is blocked while waiting for the SPR value.
mtcrf (one field) **mtocrf**	FXU (one)	N/A	1/cycle		no	
mtcrf (more than one field)	FXU (one)	N/A	N/A		no	
mfcr	CRlogic	N/A	N/A		no	

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 8 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
mfcrf	CR logic	3 cycles	1/cycle		no	
mftb	FXU (one)	~10 cycles	approxi-mately one per 10 cycles		wait for non-rename scoreboard bit to clear	Use of the FXU pipeline is blocked while waiting forthe SPR value.
lfs lfsx lfd lfdx lxsdx lxvd2x lxvw4x lxvdsx	LU	5 cycles	2/cycle	1	no	Instructions are executed in the LU unit. The FPU just arranges renames.
lfsu lfsux lfdu lfdux lxsdux lxvd2ux lxvw4ux	LU, FXU	5 cycles (FPR) 1 - 2 cycles (GPR)	2/cycle	1	no	Cracked into normal **load** and an FXU **add**.
lvebx lvehx lvewx lvx lvxl	LU	5 cycles	2/cycle	1	no	
stfs stfsx stfd stfdx	LSU, FPU	N/A	2/cycle	1	no	Instructions are dispatched to both the FPU and the LSU units.
stfsu stfsux stfdu stfdux	LSU, FPU, FXU	3 cycles for GPR	2/cycle	1	no	LSU.
stfiwx	LSU, FPU	N/A	2/cycle	1	no	
stxsdx stxvd2x stxvw4x	LSU, FPU	N/A	2/cycle	1	no	Instructions are dispatched to both the FPU and the LSU units.
stxsdux stxvd2ux stxvw4ux	LSU, FPU, FXU	1- 2 cycles for GPR	2/cycle	1	no	Cracked into a normal **store** and a FXU **add**. Normal **store** instructions are dispatched to both the FPU and the LSU units.
stvebx stvehx stvewx stvx stvxl	LSU, FPU, FXU	1 - 2 cycles for GPR	2/cycle	1	no	Cracked into a normal **store** and a FXU **add**. Normal **store** instructions are dispatched to both the FPU and the LSU units.
sync	LSU	N/A	N/A		wait until it is next to complete and for all prior load data to come home	Forces previous stores to finish into the cache/memory hierarchy (out of the STQs).

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 9 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
ptesync	LSU	N/A	N/A		wait until it is next to complete and for all prior load data to come home	Forces previous stores to finish into the cache/memory hierarchy (out of the STQs).
lwsync	LSU	N/A	N/A		wait until it is next to complete and for all prior load data to come home	Forces previous stores to finish into the cache/memory hierarchy (out of the STQs). Still broadcasts sync transaction onto the SMP interconnect (but does not block).
eieio	LSU	N/A	N/A		wait until it is next to complete and for all prior load data to come home	
isync	ls/st	N/A	N/A		wait until it is next to complete and for all prior load data to come home	Causes a flush and instruction refetch only if it is preceded by instructions that changes the content of the machine or **icbi** or **ptesync**.
icbi	LSU	N/A	N/A			After the LSU generates and translates the EA, the icbi looks like a snooped icbi to the instruction fetcher. Snooped icbi spins through 16 possible locations for the block (I-cache is indexed with effective address, so that aliasing can occur).
dcbt dbtst	LSU	N/A	1/cycle			The lead time between a **dcbt** and a subsequent load is equal to the load-to-use-latency of the level of cache hierarchy the line will be coming from.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 10 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
dcbz	LSU	N/A	1/cycle			Invalidates L1 cache line on its way to the L2 cache. Allocation and zero function occur at the L2 cache (handled by the L2 cache and treated as any other store).
dcbst	LSU	N/A	1/cycle			
dcbf	LSU	N/A	1/cycle			
slbie	LSU	N/A				Causes index-based invalidate in both the I-ERAT and the D-ERAT.
slbia	LSU	N/A				Fully invalidates the SLB, the I-ERAT, and D-ERAT.
tlbie	LSU	N/A				Causes index-based invalidate in both the I-ERAT and the D-ERAT. Is broadcast onto the SMP interconnect.
tlbiel	LSU	N/A				Causes index-based invalidate in both the I-ERAT and the D-ERAT. Is not broadcast onto the SMP interconnect.
tlbsync	LSU	N/A				
slbmte	LSU	N/A			wait for non-rename scoreboard bit to clear	Causes selective invalidates out of the I-ERAT and D-ERAT.
slbmfev slbmfee	LSU	N/A			wait for non-ren ame scorebo ard bit to clear	
mtsr mtsrin mtsrd mtsrdin	LSU				wait for non-ren ame scorebo ard bit to clear	Causes selective invalidates out of the I-ERAT and D-ERAT.
mfsr mfsrin	LSU	3 cycles	1/cycle		No	
mffs mffs. mcrfs	FPU		1/cycle		No	Alone in a dispatch group, completion serialized.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 11 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
mtfsfi. mtfsf. mtfsb0. mtfsb1.	FPU	3 cycles to dependent FP operation	1/cycle		No	Alone in a dispatch group, completion serialized.
mtfsfi mtfsf mtfsb0 mtfsb1	FPU	3 cycles to dependent FP operation	1/cycle		No	Alone in a dispatch group.
fabs fadd fadds fcfid fcfids fcfidu fcfidus fcpsgn fctid fctidu fctiduz fctidz fctiw fctiwu fctiwuz fctiwz fmadd fmadds fmr fmsub fmsubs fmul fmuls fnabs fneg fnmadd fnmadds fnmsub fnmsubs fre fres frim frin frip friz frsp frsqrte frsqrtes fsel fsub fsubs	FPU	6 cycles	2/cycle		No	
fabs. fadd. fadds. fcfid. fcfids. fcfidu. fcfidus. fcpsgn. fctid. fctidu. fctiduz. fctidz. fctiw. fctiwu. fctiwuz. fctiwz. fmadd. fmadds. fmr. fmsub. fmsubs. fmul. fmuls. fnabs. fneg. fnmadd. fnmadds. fnmsub. fnmsubs. fre. fres. frim. frin. frip. friz. frsp. frsqrte. frsqrtes. fsel. fsub. fsubs.	FPU	6 cycles (FPR) 9 cycles (CR)	1/cycle		No	Alone in a dispatch group, completion serialized.
fdiv	FPU	32 cycles	2/26 cycles		No	Makes use of microcoded sequence within the floating-point unit. Free FPU slots can be interleaved with other FPU instructions. See *Table 3-4 Latencies of Floating-Point Divide/Square-Root Instructions* on page 66 for additional information.
fdivs	FPU	26 cycles	2/20 cycles		No	
fsqrt	FPU	43 cycles	2/37 cycles		No	
fsqrts	FPU	31 cycles	2/25 cycles		No	
fdiv. fdivs. fsqrt. fsqrts.	FPU	same as non-dot forms, +1 cycles for CR	1/2 of non-dot above		No	Alone in a dispatch group, completion serialized. See *Table 3-4 Latencies of Floating-Point Divide/Square-Root Instructions* on page 66 for additional information.
fcmpu fcmpo ftdiv ftsqrt	Vector simple integer unit (XS)	4 - 9 cycles (to CR)	2/cycle		No	

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 12 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
fmrgew fmrgow	Vector permute unit (PM)	2 cycles	2/cycle		No	Latency to seven cycles if WB conflict.
xsabsdp xsadddp xscpsgndp xscvdpspn xscvdpsxds xscvdpsxws xscvdpuxds xscvdpuxws xscvspdp xscvspdpn xscvsxddp xscvuxddp xsmaddadp xsmaddmdp xsmsubadp xsmsubmdp xsmuldp xsnabsdp xsnegdp xsnmaddadp xsnmaddmdp xsnmsubadp xsnmsubmdp xsrdpi xsrdpic xsrdpim xsrdpip xsrdpiz xsredp xsrsqrtedp xssubdp	FPU	6 cycles to FPU (+1 cycle to other VSU ops)	2/cycle		No	
xsaddsp xscvdpsp xscvsxdsp xscvuxdsp xsmaddasp xsmaddmsp xsmsubasp xsmsubmsp xsmulsp xsnmaddasp xsnmaddmsp xsnmsubasp xsnmsubmsp xsresp xsrsp xsrsqrtesp xssubsp	FPU	6 cycles to FPU (+1 cycle to other VSU ops)	2/cycle		No	
xvabsdp xvadddp xvcpsgndp xvcvdpsp xvcvdpsxds xvcvdpsxws xvcvdpuxds xvcvdpuxws xvcvspdp xvcvspsxds xvcvspuxds xvcvsxddp xvcvsxdsp xvcvsxwdp xvcvuxddp xvcvuxdsp xvcvuxwdp xvmaddadp xvmaddmdp xvmsubadp xvmsubmdp xvmuldp xvnabsdp xvnegdp xvnmaddadp xvnmaddmdp xvnmsubadp xvnmsubmdp xvrdpi xvrdpic xvrdpim xvrdpip xvrdpiz xvredp xvrsqrtedp xvsubdp	FPU	6 cycles to FPU (+1 cycle to other VSU ops)	2/cycle		No	

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 13 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
xvabssp xvaddsp xvcpsgnsp xvcvspsxws xvcvspuxws xvcvsxwsp xvcvuxwsp xvmaddasp xvmaddmsp xvmsubasp xvmsubmsp xvmulsp xvnabssp xvnegsp xvnmaddasp xvnmaddmsp xvnmsubasp xvnmsubmsp xvresp xvrspi xvrspic xvrspim xvrspip xvrspiz xvrsqrtesp xvsubsp	FPU	6 cycles to FPU (+1 cycle to other VSU ops)	2/cycle		No	
xsdivdp xvdivdp	FPU	32 cycles	2/26 cycles		No	Makes use of microcoded sequence within the floating-point unit. Free FPU slots can be interleaved with other FPU instructions.
xvdivsp	FPU	28 cycles	2/22 cycles		No	
xssqrtdp xvsqrtdp	FPU	43 cycles	2/37 cycles		No	
xvsqrtsp	FPU	32 cycles	2/26 cycles		No	
vaddfp vcfpsxws vcfpuxws vcsxwfp vcuxwfp vexptefp vlogefp vmaddfp vnmsubfp vrefp vrfim vrfin vrfip vrfiz vrsqrtefp vsubfp	FPU	6 cycles to FPU (+1 cycle to other VSU ops)	2/cycle			
xsmaxdp xsmindp xvcmpeqdp xvcmpeqsp xvcmpgedp xvcmpgesp xvcmpgtdp xvcmpgtsp xvmaxdp xvmaxsp xvmindp xvminsp	vector simple integer unit (XS)	2 cycles to all VSU ops	2/cycle			See note 3.
xscmpodp xscmpudp xstdivdp xstsqrtdp xvtdivdp xvtdivsp xvtsqrtdp xvtsqrtsp	vector simple integer unit (XS)	4..9 cycles (to CR)	2/cycle			
vcmpbfp. vcmpeqfp. vcmpequb. vcmpequh. vcmpequw. vcmpequd. vcmpgefp. vcmpgtfp. vcmpgtsb. vcmpgtsh. vcmpgtsw. vcmpgtsd. vcmpgtub. vcmpgtuh. vcmpgtuw. vcmpgtud. xvcmpeqdp. xvcmpeqsp. xvcmpgedp. xvcmpgesp. xvcmpgtdp. xvcmpgtsp.	vector simple integer unit (XS)	2 cycles to all VSU ops, 4..9 cycles to CR	2/cycle			See note 3.
mtvscr	vector simple integer unit (XS)		1/cycle			Completion serialized.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 14 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
mfvscr vaddcuw vaddsbs vaddshs vaddsws vaddubm vaddubs vadduhm vadduhs vadduwm vadduws vaddudm vand vandc vavgsb vavgsh vavgsw vavgub vavguh vavguw vclzb vclzd vclzh vclzw vcmpbfp vcmpeqfp vcmpequb vcmpequh vcmpequw vcmpequd vcmpgefp vcmpgtfp vcmpgtsb vcmpgtsh vcmpgtsw vcmpgtsd vcmpgtub vcmpgtuh vcmpgtuw vcmpgtud veqv vmaxfp vmaxsb vmaxsh vmaxsw vmaxsd vmaxub vmaxuh vmaxuw vmaxud vminfp vminsb vminsh vminsw vminsd vminub vminuh vminuw vminud vnand vnor vor vorc vpopcntb vpopcnth vpopcntw vrlb vrld vrlh vrlw vshasigmad vshasigmaw vslb vsld vslh vslw vsrab vsrad vsrah vsraw vsrb vsrd vsrh vsrw vsubcuw vsubsbs vsubshs vsubsws vsububm vsububs vsubuhm vsubuhs vsubuwm vsubuws vsubudm vxor	vector simple integer unit (XS)	2 cycles to all VSU ops	2/cycle			See note 3.
xxland xxlandc xxleqv xxlnand xxlnor xxlor xxlorc xxlxor	vector simple integer unit (XS)	2 cycles to all VSU ops	2/cycle			See note 3.
vpopcntd vadduqm vaddcuq vaddeuqm vaddecuq vsubuqm vsubcuq vsubeuqm vsubecuq	vector simple integer unit (XS)	4 cycles to all VSU ops	2/2 cycles			Cracked into two dependent XS instructions.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 15 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
vbpermq vgbbd vmrgew vmrghb vmrghh vmrghw vmrglb vmrglh vmrglw vmrgow vperm vpermxor vpkpx vpksdss vpksdus vpkshss vpkshus vpkswss vpkswus vpkudum vpkudus vpkuhum vpkuhus vpkuwum vpkuwus vsel vsl vsldoi vslo vspltb vsplth vspltisb vspltish vspltisw vspltw vsr vsro vupkhpx vupkhsb vupkhsh vupkhsw vupklpx vupklsb vupklsh vupklsw	Permute unit (PM)	2 cycles to all VSU ops	2/cycle			See note 3.
xxmrghw xxmrglw xxpermdi xxsel xxsldwi xxspltw	Permute unit (PM)	2 cycles to all VSU ops	2/cycle		No	See note 3.
vmhaddshs vmhraddshs vmladduhm vmsummbm vmsumshm vmsumshs vmsumubm vmsumuhm vmsumuhs vmulesb vmulesh vmulesw vmuleub vmuleuh vmuleuw vmulosb vmulosh vmulosw vmuloub vmulouh vmulouw vmuluwm vsum2sws vsum4sbs vsum4shs vsum4ubs vsumsws	VX (Vector complex)	7 cycles to all VSU ops	2/cycle			
vcipher vcipherlast vncipher vncipherlast vpmsumb vpmsumd vpmsumh vpmsumw vsbox	CY (Vector Crypto)	6 cycles, +1 to FPU and Complex	1/cycle			Crypto pipeline shared between both VSU pipes.
mfvsrd mfvsrwz		4 or 5 cycles	1/cycle			Four if on same pipe. Five to all.
mtvsrd mtvsrwa mtvsrwz		5				
bcdadd. bcdsub. dadd dadd. dsub dsub. dcmpo dcmpu dtstdc dtstdg dtstex dtstsf dquai dquai. dqua dqua. drrnd drrnd. drintx drintx. drintn drintn. dctdp dctdp. ddedpd ddedpd. denbcd denbcd. dxex dxex. dxexq dxexq. diex diex. dscli dscli. dscri dscri. dtstdcq dtstdgq dtstsfq	DFU	13 cycles	1/cycle			DFU pipeline shared between both VSU pipes.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 16 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
dcmpoq dcmpuq drrndq drrndq. diexq diexq. dquaiq dquaiq. dtstexq ddedpdq ddedpdq. denbcdq denbcdq. dscliq dscliq. dscriq dscriq. dctqpq dctqpq. drintxq drintxq. drintnq drintnq.	DFU	13 cycles +2 in VSU = 15 cycle	1/cycle			DFU pipeline shared between both VSU . Instruction is split in DFU and permute intruction.
daddq daddq. dsubq dsubq. dquaq dquaq.	DFU	2 in VSU +13 cycles +2 in VSU = 17 cyle	2/3 cycle			DFU pipeline shared between both VSU pipes. Instruction is split in DFU and two permute intruc-tions.
drsp drsp. ddedpd ddedpd. dctfix dctfix.	DFU	25 cycles	1/ 12 cycle			DFU pipeline shared between both VSU pipes.
dcffix dcffix.	DFU	32 cycles	1/ 19 cycle			DFU pipeline shared between both VSU pipes.
dcffixq dcffixq.	DFU	32 cycles +2 in VSU = 34 cycles	1/ 19 cycle			DFU pipeline shared between both VSU pipes. Instruction is split in DFU and permute intruction.
dmul dmul.	DFU	24+N cycles at least 28 =(28 to 40) cycles	1/ 28 to 40 cycle			DFU pipeline shared between both VSU pipes. Every BCD digit of the shorter operand (N) extends the execution time by one cycle.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

Table 10-17. Instruction Latencies and Throughputs (Sheet 17 of 17)

Instruction	Pipeline	Latency	Throughput (IPC)	Notes decode / dispatch	Other Interlocks	Other Comments
dmulq dmulq.	DFU	20+2N cycles at least 28 +4in VSU =(32 to 90) cycles	1/ 28 to 86 cycle			DFU pipeline shared between both VSU pipes. Instruction is split in DFU and two permute intructions. Every BCD digit of the shorter operand (N) extends the execution time by two cycles.
ddiv ddiv.	DFU	32 +4Q cycles =(32 to 96) cycles	1/32 to 96 cycle			DFU pipeline shared between both VSU pipes. Every BCD digit of the quotient (Q) extends the execution time by four cycles.
ddivq ddivq.	DFU	32 +4Q cycles +4 in VSU =(36 to 172) cycles	1/32 to 168 cycle			DFU pipeline shared between both VSU pipes. Instruction is split in DFU and two permute intructions. Every BCD digit of the quotient (Q) extends the execution time by four cycles.

1. Store_agen IOPs are issued to the LSU, and store_data IOPs are issued to the LU. Fixed-point loads that do not have an XER destination can be issued to either the LU or the LSU. Loads with floating-point destinations can only be issued to the LU, and loads with XER sources can only be issued to the LSU.
2. Branches can issue two cycles after a producing a **cmp** instruction if they were in the same dispatch group, or the **cmp** is the youngest instruction writing a CR relative to the **cmp**. Otherwise, the issue-to-issue latency to a branch is three cycles.
3. Normally, permutes and simple operations have a latency of two cycles. However, if a 2-cycle operation is issued five cycles after a 7-cycle operation on the same VSU pipe, the 2-cycle operation is dynamically converted into a 7-cycle operation to prevent a resultant bus collision (to make use of the issue slot).

10.4 PCI Express Performance

10.4.1 Bandwidth

On the POWER8 processor, the PCIe host bridge is integrated into the chip and upgraded to ×16 PCI Express 3.0 ports.

10.4.2 Latency

Time to first byte latency calculations are approximately 220 ns when doing a DMA read without TCEs.

10.4.3 Cluster Latency 2K Message

Sending a 2K message from one cluster to another cluster using an IB adapter is 2611 ns.

10.4.4 I/O Bandwidth per Pin

Two PCIe ×16 buses on POWER8 support 28 GB/s using 142 I/O pins (0.19 GB per pin).

10.4.5 PCIe Performance Goals

PCIe ×16 bus is able to achieve 87.5% efficiency thus obtaining 14 GB/s bandwidth bi-directional (14.0 GB/s DMA writes and 14.0 GB/s DMA reads).

- PCIe payload of 512 bytes on memory writes
- PCIe payload of 256 bytes on read completions
- Assume no TCE miss
- Assume no RTC cache miss
- Assume no MSI
- Assume 64K packet operation

10.5 Performance Specific Instructions

10.5.1 Store Multiple and Store String

Store multiple and store string are processed much like load multiple and load string. Each of these instructions is microcoded with one store operation generated for each register. In a store string, only some bytes of the last register can be stored. This is also handled with a single operation of 1 - 7 bytes. For store string immediate, the first microcode group contains only NOPs. Each of the store operations results in an st-agen issue and an st-data issue. Each operation uses one SRQ entry. The store operations can execute in any order, although they are written from low to high address as maintained by the order of SRQ entries. When a group consists of four store operations, two are queued for each LSU pipe. In this way, two store operations can execute in each cycle. Many store multiple or store string instructions can require multiple groups of internal operations. Each group completes in order, without waiting for later groups to finish. Consequently, if the instruction is flushed, all groups including those that have completed, are redone. This means that these stores could be seen more than once by the Nest.

Store string indexed instructions are processed like load string indexed. The first microcode group contains an XER read and an address add. The next several microcode groups are entirely nops to fill the decode pipeline. At this point, the length is known and the correct number of stores is generated with up to six stores per group.

Unaligned string instructions occur when an individual store crosses a 4 KB page boundary. This causes the entire instruction to be flushed to be decoded again to avoid the boundary crossing. Each original store now is placed in its own group, consisting of a left shift, two stores, and a NOP. The two stores each store from 1 - 7 bytes of data left aligned in the register. The first microcode group consists of nops. For store string indexed, the first group contains an address add. There is no need to wait for XER in the unaligned case.

10.5.1.1 Store Quadword

Store quadword (stq) is a 2-way split on the POWER8 processor core because only a single st-agen and two st_data operations are used: store upper, store lower, nop, nop. The first store operates as a normal store; st-gen goes through an LSU pipe, and the st_data goes through an LU pipe. The Tag bit in XER is read by the st-agen operation of the store upper. The second store is a st_data only, and it goes through an LU pipe. A single SRQ entry is used to hold the store quadword. After the store quadword is completed, it goes out as a single store to the L1 cache (and updates it if necessary) and to the L2 cache.

In the POWER8 core, MSR[LE] = 0 mode stq data is allowed to store-forward to a load other than a **larx**, **ltptr**, **lve**, **lq** and **lqarx**. The **lq** or **lqarx**'s first doubleword always gets its data and tag bit from the cache.

An **stq**, however, does not store forward to any load, if any thread is running in MSR[LE] = 1 mode.

10.5.1.2 eieio

The **eieio** instruction is in a single instruction group. **eieio** is held at dispatch until the LMQ is drained and there are no LSU instructions in the issue queues. Cache-inhibited loads are rejected when an older PTESync is in the SRQ. The **eieio** is sent to the nest like a normal store.

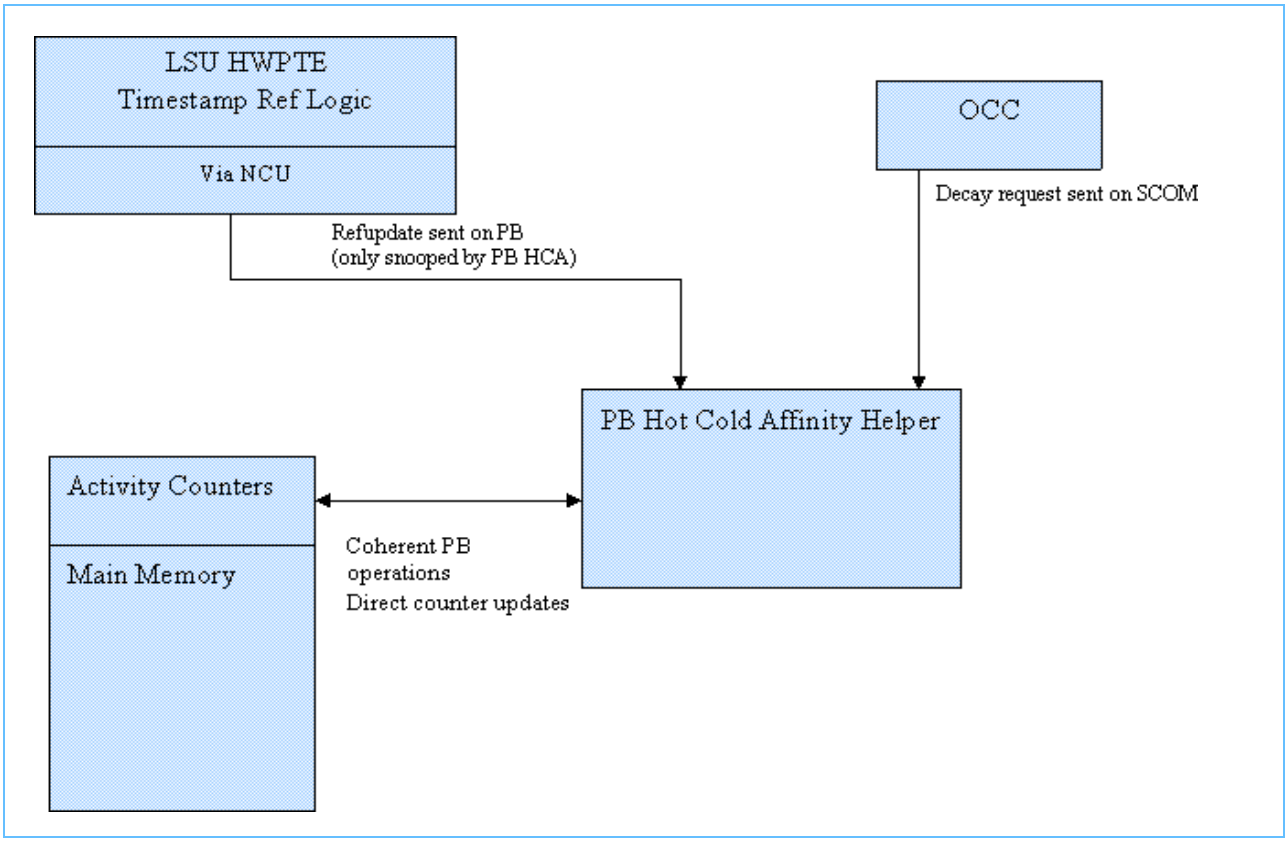
10.6 Other Topics

10.6.1 Hot/Cold Page Affinity Support

The POWER8 SMP interconnect Hot/Cold/Helper (HCA) is part of a hardware support system to enable efficient isolation of activity characteristics of memory pages. By monitoring hardware access to memory, it can gather information about the following:

- Activity rate: Enable power efficient memory management.
- Affinity information: Collect sharing pattern, enable operating-system-based page migrations.
- Long term access history: Maintain history.

Figure 10-1. Basic Building Blocks



LSU HWPTE sends a request on the PB to control the HCA. These requests are:

Reference update request. This command is only snooped by a specific HCA that has a matching value in the Base Address Register (BAR). There is a 32-byte payload; however, only the timestamp part is needed. The HCA then issues a reference update to the memory controller. The hpc_ref_updt operates on a single 4-byte data field in a 128-byte cache line, aligned on an 4-byte address boundary.

OCC Send Decay Request via SCOM

The decay request contains an address, range and a delay. The HCA then issues decay updates on the PB passing the address. The range indicates how many decay update commands to send. This works on a cache line in the counter area in the memory controller.

Activity Counters in Memory

The memory controller receives the new PB commands (activity update, reference update, and decay update) and updates the counters in memory.

PB Hot/Cold/Affinity Helper

Contains logic to snoop each memory reference in the PB. BAR registers are used to identify the memory location and range that belongs to this chip. The HCA contains a cache of the counter for each 4 KB memory page. It increments these counters as it detects a memory reference. HCA only works on 4 KB pages. Only the memory PB is snooped. HCA does not look at the combined response. The cache of

these counters is not the full counter, but only 12 bits. As these counters fill, an activity update command on the processor bus is sent to the main memory to update the actual counter. The counter cache does not contain all possible 4 KB counters for this chip. As new references are detected, a random LRU algorithm is used to determine which line to cast out to memory to make space for this new counter. HCA will castout assuming memory is a 4 KB page size. Even if memory is at a different page size, the counters are mapped at a 4 KB granularity.

The following ttypes are snooped by the HCA:

- All read groups
- rwitm
- dclaim, dcbz
- cl_dma_wr
- bsr_cp
- cp_m, cp_t, cp_tn, htm_cl_w
- cp_ig_me
- cl_w_cln
- dma_pr_w
- dcbf, dcbfl
- bkill
- armw
- armwf
- ci_pr_rd
- dma_pr_rd
- ci_pr_w
- ci_pr_ooo_w
- cl_cln
- bsr_get

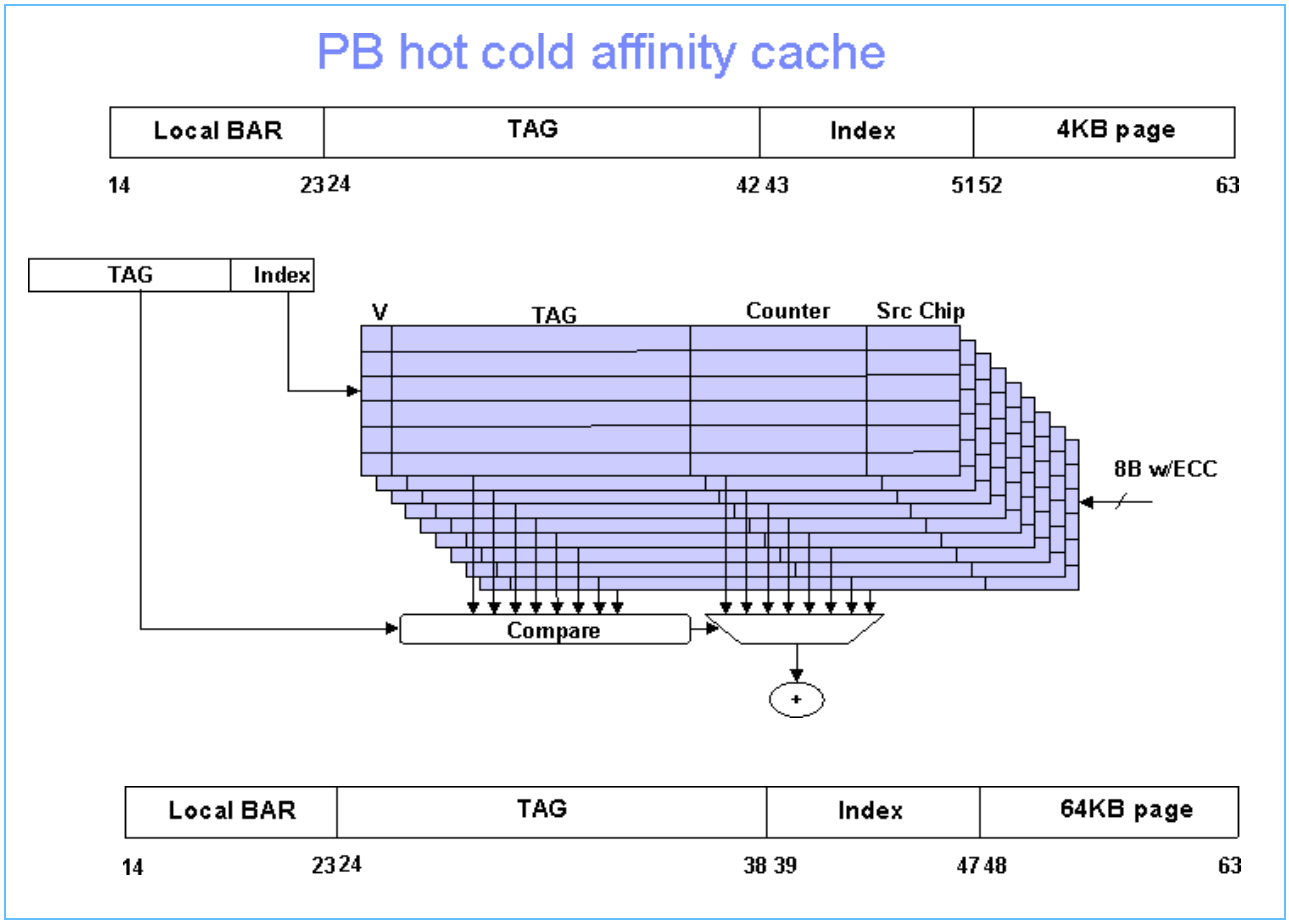
The following commands are issued by the HCA:

- Activity update: operands (counter address, count value, source chip)
- Reference update: operands (counter address, PTE time)
- Decay update: operands (counter region)
- rd_go_s: to obtain HPC on chip scope.

To reduce the bandwidth requirements for these updates, there is a mode to only send the 16th or 32nd hca.update. In such a case, the count is multiplied by 32 or 16 accordingly. However, this leads to inaccuracy in the count because not all hca.updates have the same count.

Figure 10-2 shows the HCA cache.

Figure 10-2. HCA Cache



Glossary

ABIST	Array built-in self test
AES	Advanced Encryption Standard
AMC	Architected mapper cache
ARF	Architected register file
ASIC	Application-specific integrated circuit
BA	Base address
BFP	Binary floating-point
BFU	Binary floating-point unit
BHT	Branch history table
BIST	Built-in self-test
BMC	Baseboard management control
BR	Branch register unit
BTAC	Branch target address cache
CAI	Coherent accelerator
CAM	Content-addressable memory
CEC	Central electronics complex
CI	Cast-in
CIABR	Current Instruciton Address Breakpoint Register
CIR	Chip information register
CIU	Core interface unit
CLB	Cache load buffer (IBuffer)
CMOS	Complementary metal–oxide–semiconductor
CO	Cast-out
CR	Condition Register
CRC	Cyclic redundacy check
DAWR	Data Address Watch Register
DDR	Double data rate
DECFP	Decimal floating-point unit

DFP	Decimal floating-point
DFU	Decimal floating-point unit
DIMM	Dual in-line memory module
DMA	Direct memory attach
DMI	Differential memory interface
DPLL	Digital phase-locked loop
DTS	Digital thermal sensor
EADIR	Effective address directory
EAT	Effective address translation
ECC	Error correcting code
ECID	Electronic chip identification
ECO	Extended cache option
ECRC	End-to-end cyclic redundancy check
EDI	Elastic differential I/O
EEH	Enhance error handling
EEPROM	Electrically erasable programmable read-only memory
EFK	East Fishkill
EMQ	ERAT miss queue
ERAT	Effective-to-real address translation
ESID	Effective segment identifier
FBC	SMP interconnect controller
FCPBGA	Flip-chip plastic ball grid array
FIFO	First-in, first-out
FIR	Fault Isolation Register
FLOPs	Floating-point operations per second
FPR	Floating-point register
FPSCR	Floating-Point Status and Control Register
FPU	Floating-point unit
FXU	Fixed-point units

GCM	Galios counter mode
GCT	Global completion table
GFW	Global firmware
GHV	Global history vector
GPE	General purpose engine
GPR	General purpose register
GPS	Global Pstate
GPST	Global Pstates table
HCSL	Host clock signal level
HDEC	Hypervisor decrementer
HMER	Hypervisor Maintenance Exception Register
HMI	Hypervisor maintenance interrupt
HRMOR	Hypervisor Real Mode Offset Register
HSS	High Speed Serial
HTM	Hardware trace monitor
ICA	Instruction cache access
ICP	Interrupt control presenter
ICS	Interrupt controller source
IEEE	Institute of Electrical and Elctronics Engineers
IFAR	Instruction fetch address register
IFU	Instruction fetch and decode unit
IMA	In memory accumulate
IOP	Internal operation
IPC	Instruction per cycle
IPL	Interrupt presenter layer; orinitial program load
IRL	Interrupt routing layer
ISU	Instruction sequencing unit
LBIST	Logic built-in self test
LHR	Load-hit-reload

LMQ	Load miss queue
LPAR	Logical partition
LPID	Logic partition ID
LPST	Local Pstate table
LRDIMM	Load-reduced dual in-line memory module
LRQ	Load reorder queue
LRU	Least-recently used
LSI	Level signalled interrup
LSSD	Level-sensitive scan design
LSU	Load store units
LU	Load-only unit
MBA	Memory buffer asynchronous
MCI	Memory channel interface
MCS	Memory controller synchronous
MDS	Memory domain status
MHCRO	Model hardware correlation ring oscillator
MPG	Multi-protocol gateway
MSI	Message signalled interrupt
NaN	Not a number
NCU	Noncacheable unit
NPS	Nap Pstate
NTC	Next-to-complete
OCC	On-chip controller
OCTS	On-chip thermal sensor
OEM	Original equipment manufacturer
OHA	On-chiplet hardware assist
PAPR	Power Architecture Platform Reference
PB	Processor bus
PC	Pervasive core unit

PCR	Processor Compatibility Register
PE	Partitionable endpoints
PEC	PCI Express controller
PEC	PCI Express controller
PID	Process ID
PLL	Phase-locked loop
PMC	Power management control
POR	Power-on reset
PRQ	Prefetch request queue
PSPB	Problem-state priority boost
PSRO	Performance sort-ring oscillator
Pstate	Performance state
PURR	Processor Utilization Resource Register
PVR	Processor Version Register
QNaN	Quiet Not a number
qpos	Queue position
RAIM	Redundant array of independent memory
RAS	Reliability, availability, and serviceability
RAW	Read after write
RC	Root complex
RCD	Register clock driver
RDIMM	Registered dual in-line memory module
RMSC	Real mode storage control
RNG	Random number generator
ROB	Re-order buffer
SAO	Strong access ordering
SAR	Second-level Architected Register
SBE	Self-boot engine
SCM	Single-chip module

SCOM	Scan communications
SDQ	Store data queue
SECDED	Single-error correction, double-error detection
SEEPROM	Serial electrically erasable programmable read-only memory
SHA	Secure hash algorithm
SHR	Store-hit-reload
SIMD	Single-instruction, multiple-data
SLB	Segment lookaside buffer
SMP	Symmetric multiprocessing
SMT	Simultaneous multithreading
SOI	Silicon-on-insulator
SPI	Serial peripheral interconnect
SPIVID	Serial Peripheral Interface - Voltage ID
SPR	Special purpose register
SPS	Sleep Pstate
SPURR	Scaled Processor Utilization Resource Register
SRAM	Static random access memory
SRQ	Store reorder queue
ST	Single thread
STAG	Storage tag
SVIC	Slave VME interface controller
TCE	Translation control entry
TDP	Thermal design point
TEXASRU	Transaction Exception And Summary Register Upper
TID	Thread ID
TLB	Translation lookaside buffer
TLP	Translation layer packet
TM	Transactional memory
TOD	Time of day

UE	Uncorrectable error
UniQ	Unified issue queues
VLE	Variable length encoding
VMX	Virtual machine extensions
VPD	Vital product data
VPN	Virtual page number
VRF	Vector scalar register file
VRM	Voltage regulator module
VRMA	Virtualized real mode area
VS	Vector scalar
VSCR	Vector Status and Control Register
VSU	Vector and scalar unit
VSX	Vector-scalar extension
WAW	Write after write
WPS	Winkle Pstate
XER	Fixed-Point Exception Register